

Department of Creative Informatics
Graduate School of Information Science and Technology
THE UNIVERSITY OF TOKYO

Master Thesis

**AI-Assisted Embedding of Natural-Language
Instructions into Racket with Runtime-Aware AI
Execution**

自然言語指示の Racket 言語への AI 支援つき埋め込みと実行時の値を
考慮したAIによる実行

Yugu Xie
謝 雨谷

Supervisor: Professor Shigeru Chiba

January 2026

Abstract

Large language models have enabled developers to describe programming tasks in natural language and receive generated code in return. However, current AI-assisted coding tools still face challenges. They sometimes modify code outside the region developers intend to change, they rely only on static source code without access to runtime values, and they provide no early warning when natural language instructions are ambiguous.

This thesis presents Promo, a system that embeds natural language instructions directly into the Racket programming language. Developers write instructions enclosed in square brackets within ordinary Racket code, and the Promo interpreter translates these instructions into executable code using a large language model at runtime. Promo tries to address the three challenges through specific mechanisms. Position control ensures that generated code replaces only the bracketed instruction without modifying surrounding code. Runtime information captures variable values during execution and includes them in the prompt so the model can see actual data structures and available functions. An ambiguity checker implemented as a language server inspects instructions and warns developers when an instruction is too vague.

Experiments on a dataset of 48 Promo programs evaluate these features. Position control achieved consistency with edit boundaries. Providing runtime information improved the overall success rate. The ambiguity checker showed a correlation between instructions flagged with vagueness “warning” and lower execution success rates.

概要

大規模言語モデルの発展により、開発者は自然言語でプログラミングタスクを記述し、AIにコードを生成させることが可能になった。しかし、現在のAI支援コーディングツールにはいくつかの課題がある。開発者が意図した編集範囲を超えてコードを変更してしまうことがあり、静的なソースコードのみを参照するため実行時の変数の値やデータ構造にアクセスできず、自然言語の指示が曖昧な場合でも事前に警告が出ない。

本論文では、自然言語の指示をRacketプログラミング言語に直接埋め込むシステムPromoを提案する。開発者は通常のRacketコード内で角括弧に囲んだ指示を記述し、Promoインタプリタは実行時に大規模言語モデルを用いてこれらの指示を実行可能なコードに変換する。Promoは三つの仕組みで上記の課題に対処する。位置制御により生成コードは括弧内の指示のみを置き換え、周囲のコードは変更されない。実行時情報の機能により、実行中の変数の値をプロンプトに含めることで、モデルが実際のデータ構造や利用可能な関数を把握できるようになる。曖昧性検査は言語サーバとして実装され、指示が曖昧な場合にエディタ上で開発者に警告を表示する。

48個のPromoプログラムを用いた実験により各機能を評価した。位置制御は編集境界を正しく守ることができた。実行時情報を提供することで全体の成功率が向上した。曖昧性検査による警告が出た指示は実行成功率が低い傾向があることが示された。

Contents

Chapter 1	Introduction	1
Chapter 2	Coding in natural language	4
2.1	AI Coding	4
2.2	Motivation	9
Chapter 3	Embedding Natural Language into Racket	14
3.1	Execution of Promo Programs	15
3.2	Position Control	17
3.3	Runtime Information	20
3.4	Ambiguity Checker	24
3.5	Implementation Details	26
Chapter 4	Experiments	32
4.1	Dataset Construction	32
4.2	Position Control	33
4.3	Runtime Information	36
4.4	Ambiguity Checker	38
Chapter 5	Conclusion and Future Work	40
5.1	Summary	40
5.2	Future Work	41
	Publications and Research Activities	42
	References	43

Chapter 1

Introduction

Large language models have made it possible to describe programming tasks in natural language and let AI generate corresponding code. This capability has been integrated into many development tools, from code completion systems to chat-based assistants. Developers can now write a short English description and receive executable code in return. The appeal is obvious since natural language is more accessible than formal syntax, and expressing intent directly can be faster than writing code manually. A developer can start from an idea, describe it, and quickly get a draft implementation. They can also ask for refactors, bug fixes, or small feature additions. In practice, this workflow often happens inside an editor. The developer selects code, writes an instruction, and expects the tool to change only that part.

However, current tools that support this workflow still have limitations. When a developer uses an AI assistant to modify a specific region of code, the assistant sometimes changes code outside the intended region. The tools also work with static information only. They cannot access runtime values like the actual contents of a variable or the structure of data loaded from a file. Additionally, natural language is ambiguous itself. When an instruction is vague, the generated code may not match what the developer actually wants. These problems lead to extra work. Developers must review changes carefully, provide more context manually, and iterate on prompts until the result is acceptable.

Position mistakes are disruptive because they can break existing code that the developer did not want to change. The lack of runtime context is also a serious constraint. Many programs depend on values that appear only during execution, such as parsed input, API responses, or data created by earlier computations. When an AI tool cannot see those values, it has to guess. The guess may be wrong even if the instruction sounds clear. Ambiguity adds additional difficulty. Instructions like “sort the rows” or “convert this into a better format” may hide necessary decisions. The decisions include which key to sort by, which order to use, or what is the definition of “better”. A model will still produce code, but the code may reflect assumptions that the developer did not intend.

A common solution is to add more text to the prompt. Developers may paste variable values, describe data formats, or explain requirements in detail. This approach helps, but it also reduces the original benefit. The workflow becomes a loop of prompting, reading, and editing prompts again. It also moves intent away from the code. The prompt exists in a chat window or a

2 Chapter 1 Introduction

temporary edit box. The next time someone reads the program, the intent may be lost. The program may also become harder to maintain because the intent and reasoning are not recorded in a structured way.

To address these challenges, researchers have explored different approaches. Some academic systems try to mix natural language prompts directly into the code. For example, frameworks like AskIt and APPL allow developers to write prompts as part of their programs. These tools make it easier to format AI outputs and manage AI calls. However, they do not focus on code generation. They also do not automatically capture the runtime environment to help the AI understand the context. Other studies focus on improving the accuracy of the code generation. Some methods force the AI to follow strict syntax rules. Other systems generate multiple code options and use tests to pick the best one. While these ideas improve reliability, they do not warn developers when their natural language instructions are too vague.

This thesis presents Promo. Promo is a system that embeds natural language instructions directly into the Racket programming language. In Promo, developers write instructions enclosed in square brackets within ordinary Racket code. When the program runs, the Promo interpreter translates these instructions into executable Racket code using a large language model. By treating natural language as part of the program, the system can make constraints on where code is generated, provide runtime information to the model, and check possible ambiguity of instructions before generation. Instead of asking the developer to move between code and an external assistant, Promo keeps the instruction next to the regular code it affects. The brackets act as a precise edit boundary and also a placeholder for an expression. This matches how developers write programs. They often sketch a structure and fill in details later, or use an AI assistant to generate code at the specific positions.

Promo tries to address the three limitations mentioned above through specific mechanisms. Position control ensures that generated code replaces only the bracketed instruction and does not modify surrounding code. Runtime information captures variable values during execution and includes them in the prompt so the model can see actual data structures of runtime variables. An ambiguity checker implemented as a language server inspects instructions and warns developers when an instruction is too vague.

Promo is implemented as two cooperating components. The first component is an interpreter that executes programs containing embedded natural-language instructions. It runs ordinary Racket code, but it also detects bracketed instructions and decides when to call an LLM. The second component is a language server that integrates with editors through the Language Server Protocol. It provides ambiguity diagnostics while the developer is writing code. During editing, the system can give fast feedback and highlight unclear instructions. During execution, the system should have access to runtime values and should translate instructions into concrete code.

This thesis also evaluates Promo with experiments on a dataset of Promo programs derived from open-source Racket code. The experiments focus on the three mechanisms above. They test whether position control is maintained, whether runtime information improves execution success, and whether ambiguity warnings correlate with lower success rates.

The rest of this thesis is organized as follows. Chapter 2 reviews existing approaches to AI-assisted programming, including editor-based tools, language-integrated frameworks for LLM calls, constrained generation methods, and ambiguity detection techniques. It also describes the motivation for this work by illustrating problems with current tools. Chapter 3 presents the

design of Promo, explaining how position control, runtime information, and ambiguity checking are implemented. Chapter 4 describes experiments that evaluate these features. Chapter 5 summarizes the thesis and discusses directions for future work.

Chapter 2

Coding in natural language

2.1 AI Coding

Current Large Language Models (LLMs) can translate natural language (NL) specifications into code. Such progress has made it possible to describe programming tasks in natural language and ask an AI system to generate or revise code. In practice, this AI component is often embedded in toolchains such as editors, build systems, test runners, and deployment pipelines. So questions like where edits happen, what context is used, and whether AI is aligned with users become naturally important.

This section reviews several representative directions. One group is intelligent code editors that integrate LLMs through chat or inline chat (Inline Edit) workflows. Another group of work provides language-integrated interfaces. Such frameworks treat LLM calls as part of the code rather than as a separate assistant. A third direction focuses on reliability. Existing work has explored constraining decoding so that model outputs satisfy syntax rules, or using execution to verify and guide generation. Finally, some methods focus on addressing ambiguity in natural language specifications, since ambiguity is often a critical cause of inconsistent or incorrect code generation.

2.1.1 AI-assisted Editors and in-IDE Code Generation

Commercial tools increasingly integrate LLMs into editors as chat assistants and inline chat (Inline Edit) features. Such tools improve the efficiency of using AI to write code by eliminating manual interface switching, providing context to the LLM, and adapting generated code to existing programs. For example, Cursor provides an inline chat (Inline Edit) workflow in which the user selects a region, invokes an edit command, and then writes a natural-language instruction to modify that region in place [1,2]. Tools with similar interaction patterns appear in other IDE assistants. Such tools are treated as fast assistants for code refactoring, code generation, and local fixes. When using these features, the developer is responsible for reviewing the generated results.

This interaction pattern is attractive. In actual development, many edits are local refactors, small feature additions, code cleanups, or function-level changes that can be described briefly. The user can repeatedly adjust the instruction until the output looks right and then apply additional

edits elsewhere if needed. At the same time, the pattern makes an assumption that the intended change can be captured by a natural-language request plus the context that the tool decides to include. In practice, the user still bears responsibility for correctness because the tool is not guaranteed to respect the boundaries perfectly or to infer missing information or constraints. The quality of the workflow depends on model capability, how the tool chooses surrounding context, and how it helps the user detect unintended changes.

There are studies on how such tools affect productivity. It appears to depend on factors such as developer experience, task type, and research setting. A large-scale study on GitHub Copilot reports productivity improvements in programming tasks [3]. In contrast, another study on experienced open-source developers reports that the use of AI coding tools can increase the total time required for assigned tasks [4]. These results are not necessarily contradictory. In addition to the factors we mentioned, how well the tool is integrated, the granularity of the edits, and the evaluation workflow all affect the outcome. An important point we can learn is that while one purpose of such AI tools is to accelerate programming, current systems still have room to improve. It is important for the AI to clearly understand developers' actual intent and reduce rework, such as manual debugging and prompt refinement. Occasional problems include edits that stray outside the intended region, or cases where the model lacks critical context information.

Recent human studies also reveal common problems in daily use. Vaithilingam et al. report that many users treat Copilot as a useful starting point, but they still spend effort understanding and debugging the generated code [5]. Beyond functional correctness, security is another concern. Pearce et al. show that Copilot can suggest insecure code in scenarios relevant to security, so developers still need to conduct careful review and testing [6].

2.1.2 Embedding Natural Language Instructions in Code

In contrast with editor tools, another line of work aims to integrate LLM calls more directly into the *programming model*. Rather than treating the LLM as an assistant, these systems provide language-integrated constructs so that natural-language instructions can be written together with ordinary code, with explicit control over inputs, outputs, and error handling.

AskIt

AskIt, proposed by Okuda and Amarasinghe, introduces a domain-specific language to simplify the integration of an LLM into code [7]. In AskIt, developers can define functions using prompt templates and specify the expected return type of LLM-generated output. This mechanism mitigates the mismatch between unstructured text generation and the structured values expected by a programming language. The system will guide the LLM to produce output in a given type signature such as a boolean, an integer, or a specific JSON schema. As shown in Figure 2.1, AskIt lets developers define functions in a prompt template, and it generates system prompts and validation logic so that the model output matches the declared return type. AskIt attempts to parse the output into the specified type and will re-invoke the model with adjusted prompts if parsing fails. This mechanism eliminates extra manual work. Developers save time by avoiding manual parsing of LLM outputs or handling answers in the wrong format. It was reported to improve reliability and reduce prompt length in the experiments. By combining prompting func-

6 Chapter 2 Coding in natural language

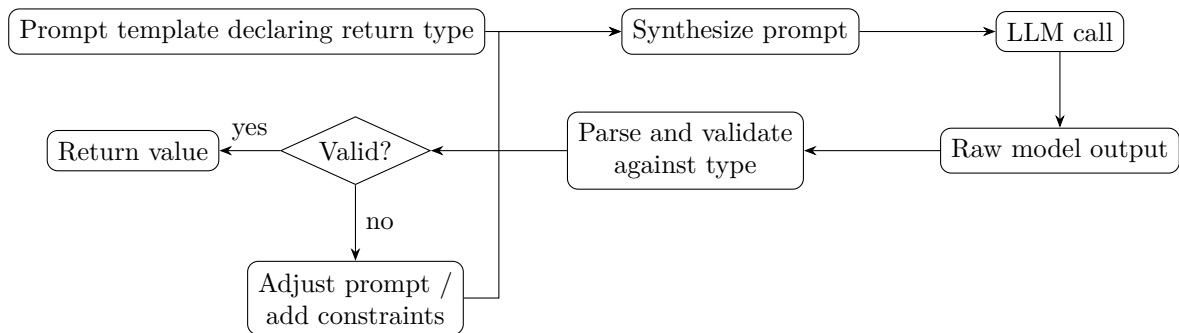


Figure 2.1. AskIt loop: type-guided prompting plus parse/validate-and-retry until the output matches the declared type.

tions with type checking, AskIt mitigates the burden of parsing and error handling by integrating the processing into the framework.

APPL

APPL (A Prompt Programming Language) similarly embeds prompts into Python, but emphasizes execution structure and ease of debugging [8]. Dong et al. propose a syntax for mixing program logic and prompt instructions, with an asynchronous runtime that can execute independent LLM calls concurrently. Since inference latency is often a main cost, this concurrency model can improve the efficiency of prompt-heavy pipelines. Another challenge is reproducibility and diagnosis when a run fails or produces an unexpected result. To address this, APPL includes a tracing module that records inputs, prompt variations, and raw outputs. So developers can replay and inspect executions. For systems that integrate LLM output into code, typed interfaces, concurrency, and traceability can be helpful.

LMQL and DSPy

Several other systems also try to make prompting more like programming. LMQL mixes prompt text with control flow and output constraints. It compiles the query into an efficient inference procedure that can reduce the number of model calls [9]. DSPy represents LLM calls as declarative modules with input and output signatures, and compiles them to optimize an end metric, which can reduce manual prompt tuning in larger pipelines [10].

2.1.3 Constrained Decoding and Execution-guided Synthesis

Another kind of work focuses more on making the generation reliable under constraints. One idea is to constrain the decoding process so that the generated sequences respect syntax rules. Another direction is to use execution to filter or guide generation to improve correctness.

Synchromesh and Constrained Semantic Decoding

Synchromesh introduces *Constrained Semantic Decoding* that restricts generation at each decoding step [11]. So the next tokens are valid under the target language or API. This replaces unconstrained next-token sampling with a procedure that is controlled by a language-specific completion engine to mask invalid tokens. This is especially useful when invalid outputs are expensive or when the generation must always satisfy hard constraints like API schemas. It provides a method that prevents invalid outputs rather than detecting them after generation.

Learning to Verify

Although constrained decoding can ensure that the code is legal in syntax, it does not guarantee correctness. LEVER (Learning to Verify) addresses the challenge of semantic correctness by using execution results for verification and reranking [12]. LEVER generates multiple candidates for a natural-language query, executes them, and uses a learned verifier model to score correctness based on the query, the program, and the execution outcome. This is an explicit generate-execute-verify loop that trains a verifier to recognize meaningful properties of execution results. Listing 2.1 shows this loop in pseudocode.

Listing 2.1. LEVER generate-execute-verify reranking

```

1 def lever(nl_query, code_llm, verifier, K):
2     # 1) sample multiple candidate programs
3     programs = [code_llm.sample(nl_query) for _ in range(K)]
4     logp     = [code_llm.logprob(p, nl_query) for p in programs]
5
6     # 2) execute each candidate (in a sandbox in practice)
7     results  = [safe_execute(p) for p in programs]
8
9     # 3) learned verification using (query, program, execution
10         outcome)
11     vscore   = [verifier.score(nl_query, p, r)
12                 for (p, r) in zip(programs, results)]
13
14     # 4) rerank by combining base model score and verifier score
15     combined = [lp + vs for (lp, vs) in zip(logp, vscore)]
16
17     # (optional) marginalize/merge candidates with identical
18         execution results
19     # combined = marginalize_by_execution_result(programs, results
20         , combined)
21
22     return programs[argmax(combined)]

```

Execution-guided Program Synthesis

The idea of conditioning generation on execution feedback also appears in earlier work on execution-guided program synthesis [13]. Instead of generating an entire program in a single pass, the system uses intermediate execution information to redirect the search by reducing the space of candidates. One important idea is that execution provides information that static text alone cannot provide in some cases. This research uses runtime information to inform code generation.

Execution feedback is also used as a selection signal at a larger scale. AlphaCode generates many candidate programs and filters them using tests and clustering before selecting final submissions [14]. CodeT follows a similar generate-and-test workflow, but it also generates tests to help rank candidates when tests are missing [15]. For code completion, CCTEST tests a completion model using mutated contexts to expose inconsistent outputs [16]. It selects a completion that is more consistent across those outputs.

2.1.4 Ambiguity Detection

Even with better interfaces and reliability mechanisms, natural language could be the source of ambiguity. Ambiguity shows up in many forms, such as polysemy, scope uncertainty, or unspecified thresholds such as “small” or “fast”. In programming targets, such ambiguity can be costly because the output needs to be precise ultimately.

Ambiguity-aware Interfaces

Ding et al. propose AmbiSQL for detecting ambiguity in natural-language queries for text-to-SQL and for supporting interactive resolution [17]. The approach is relatively creative in treating ambiguity as a detectable factor rather than as an implicit model failure.

Automated Ambiguity Repair

SpecFix (Jia et al.) explores a different direction by repairing ambiguous natural-language requirements automatically [18]. The repair is implemented by analyzing variance across generated programs. The intuition is that if a prompt yields a wide distribution of distinct outputs, the prompt itself may be underspecified. SpecFix then refines the specification to reduce that variance, aiming to achieve a more stable interpretation without requiring immediate user intervention. As shown in Figure 2.2, SpecFix first analyzes and repairs the LLM’s interpretation captured by the distribution of induced programs using traditional testing and program repair, then refines the problem description based on how that distribution changes.

Other Proposals

Li et al. propose a code generation setup where the model asks clarification questions first and uses the answers before generating Python code [19]. Nandan and Kumar explore a prototype that suggests inputs when a task specification may be ambiguous, requests limited human feedback, and uses it to resolve ambiguity during code generation [20]. In a different setting, Yen et al.

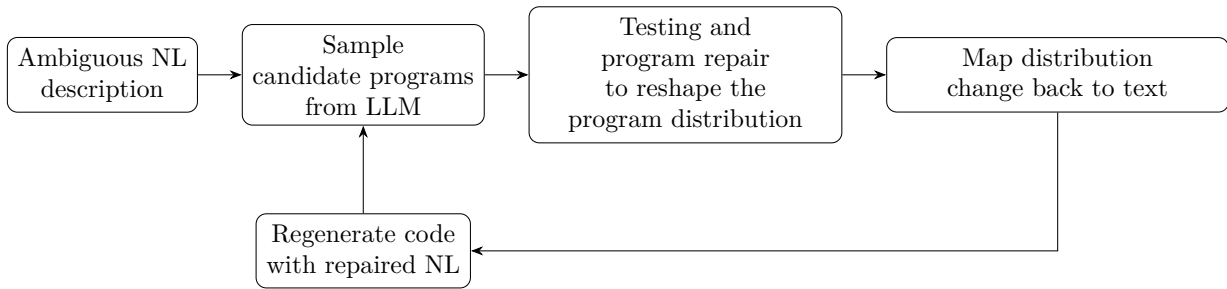


Figure 2.2. SpecFix reduces ambiguity by repairing the induced program distribution and then rewriting the NL description accordingly.

analyze natural-language protocol specifications, detect ambiguous or under-specified sentences, and generate code after the text is clarified by the author [21].

2.2 Motivation

While existing AI-assisted programming tools have made significant progress, several practical challenges remain unsolved. These challenges motivate the design of Promo and the features described in Chapter 3.

2.2.1 Position Control of Code Generation

The position of code generation in current tools is often difficult to control. Developers may want to generate a small expression, a single statement, or an entire function block. However, most assistants do not offer strict control over the position and scope of generated code, which can disrupt existing code structure. There is a lack of fine-grained controllability that allows developers to specify exactly where and how much code should be generated. This often leads to manual adjustments or refactoring after generation.

A simple experiment with JetBrains AI Assistant (version 251.23774) illustrates this problem. As shown in Figure 2.3, we have a function that returns a tuple of three elements. The intention is to modify only the third element of this tuple by selecting it and prompting the AI to generate a “sorted list”. However, the result shown in Figure 2.4 demonstrates a mistake. Instead of modifying only the selected element, the assistant regenerates the entire return statement, altering all three elements of the tuple. This exemplifies a common problem: the AI’s edit position does not match the user’s selection, and generation often deviates from the user’s initial intention.

2.2.2 No Runtime Context

Current tools rely only on static context from source code and the prompt. They have no access to the runtime environment of the program, such as dynamic values of variables or current states of data structures. This limits their ability to generate context-aware code. For instance, a

```
def read_csv_rows(path):
    rows = []
    with open(path, newline='', encoding='utf-8') as f:
        for line in f:
            rows.append(line.rstrip('\n').split(','))
```

sorted rows

```
return rows[1: len(rows) // 2], rows[len(rows) // 2 + 1:],
```

Figure 2.3. A trial with JetBrains AI Assistant: selecting only the third element

```
1 def read_csv_rows(path):
2     rows = []
3     with open(path, newline='', encoding='utf-8') as f:
4         for line in f:
5             rows.append(line.rstrip('\n').split(','))
6     return sorted(rows[1: len(rows) // 2]), sorted(rows[len(rows) // 2 + 1:], rows[0])
```

Figure 2.4. Result from JetBrains AI Assistant: the entire return statement was modified

developer’s instruction might depend on the content of a list that is only determined at runtime, which is invisible to today’s assistants.

In the example in Figures 2.3 and 2.4, the developer’s intention is to sort a list. However, the correct sorting logic depends on the actual content of the data at runtime. For example, if a column in the file contains numeric strings such as "10" or "2", it should be sorted numerically rather than lexicographically. AI assistants working with static context have no way to know this.

We discuss a similar failure case with Cursor’s inline chat (Inline Edit) feature. As shown in Figure 2.5, we use Cursor to generate the body of a Racket function. This Racket program requires another Racket file `closest-data.rkt`, which is shown in Figure 2.6. The file `closest-data.rkt` defines a `point` struct with `x` and `y` fields and creates a variable `points` containing a boxed, shuffled list of 1000 point structures, where each point has coordinates (i, i^2) . When Cursor attempts to generate code for the instruction “find the closest pair of points,” it cannot see the actual runtime state of the `points` variable. As shown in Figure 2.7, Cursor generates code that attempts to solve the problem, but it has no way of knowing that `points` is wrapped in a box or of knowing the internal structure of the point data. The generated code tries to reference functions by identifiers like `sequence->vector`, which does not actually work in this context. Without access to runtime information showing the internal structure of `points`, Cursor may

```

1  #lang racket
2  (require "closest-data.rkt")
3
4  (define (closest-pair points)
5
6
7  ;; vector -> vector -> list
8  (define (closest-pair/sorted Px Py)
9    (define L (vector-length Px))
10   (cond ((= L 2) (vector->list Px))
11         ((= L 3) (apply min-pair (combinations (vector->list Px) 2)))
12         (else (let*-values (((Qx Rx) (vector-split-at Px (floor (/ L 2)))))
13                 ; Rx-min is the left most point in Rx
14                 ((Rx-min) (vector-ref Rx 0))
15                 ; instead of sorting Qx, Rx by y
16                 ; - Qy are members of Py to left of Rx-min
17                 ; - Ry are the remaining members of Py
18                 ((Qy Ry) (vector-partition Py (curryr left? Rx-min)))
19                 ((pair1) (closest-pair/sorted Qx Qy))
20                 ((pair2) (closest-pair/sorted Rx Ry))
21                 ((delta) (min (distance^2 pair1) (distance^2 pair2)))
22                 ((pair3) (closest-split-pair Px Py delta)))
23                 ; pair3 is null when there are no split pairs closer than delta
24                 (min-pair pair1 pair2 pair3))))))

```

Figure 2.5. A trial with Cursor

guess about the data structure and available operations. Similar challenges may occur in cases such as API requests or raw data input. In such cases, static context alone may be insufficient to decide the correct implementation.

2.2.3 Misalignment with Developer Intent

The generated code often suffers from misalignment with the developer’s true intention. This happens because translating a high-level, sometimes ambiguous, natural-language instruction into precise code is inherently lossy. LLMs might misinterpret a nuance, or users may provide vague specifications. This forces the developer into a tedious cycle of prompt refinement, code review, and manual correction, which defeats the purpose of using an AI assistant for efficiency.

The same example in Figure 2.3 also highlights this problem. The user’s prompt, “sorted rows”, is ambiguous. It does not specify the sorting key or the order. The LLM is forced to

12 Chapter 2 Coding in natural language

```
1 #lang racket
2 (struct point (x y))
3
4 (define points
5   (box (shuffle
6         (for/list (( i (in-range 1000))) (point i (* i i))))))
7
8 (provide (struct-out point) points)
```

Figure 2.6. closest-data.rkt

make an implicit assumption to sort by the first column, alphabetically, or numerically. In longer generations, the model may make more unseen and unintended decisions.

These challenges of uncontrolled position, no runtime context, and intent misalignment motivate the features we develop in Chapter 3. Position control addresses the first challenge; runtime information addresses the second; and the ambiguity checker addresses the third.

```

4  (define (closest-pair points)
    find the Closest Pair of Points
    Reject Accept ⌘↵ ×
    Add a follow-up
    Auto ▾ Edit Selection ▾ ⬆
)
5  (check-type points)
6  (define ps (sequence->vector points))
7  (define L (vector-length ps))
8  (cond ((< L 2) (error 'closest-pair "Need at least 2 points"))
9        ((= L 2) (vector->list ps))
10       (else (let ((Px (vector-sort ps point<?))
11                  (Py (vector-sort ps point<? #:key point-y)))
12               (closest-pair/sorted Px Py))))
13
Undo ⌘N Keep

```

Figure 2.7. Result from Cursor's inline chat (Inline Edit)

Chapter 3

Embedding Natural Language into Racket

Given the challenges outlined in Section 2.2, developers still lack fine-grained, context-aware control over AI-driven code generation. This chapter presents our approach by treating natural-language instructions as part of the code within the Racket programming language. Any expression enclosed in square brackets, for example, `(+ 1 2 [sum remaining numbers to 10])`, is translated by an LLM into concrete Racket code at runtime, with access to the runtime environment.

Promo consists of two components: an interpreter that executes programs containing natural-language instructions, and a language server based on the Language Server Protocol (LSP) to support NL inspection within editors. The LSP is a protocol for communication between editors and language-specific servers; an editor extension following this protocol can provide features such as code inspection, auto-completion, quick fixes, and code actions.

Promo's surface syntax is intentionally minimal. Any expression enclosed in a pair of square brackets `[and]` is treated as a natural-language instruction. This allows developers to embed instructions at any level of granularity: from a single expression to an entire function body, simply by wrapping the desired region with brackets. The text within the brackets can be any natural-language string.

Figure 3.1 presents an example of Promo, demonstrating how a developer can find the item most semantically similar to a specific query within a dataset. The code consists of three parts: a helper function to load data, a statement that loads a vector dataset into the `data` variable (with its content shown in Figure 3.2), and a final statement that embeds an NL instruction within a Racket `print` form. This entire bracketed substring, `[the name of the item most semantically similar to dog in data]`, is a Promo expression. At runtime, the Promo interpreter recognizes this expression and sends its text content along with other information, including the variable `data` from Figure 3.2, to an LLM. The LLM returns executable Racket code, which the interpreter splices back into the program for immediate evaluation.

The following sections describe the overview of the Promo execution workflow and the three main features that we implement to address the challenges of Section 2.2: position control, runtime information, and ambiguity checking.

```

1 (define (read-data-file filename)
2   (with-input-from-file filename
3     (lambda () (read))))
4
5 (define data (read-data-file "../data.rkt"))
6
7 (print
8   [the name of the item most
9     semantically similar to dog in data])

```

Figure 3.1. A code example of Promo

```

1 (("cat" #(0.1 0.8 0.3 0.2 0.5))
2  ("car" #(0.9 0.1 0.2 0.8 0.4))
3  ("banana" #(0.3 0.2 0.9 0.1 0.6))
4  ("wolf" #(0.2 0.9 0.4 0.3 0.5))
5  ("apple" #(0.4 0.3 0.8 0.2 0.7))
6  ("bicycle" #(0.8 0.2 0.3 0.7 0.5))
7  ("grape" #(0.3 0.4 0.7 0.1 0.8))
8  ("lion" #(0.2 0.85 0.35 0.25 0.55))
9  ("truck" #(0.85 0.15 0.25 0.75 0.45))
10 ("orange" #(0.35 0.25 0.85 0.15 0.65)))

```

Figure 3.2. First 10 lines of data.rkt used in Figure 3.1

3.1 Execution of Promo Programs

Promo runs a program by combining two components. The Promo interpreter manages natural-language instructions and decides when to call an LLM. The ordinary Racket runtime evaluates the resulting Racket code. This allows Promo to reuse Racket’s standard evaluation rules.

In Promo, square brackets are reserved for natural-language instructions. Normal Racket code should be in parentheses. This rule makes it easy to locate instructions in the source text and avoids confusion with regular Racket syntax.

Promo reads the source program and splits it into top-level expressions. A top-level expression is a form that Racket would normally evaluate in sequence. Promo processes these expressions one by one. It preprocesses the next top-level expression and then evaluates the result. This keeps the original order of evaluation, so earlier definitions and side effects affect later code in the usual way.

When a top-level expression contains at least one bracketed instruction, Promo asks an LLM to translate the expression into Racket code. The prompt includes the original expression and the full program text. The model is asked to produce a single top-level Racket expression that can replace the original one.

The preprocessing step does not require the LLM to resolve every instruction at once. The model is allowed to keep some bracketed instructions unchanged. It is also allowed to introduce new bracketed expressions. This behavior is useful when the model can outline a solution

```
1 ;; LLM output after preprocessing (still contains NL)
2 (define (factorial n)
3   [compute factorial of n])
4
5 ;; Code executed by Racket after rewriting
6 (define (factorial n)
7   (generate-and-run-code "compute factorial of n"))
```

Figure 3.3. Promo interpreter rewrites LLM outputs containing NL expressions

but wants to defer concrete or detailed implementation. It is possible that the implementation depends on information that is difficult to infer before the execution of the top-level expression.

After receiving the LLM output, Promo scans it for any remaining bracketed instructions. Each remaining instruction is rewritten into a call to a runtime helper. We call this helper `generate-and-run-code`. The bracket content becomes a string argument. The rewritten code is now valid Racket code because it does not contain free-form natural language inside brackets. The example in Figure 3.3 shows the idea. The model may output a function definition that still contains a natural-language expression. Promo rewrites the expression into a runtime call before evaluation.

The `generate-and-run-code` helper is evaluated by the Racket runtime during execution. When it runs, it builds a new prompt and queries an LLM again. This prompt includes the instruction string and the current runtime variable bindings. The information about local variables is also included in the prompt here, which is not provided in the LLM queries at the preprocessing step. The model returns a Racket expression as text. Promo reads this text as an S-expression and evaluates it with `eval` in the current namespace. The evaluation result becomes the value of the original bracketed instruction.

This mechanism does not support nested instructions. The LLM is asked to generate pure Racket code at this step. In Racket, square brackets are originally treated in the same way as parentheses. If the model output still contains bracketed code, the Promo interpreter considers it a regular Racket S-expression rather than a natural language instruction.

To provide runtime variable bindings, Promo tracks bindings as the program runs. Promo replaces key binding forms with tracked macros, such as `define`, `let`, and `let-values`. These tracked forms behave like the original forms, but they also record bindings into a global structure. In our implementation, this structure is called `current-variable-stack`. It is a stack of hash tables, where each table represents one lexical scope. Promo pushes a new table when entering a scope, records new bindings into the top table, and pops the table when leaving the scope.

For instance, function parameters are recorded at call time. When a tracked function is invoked, Promo records the actual argument values under the parameter names in a fresh scope. This makes the runtime information useful for generation, because the model can see concrete values that are only available during execution.

This execution workflow is the foundation for the mechanisms described in the following sections. Position control relies on the top-level preprocessing. Runtime-aware generation depends

on the tracked bindings collected during execution.

3.2 Position Control

By position control, we mean that the LLM can only modify the code inside brackets. As described in Section 3.1, the code generation may occur in either the preprocessing of each top-level expression or the call to the `generate-and-run-code` helper. The position control is achieved automatically because of the implementation of `generate-and-run-code`. When preprocessing, we replace the Promo code with LLM output at the level of top-level expressions. Each original top-level expression may contain multiple bracketed expressions. We implement two ways to replace the top-level expression that contains natural language instructions. The first is to replace the whole top-level expression with the output of the generation. The second is to replace only bracketed expressions with generation output. The position control is independent of the correctness of the generated code.

3.2.1 Whole Expression Replacement

While the second way guarantees that only bracketed parts are replaced, the first way may modify other parts of the top-level expression. Even if the LLM is required to modify only the bracketed parts, due to the randomness and limited capability of the LLM to follow the instructions, allowing AI to modify other parts of the top-level expression may lead to a violation of position control. So when querying to generate the whole top-level expression, the interpreter checks whether the code outside brackets is changed. The interpreter checks this by comparing the original top-level expression with the generated top-level expression. Before comparing the non-bracketed parts, Promo first checks that the LLM output contains exactly one top-level Racket expression. If there is an additional top-level expression remaining after the first expression, the candidate is rejected, and we retry generation. This prevents the model from generating multiple top-level expressions. It then finds and saves all non-bracketed parts of the original expression in order. Then it checks whether these parts appear in the generated expression. In the generated expression, these parts need to appear in non-overlapping positions in the same order. If not, the code is considered to be out of position control.

Figure 3.4 illustrates the workflow when the entire top-level expression is generated. In this method, the interpreter sends the whole top-level expression containing the natural language instruction to the model. The LLM generates a full replacement expression. To ensure that the model has not modified the code outside the brackets, the interpreter performs a validation. It validates that only one top-level expression is generated, then extracts the non-bracketed elements from the original source and verifies their presence and order in the generated output. If this validation passes, the code is accepted; otherwise, it is flagged as a position control violation.

3.2.2 Bracket-only Replacement

For the second method, the interpreter calls the LLM to generate only the bracketed parts in the specific top-level expression. Once the bracketed parts are generated, the interpreter replaces the

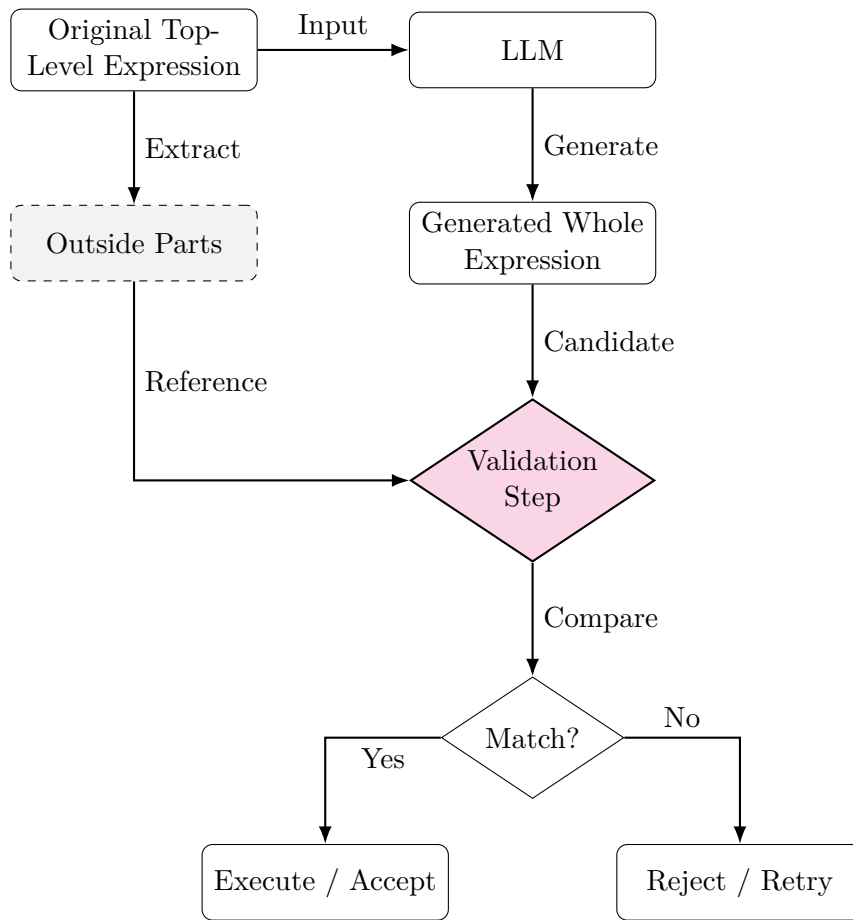


Figure 3.4. Method 1: whole top-level expression replacement with verification

bracketed parts with the generated code. Then the top-level expression is changed into a Racket expression, which is evaluated by the Racket runtime embedded in the interpreter.

Figure 3.5 shows the second approach, which enforces position control by constructing the expression including NL instructions. In this pipeline, the interpreter extracts the top-level expression. The model returns replacement fragments specifically for the bracketed instructions. The interpreter then fills these fragments back into the parsed template. This process guarantees that the surrounding code remains strictly unchanged, as it will not be exposed to the generation process for modification.

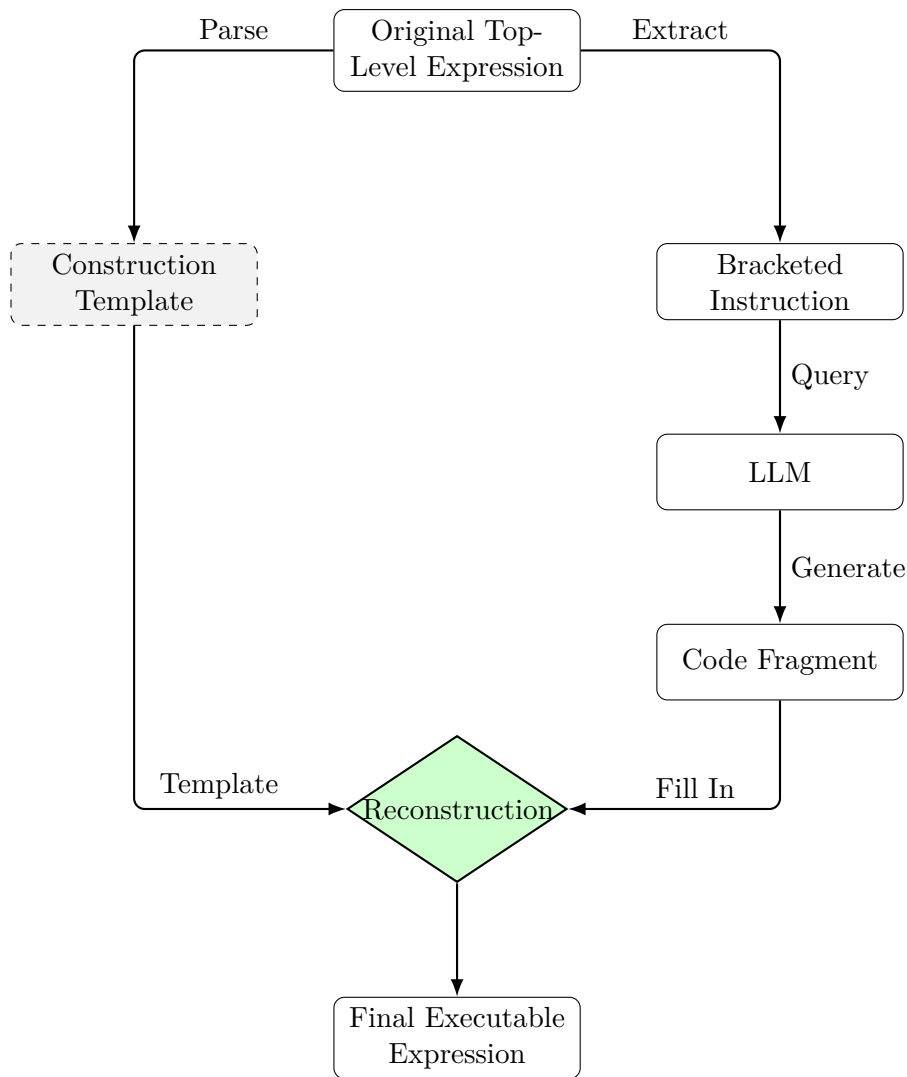


Figure 3.5. Method 2: Bracket-Only Replacement with Structural Preservation

```
1 BindingStack ← [] // Stack of dictionaries
2
3 function push_scope():
4
5     BindingStack.push(new Dictionary())
6
7
8 function pop_scope():
9
10    if BindingStack is not empty:
11        BindingStack.pop()
12
13
14 function track_var(name, value):
15
16    if BindingStack is not empty:
17        BindingStack.top()[name] ← value
```

Figure 3.6. Pseudocode for binding stack operations

3.3 Runtime Information

One limitation of inline code generation tools, such as the quick edit feature of Cursor, or Modify Selected Code in JetBrains AI Assistant, is that they only provide static context to the LLM. This means that the LLM cannot access the current state of the program, such as the types, structures, or values of variables. We propose a way to add runtime context to the prompt for code generation. Our pipeline includes a tracker to track variables, formatting each variable to an easy-to-read format, and selection to reduce irrelevant variables.

3.3.1 Variable Tracking

To provide runtime context to the LLM, the interpreter needs to track variables during program execution. We implement a variable tracking mechanism to track variables. The variable tracking mechanism overrides original binding forms of Racket and records variable bindings when they are created.

As shown in Figure 3.6, the tracker maintains a stack of binding environments. Each element in this stack represents a scope, ensuring variable shadowing and scope boundaries are correctly implemented.

The tracker also overrides standard binding forms. It provides replacement macros with the same interfaces. The tracked binding forms include let forms such as `let` and its variants, function forms like `lambda` or `define`, loop forms such as `for`, pattern matching like `match`, and other forms such as `parameterize`. Each tracked form follows the same pattern: push a new scope, register all bound variables, execute the body, then pop the scope. For example, the tracked

```
1 macro tracked_let(bindings, body):
2
3     push_scope()
4
5     for (var, val) in bindings:
6
7         actual_value ← evaluate(val)
8         bind(var, actual_value)
9         track_var(var, actual_value)
10
11    result ← evaluate(body)
12    pop_scope()
13
14    return result
```

Figure 3.7. Pseudocode for tracked let macro

version of `let` is implemented as shown in Figure 3.7.

For function definitions, the tracker records the function name when the definition is evaluated. And the parameter bindings are recorded each time the function is invoked. So the actual argument values are captured at the function calls. The overridden function definition is shown as pseudocode in Figure 3.8.

The tracker provides functions to retrieve tracked variables. As shown in Figure 3.9, `get-all-tracked-vars` returns a hash table containing all bindings, with inner scope bindings shadowing outer scope bindings when names conflict.

During execution, all variables in the scope are available to be provided in the prompt. This allows the LLM to access the current runtime values of variables when generating code. We will introduce how they are provided in prompts next.

3.3.2 Variable Formatting

Raw runtime values are sometimes not suitable to be included in the prompt directly. For example, a variable may have a deeply nested structure, be a very long string, or a binary blob. Providing them directly will make it hard for the model to understand and use this information effectively. Also, including variables like long strings in the prompt may deteriorate the LLM’s performance, cost more tokens, or exceed the token limits. We implement a formatting mechanism to convert the tracked values into human-readable formats. We expect our formatting to preserve enough information for the LLM to understand the variables well, at the same time, keep the output short to avoid waste of tokens or exceeding token limitations.

The formatter recognizes the data types of Racket variables and applies specific formatting. Primitive values like booleans, symbols, and numbers are printed directly. Strings are truncated when they exceed a length threshold. For compound structures like lists and hash tables, instead of enumerating every element, the formatter samples the first several elements and provides them

```
1 macro tracked_define_function(name, params, body):
2
3     function impl(args):
4
5         push_scope()
6
7         for (param, arg) in zip(params, args):
8             track_var(param, arg)
9
10        result ← evaluate(body)
11        pop_scope()
12
13        return result
14
15
16    bind(name, impl)
17    track_var(name, impl)
```

Figure 3.8. Pseudocode for tracked function definition

```
1 function get_all_tracked_vars():
2     result ← new Dictionary()
3     for scope in BindingStack (bottom to top):
4         for (name, value) in scope:
5             result[name] ← value // Later scopes override
6     return result
```

Figure 3.9. Pseudocode for retrieving tracked variables

```
1  === Runtime Variables ===
2
3  seq: string (length 195): "
      GGATGATATTGTACACTGTACAGAGTTGGGAGGGTTTTTTGCGAGAATCTTT..."
4
5  k0s: list of 4 elements: [integer: 0, integer: 0, empty list (null
      ), empty list (null)]
6
7  kons: procedure (takes 5 arguments)
8
9  finalise: procedure (takes 4 arguments)
10
11 s: string (length 195): "
      GGATGATATTGTACACTGTACAGAGTTGGGAGGGTTTTTTGCGAGAATCTTT..."
12
13 as: integer: 53
14
15 ts: integer: 53
16
17 gs: integer: 48
18
19 (Total: 8 variables)
```

Figure 3.10. Example output from the variable formatter

with the total count. An example output from the formatter is shown in Figure 3.10. The long string is sampled instead of fully shown.

3.3.3 Variable Selection

Irrelevant prompts are a reason for hallucination in LLMs. The quality of LLM’s output relies on providing informative and relevant context. Distracting or long input can degrade the performance of the LLMs. Providing information about irrelevant variables to the LLM may be confusing or misleading. Also, information about variables, especially variables with large or complex structures, can take a considerable portion of the input token count. Including irrelevant variables can affect both the accuracy and the cost. We propose a selection strategy to reduce irrelevant variables provided.

The natural language prompt could be vague and refer to variables without explicitly using their names. The necessary information cannot always be parsed from the natural language instruction. But it is possible to infer the necessary variables from the context. So we need to extract and compare the semantic information about the runtime variables and the natural language instruction. This requires us to understand the semantics of the runtime information and the natural language instruction. We provide two different methods to achieve this and select

the necessary information for code generation.

The first selection method is based on an LLM. This selection strategy treats the filtering of variables as a ranking problem. Previous research has explored using LLMs as re-ranking agents with or without additional training [22]. Their results suggest zero-shot LLMs can perform as well as or even outperform specialized models. By providing a set of all tracked variables and the natural language instruction in the prompt, we ask the LLM to order the variables according to their relevance to the instruction. Then the interpreter takes the top-k most relevant variables that will be included in the prompt for code generation.

As we introduced in Section 3.3.2, the interpreter gets all tracked variables and formats each variable as a string including its name and description. For example, a list variable ‘records’ will be formatted as “records: list of 150 elements, first 3: [...]”. Such formatted strings are ranked then.

The interpreter makes a reranking query. This query provides all tracked variables with indexes attached. The LLM then orders them by their relevance to the corresponding natural language instruction. The LLM returns a list of indexes sorted from the most relevant to the least relevant.

3.4 Ambiguity Checker

Natural-language instructions are convenient because they can be informal. The downside is that informality often hides decisions that should be explicit, especially for instructions that will be turned into code.

Promo mitigates this by an ambiguity checker implemented as a language server. To integrate the ambiguity detection into the coding environment, we utilize the Language Server Protocol (LSP) [23]. The LSP is a standardized protocol that supports the implementation of language features, such as auto-completion, go-to-definition, or error diagnostics. By implementing the ambiguity checker as a language server, the system can run as a background process that analyzes the code and communicates directly with any editor compliant with the LSP, such as VS Code or Emacs.

The Promo language server inspects each bracketed instruction when connected to an editor. It produces feedback directly in the editor, pointing out possible vagueness and suggesting what to clarify. The checker operates on a single bracketed instruction together with its program context.

Each inspected instruction is assigned one of three levels. “Pass” means the instruction is clear enough that a reasonable implementation can be implied by the surrounding code. “Warning” means the instruction is interpretable, but there are multiple understandings or missing details. “Error” means the instruction is meaningless or unrelated to the context, or too vague to implement.

Figure 3.11 shows a concrete example of the ambiguity checker. The program defines a polynomial structure and includes a function `string->poly` with the bracketed instruction `[convert into poly]`. At the top of the file, the program uses `(require racket/sequence "poly-display.rkt" "term-ops.rkt")` to import helper functions such as `poly-print` from external Racket files. Because these functions are defined in separate files rather than in the current

```

Emacs-arm64-11
> mylisp > racket > c_src > new_source > polynomial > simple-polynomial > private > poly-struct.promo

(struct polynomial (terms)
 #:guard p-guard
 #:property prop:procedure
 (lambda (p at)
 (when (and (not (polynomial? at))
            (not (coefficient? at)))
 (raise-argument-error 'polynomial "(or/c coefficient? polynomial?)" 1 p at))
 (let ((res (foldl (lambda (term accum) (poly+ term (poly* at accum))) 0 (polynomial-terms p))))
 (if (= 1 (length (polynomial-terms res))) ;just a number
 (car (polynomial-terms res))
 res)))
 #:methods gen:custom-write
 ((define write-proc p-print)))

(define (poly . ts)
 (polynomial ts))

(define poly? polynomial?)

(define (poly->string p)
 (parameterize ((display-short? #f))
 (poly-print (polynomial-terms p) #f)))

(define (string->poly s)
 ! [convert into poly])
Warning: "convert into poly" - The instruction is too high-level and does not specify that the input string 's' needs to be parsed to extract the polynomial coefficients. While the function name 'string->poly' implies this action,
(parameterize ((display-short? #t))
 (let ((p1 (poly 1 2 3 2 9 10 11))
 (p2 (poly 1 1)))
 (poly= p1 (string->poly (poly->string p1)))))

U:*~ poly-struct.promo 35% L50 Git-example (Promo Flymake[0 1] LSP[promo-lsp:58103]) 1

```

Figure 3.11. An example of editor warning

source, their implementations are not visible to the LLM. Neither the language server performing ambiguity checking nor the interpreter generating code can see the function bodies of `poly-print` or other imported procedures. In this example, the language server flags the instruction `[convert into poly]` with a warning, explaining that the instruction is too high-level and does not specify how the input string should be parsed to extract polynomial coefficients. The warning also notes that while the function name `string->poly` implies the intended action, the exact parsing format remains unclear. This case illustrates how the ambiguity checker operates with limited context.

Figure 3.12 shows the mechanism. The editor sends the current buffer to the language server; the server extracts bracketed instructions, builds a compact inspection prompt for each of them, and queries an LLM. The LLM is asked to respond in a small JSON object with a categorical verdict and an explanation string. The server then maps the verdict into LSP diagnostics and displays them inline.

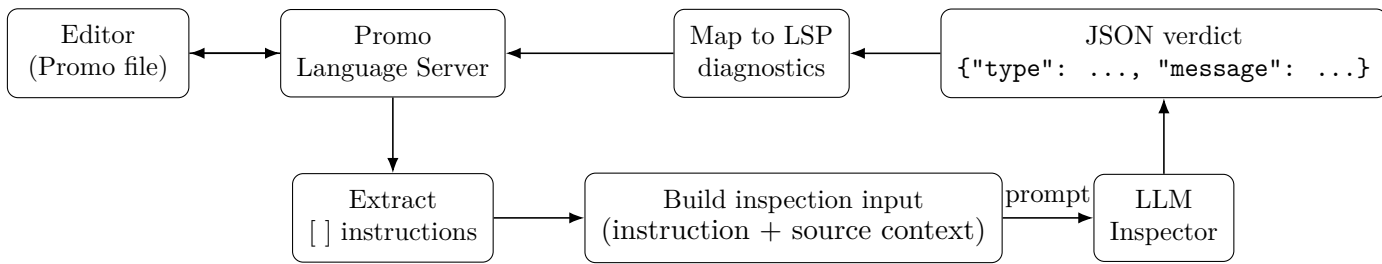


Figure 3.12. Ambiguity checking workflow in the Promo language server. Each bracketed instruction is inspected with its surrounding source context, and the result is surfaced as LSP diagnostics.

3.5 Implementation Details

3.5.1 Interpreter

The Promo interpreter needs to correctly identify natural language instructions in Promo code. Simply extracting the contents in all bracket pairs is not sufficient. Some kinds of bracketed elements should not be treated as natural language instructions. For instance, bracket pairs could be used in comments or strings, but such bracket pairs should not be considered as natural language instructions. To correctly identify bracketed instructions, the interpreter handles these cases when identifying natural language instructions. Figure 3.13 is an example of how the interpreter extracts all natural language instructions from Promo code. The function skips all non-instruction brackets before identifying the actual NL instructions.

The choice of temperature is relatively important in LLM-related research. The temperature parameter can affect the accuracy, and even the length of LLM output. Research also shows that different targets may have different optimal temperatures. While most LLMs support temperature control, there are some LLMs, especially some new models such as the gpt-5 series, that do not support temperature control.

When using these models, we cannot control the randomness of LLM output with the temperature parameter. Based on our observation, the outputs of these models, such as gpt-5, change in different calls with the same prompt. So models like gpt-5 do not behave like 0-temperature models, which output the same result every time. When using claude-sonnet-4.5 for code generation, the temperature parameter is changeable in the Promo interpreter. We set the default temperature to 1.0. One reason is that the Promo interpreter requires randomness in code generation. In Section 3.2, we explained the interpreter may query the LLM to generate code from the same prompt repeatedly to ensure position control. Also, some latest advanced LLMs, such as Gemini 3, have an adjustable temperature parameter; however, they are most optimized for temperature 1.

For the method proposed in Section 3.2.2, the LLM might generate multiple Racket replacements for the bracketed instructions. In such a case, generating a single replacement in one query and making multiple API calls is costly and inefficient. The interpreter queries the LLM

```
1 def extract_racket_brackets(code: str) -> list[str]:
2     """
3     Extracts content within square brackets from Racket code,
4     correctly ignoring comments, strings...
5     """
6     result = []
7     i = 0
8     n = len(code)
9
10    while i < n:
11        # Skip Line Comments ;...
12        if code[i] == ';':
13            i = code.find('\n', i)
14            if i == -1:
15                break
16            i += 1
17            continue
18
19        # Skip Block Comments #|...|#
20        if i < n - 1 and code[i:i + 2] == '#|':
21            end = code.find('|#', i + 2)
22            i = end + 2 if end != -1 else n
23            continue
24
25        # handle other cases
26        # ...
27
28
29        # Find Square Brackets
30        if code[i] == '[':
31            # extract_bracket_pair_logic returns (content,
32                new_index)
33            content, end_index = _extract_bracket_pair_logic(code,
34                i)
35            if content is not None:
36                result.append(content)
37                i = end_index
38            else:
39                i += 1
40                continue
41
42        i += 1
43
44    return result
```

Figure 3.13. Python function for extracting bracketed instructions

```
1 (define *max-string-length* 100)
2 (define *max-list-elements-shown* 5)
3 (define *max-hash-entries-shown* 10)
4 (define *max-total-output-length* 4000)
```

Figure 3.14. Configurable formatting thresholds

to generate replacements in the format of a JSON array of strings. The interpreter then parses the JSON array and replaces the corresponding bracketed instructions with the generated replacements. Another consideration is that a Promo top-level expression may contain 2 or more of the same bracketed instructions, so from the prompt, the LLM cannot distinguish them.

LLMs can generate structured output. Many LLM series, such as gpt or Gemini, support structured output by an additional parameter rather than defining format in the prompt. While the main model we use for code generation, claude-sonnet-4.5, does not support such a feature, the interpreter specifies the format of the output JSON array in the prompt.

The interpreter utilizes the structured output feature when using reranking by an LLM as a variable selection strategy. As introduced in Section 3.3.3, the LLM orders the indexes of variables by their relevance to the bracketed instruction, then the interpreter adds the top-k variables into the prompt for code generation. For the selection task, we use a separate model, gpt-5.1. The reranking query uses gpt’s structured output feature to ensure the LLM outputs a JSON object containing the ordered index list. The default top-k value is set to 8 in order to balance the adequacy of information and the filtering quality.

The selector handles errors including hallucinations from the LLM. If the response is not a valid JSON object or is empty, the selector returns all variables. So all tracked variables will be provided for code generation if errors such as API errors occur.

Here we introduce the implementation of variable formatting. As shown in Figure 3.14, the formatter uses configurable thresholds to control output size. The choice of default thresholds requires a balance between information content and token cost. The 4000-character total limit fits typical prompt budgets with room for enough information.

The core function `describe-value-for-llm`, shown in Figure 3.15, conducts recursive traversal with depth limitations. For byte strings, special handling prevents NUL characters and non-printable data from corrupting the output. Figure 3.16 shows the processing. Code in Figure 3.17 is the Racket function `get-all-tracked-vars-for-llm` that collects all bindings, formats each variable, and concatenates the final output with a limit.

3.5.2 Language Server

Like diagnostic features of many language servers, the Promo language server repeats diagnostic analysis when the text of the programs is changed. When the language server conducts a new analysis on the changed program, it will query the LLM again to categorize the natural language instructions. Even if the text of the natural language instructions remains unchanged, the modification of the Racket code part could lead to changing the categorization of natural language instructions. For this reason, the language server will query the LLM to evaluate the

```

1 (define (describe-value-for-llm v [depth 0] [max-depth 3])
2   (cond
3     [(> depth max-depth) "..."]
4
5     [(null? v) "empty list (null)"]
6     [(boolean? v) (format "boolean: ~a" v)]
7     [(exact-integer? v) (format "integer: ~a" v)]
8     [(real? v) (format "number: ~a" v)]
9
10    [(string? v)
11     (define len (string-length v))
12     (if (> len *max-string-length*)
13         (format "string (length ~a): ~a...\"
14             len (substring v 0 *max-string-length*))
15         (format "string: ~a\" v))]
16
17    [(pair? v)
18     (if (list? v)
19         (let* ([len (length v)]
20                [sample-count (min *max-list-elements-shown* len)]
21                [samples (take v sample-count)]
22                [elem-descriptions
23                 (map (e) (describe-value-for-llm e (+ depth 1)
24                    max-depth))
25                 samples])]
26             (format "list of ~a elements, first ~a: [~a, ...]"
27                 len sample-count (string-join
28                     elem-descriptions ", "))
29             (format "list of ~a elements: [~a]"
30                 len (string-join elem-descriptions ", ")))
31         (format "pair: (~a . ~a)"
32             (describe-value-for-llm (car v) (+ depth 1)
33                 max-depth)
34             (describe-value-for-llm (cdr v) (+ depth 1)
35                 max-depth))))]
33
34 ;; ... additional cases for vector, hash, bytes, etc.
35 ))

```

Figure 3.15. Core variable description function

```

1 [(bytes? v)
2  (format "#\~a\"")
3      (apply string-append
4          (for/list ([b (in-bytes v)])
5              (if (and (>= b 32) (<= b 126)
6                  (not (= b 34)) (not (= b 92)))
7                  (string (integer->char b))
8                  (format "\\x~a"
9                      (let ([hex (number->string b 16)])
10                         (if (< (string-length hex) 2)
11                             (string-append "0" hex)
12                             hex)))))))]

```

Figure 3.16. Handling byte strings in formatting

```

1 (define (get-all-tracked-vars-for-llm)
2  (define vars (get-all-tracked-vars))
3  (define output
4    (call-with-output-string
5      (lambda (out)
6        (displayln "=== Runtime Variables ===" out)
7        (displayln "" out)
8        (define sorted-keys (sort (hash-keys vars) symbol<?))
9        (for ([name sorted-keys])
10           (define val (hash-ref vars name))
11           (define description (describe-value-for-llm val))
12           (fprintf out "\~a: \~a\n" name description))
13        (displayln "" out)
14        (fprintf out "(Total: \~a variables)" (hash-count vars))))))
15
16 (if (> (string-length output) *max-total-output-length*)
17     (string-append
18       (substring output 0 *max-total-output-length*)
19       (format "\n... [truncated, \~a more characters]"
20             (- (string-length output) *max-total-output-length*)))
21     output))

```

Figure 3.17. Function to collect and format tracked variables

categorization of all natural language instructions no matter what part of the Promo program is changed. As we explain in Section 3.5.1, the models with non-zero temperature or the models without an adjustable temperature parameter could output different results every time, even when the prompt is the same. To ensure the consistency of the categorization and to avoid different categorization results caused by unrelated changes in the program, the language server sets the temperature parameter to 0 when querying. Some previous studies use methods like voting by multiple models to improve consistency. However, such methods are essentially single LLM usage.

Since the language server will query the LLM every time when conducting the diagnostic, it's costly and unnecessary to analyze the natural language instruction every time the user types a character. To minimize unnecessary API calls to the LLM during typing, we implemented a debouncing mechanism in the Language Server. The ambiguity check is triggered only after the user has stopped typing for a predefined interval of 0.5s.

When a diagnostic sweep is finally triggered, the server identifies all bracketed instructions within the source text. Since a single file may contain numerous natural language instructions, sequential API calls will lead to a very long waiting time. To mitigate this, rather than processing the natural language instructions in a linear loop, we use a parallel query mechanism to query the LLM in parallel.

The response time of the language server is important for the user experience. While the latency is mainly caused by the inference time of the model, the choice of the model affects the response time greatly. The Promo language server uses the Gemini 3 Flash model to balance accuracy and latency.

The language server queries the model to categorize the natural language instructions, and the model responds with a JSON object containing two properties. The first property is "type", an enum value with string values "pass", "warning" or "error". The second property is "message", a string to explain the categorization result. The language server will parse the response and emit diagnostics accordingly. We introduced the concept of an LLM's structured output in Section 3.5.1. Structured output is ideal for such a structured classification task. Since the Gemini series supports structured output by providing a JSON schema as an additional parameter, we use the structured output feature to ensure consistency of the response format.

Chapter 4

Experiments

We conduct experiments to evaluate the 3 mechanisms proposed in Chapter 3. First, we test whether and how well the Promo interpreter maintains position control. We do this experiment by executing Promo programs on the Promo interpreter using the strategy of whole expression replacement. The number of position control verification failures and the success rate of execution were recorded. In comparison, we conduct experiments on the Cursor inline chat (Inline Edit) feature under similar conditions. For runtime information, we want to assess whether providing runtime information can improve the accuracy of code generation. We design an experiment executing Promo programs on our Promo interpreter under 2 configurations. For the different configurations, the LLM generates code with or without runtime information. We examine whether runtime information helps by comparing the success rate of execution under different configurations. For the ambiguity checker, its ability to capture possible ambiguity is assessed by comparing the success rate of programs with different diagnostic results. If the programs with error or warning account for a lower success rate than the programs without error or warning, we consider that the ambiguity checker can somewhat detect the ambiguity. The following content of this chapter has 4 parts. First, we introduce how the dataset is constructed. Then, we describe the experiments on the 3 mechanisms.

4.1 Dataset Construction

This section describes how we constructed the dataset used in our experiments. All three experiments in this chapter share a common base of 48 Promo programs, though Section 4.2 uses a subset of 18 programs randomly selected.

We constructed the dataset by transforming open-source Racket programs into Promo programs. Original Racket programs are collected from Rosetta Code and open-source repositories [24–43]. Each transformation involves selecting one or more S-expressions or S-expression blocks from the original Racket code and replacing them with natural language instructions enclosed in square brackets. To create test cases, we employed two selection strategies. One is to randomly select expressions using a tool that parses the Racket source, identifies all parenthesized expressions, and picks a subset at random. However, we also manually selected some expressions to ensure coverage of cases we expected to be representative or challenging. Once the target

expressions were identified, we used an LLM to convert the original Racket code fragments into natural language instructions in order to reduce bias.

Some original Racket programs included test suites written with the rackunit testing framework, some contained test expressions without using rackunit, and a few had no tests. We normalized the dataset so that every program has test cases expressed in rackunit. For programs that already used rackunit, we kept the tests unchanged. For programs with non-rackunit tests, we rewrote them into the rackunit format. And for programs without any tests, we used an LLM to generate a test suite.

Table 4.1 lists all 48 programs in the dataset together with a brief description and their line counts. The programs span a range of domains including algorithmic problems such as closest-pair computation and SAT solving, data structure implementations like doubly linked lists and dynamic arrays, parser utilities, cryptographic helpers, and domain-specific tools. Line counts range from 11 to 281.

For the position control experiment in Section 4.2, we randomly sampled 18 programs from this pool to keep the number of manual Cursor trials manageable. The runtime information experiment in Section 4.3 and the ambiguity checker experiment in Section 4.4 both use the full set of 48 programs.

4.2 Position Control

As introduced in Section 3.1, the Promo interpreter inspects the generated top-level expressions corresponding to the top level expressions that contain bracketed instructions. The interpreter checks whether code outside brackets is modified by the LLM. New queries will be issued if the check fails. Here we evaluate our proposal by executing Promo programs on the Promo interpreter. We collect how many repeated code generation queries are invoked. We also check if the final generated result obeys position control if the repetition limit is reached.

4.2.1 Setup

We prepared 18 test cases for this experiment. Each test case is a Promo program with one or more natural language instructions in brackets. For the Promo interpreter, we ran each program 10 times. We record whether position control was maintained and whether the program executes correctly. We also count how many repeated LLM queries were triggered by position control violations.

For comparison, we conduct a similar experiment on Cursor. The source programs are the same as in the experiments on the Promo interpreter. For each program, we use Cursor to generate corresponding Racket code for the natural language instructions in the order of occurrence. For each bracketed instruction, we first replace the whole instruction with a space, then select the space and invoke the inline chat (Inline Edit) feature of Cursor. We use the original natural language instruction as the prompt for code generation. After all natural language instructions are replaced, we check if code outside brackets is modified by Cursor. The processed Racket program will be executed by the Racket interpreter to check the correctness. In this experiment, we use the same model Claude Sonnet 4.5.

Case	Brief description	LOC
sudoku_validator	Sudoku board utilities and solver scaffolding.	163
island_perimeter	Compute the perimeter of a 0/1 island grid.	56
lemonade_change	Greedy change-making for a lemonade stand.	44
max_ascending_sum	Search for the maximum sum of an ascending subarray.	47
alternating_sign_rearrange	Rearrange an array so positives and negatives alternate.	49
closest_pair	Divide-and-conquer closest pair of 2D points.	95
chinese_zodiac	Compute Chinese zodiac stems/branches for given years.	42
matrix_inverse	Wrapper around matrix inversion for boxed arrays.	11
dna_base_count	Count nucleotide bases and pretty print DNA sequences.	45
dna_mutation_simulator	Mutate random DNA strings and report base counts.	78
doubly_linked_list_ops	Mutable doubly linked list insert/remove helpers.	60
elementary_cellular_automata	Simulate elementary cellular automata.	128
sexp_to_latex	Convert S-expressions into LaTeX math markup.	276
run_length_encoding	Encode/decode strings with run-length compression.	70
poly_struct	Polynomial struct guards, printing, and conversions.	82
knn_classifier	KNN classifier.	66
case_conversion_suite	Convert strings across kebab, snake, and camel cases.	35
avro_codec_tests	Avro codec construction with round-trip property tests.	195
elliptic_curve_multiply	Scalar multiplication on an elliptic curve point.	29
bitcoin_address_validation	Validate Base58Check Bitcoin addresses.	51
adapton_testing_tools	Utilities for inspecting Adapton merge-sort structures.	219
continued_fraction	Continued-fraction combiner.	78
family_tree_queries	Family tree dataset with queries for ages and relations.	281
cieede2000_color_delta	Compute CIEDE2000 color differences.	82
mbox_parser	Lazy parser for mboxrd mailbox files.	36
sat_solver	Skeleton SAT solver over parsed CNF input.	29
root_finding	Find roots of a function.	34
secp256k1_point_multiply	Point addition/multiplication on secp256k1.	57
ulid_generator	ULID generator with monotonic randomness handling.	97
digit_remap_difference	Max/min digit remapping difference.	53
array_marking_score	Score array after marking neighbors.	58
prefix_suffix_pairs	Count strings that are both prefix and suffix of others.	36
dynamic_shortest_path_queries	Track shortest path lengths after edge insertions.	48
block_word_checker	Decide if a word can be built from letter blocks.	42
msk_iam_auth	AWS MSK IAM SASL client scaffolding.	227
markov_chain_generator	Markov chain construction and sampling utilities.	269
dynamic_array_impl	Dynamic array type implementing array protocols.	118
password_generator	Rosetta-style password generator solution.	49
brainfuck_interpreter	Brainfuck language interpreter.	109
puzzle	Numeric equation puzzle.	85
resonant_collinearity_solver	Antenna alignment solver.	127
reverse_string_gender	Reverse gender of the text.	87
balanced_ternary_arithmetic	Balanced ternary conversion and arithmetic helpers.	66
shunting_yard	Shunting-yard expression parser implementation.	51
historian_hysteria	Compute paired list distances and similarity score.	112
mull_it_over	Parse mul(x,y) instructions.	94
closest_partition	Closest-point partitioning.	100
posix_manual_data	POSIX manual data packaging and accessor utilities.	72

Table 4.1. All 48 Promo programs used in the runtime-information experiments, with short descriptions and line counts.

4.2.2 Results

Table 4.2 shows the results of all cases. The “Cursor PC” column indicates what percentage of runs maintained position control (out of 10 runs), and “Cursor Code” shows how many runs produced correct code. For the interpreter, we report the success rate of programs that passed all tests and the total number of repeat queries needed across all 10 runs. Because the Promo interpreter fails to control position only for case `lemonade_change` (in all runs), the table does not show the success rate of the Promo interpreter’s position control.

Case	Cursor PC	Cursor Code	Interpreter	Repeats
<code>sudoku_validator</code>	0%	50%	10/10	0
<code>island_perimeter</code>	100%	100%	10/10	0
<code>lemonade_change</code>	0%	100%	8/10	20
<code>max_ascending_sum</code>	100%	100%	9/10	0
<code>alternating_sign_rearrange</code>	100%	100%	10/10	0
<code>closest_pair</code>	100%	0%	3/10	0
<code>chinese_zodiac</code>	80%	90%	10/10	0
<code>matrix_inverse</code>	100%	100%	10/10	0
<code>dna_base_count</code>	100%	100%	4/10	0
<code>family_tree_queries</code>	100%	100%	10/10	4
<code>continued_fraction</code>	100%	100%	10/10	0
<code>dna_mutation_simulator</code>	100%	100%	10/10	0
<code>doubly_linked_list_ops</code>	100%	100%	10/10	0
<code>elementary_cellular_automata</code>	0%	100%	10/10	0
<code>sexp_to_latex</code>	100%	100%	9/10	0
<code>run_length_encoding</code>	100%	100%	10/10	2
<code>poly_struct</code>	100%	0%	10/10	0
<code>knn_classifier</code>	100%	0%	8/10	0

Table 4.2. Position control comparison between Cursor and the Promo interpreter across 18 test cases.

4.2.3 Analysis

As shown in the results, there are cases where Cursor keeps failing to maintain position control. For `sudoku_validator`, `lemonade_change`, and `elementary_cellular_automata`, Cursor’s position control rate was 0%. This means that in every single run, the LLM modified code outside the designated edit region. By contrast, the Promo interpreter handled these cases much better. `sudoku_validator` and `elementary_cellular_automata` both achieved 10/10 success without any retries. The case of `lemonade_change` required 20 retries across 10 runs, averaging 2 retries per run. But this case still achieved 8/10 successful executions.

Position control and code correctness show no significant correlation. The `closest_pair` and `poly_struct` cases illustrate this clearly. For `closest_pair`, Cursor maintained position

control in all runs but produced correct code in 0 of them. Similarly, `poly_struct` had perfect position control with Cursor but zero correct outputs. These cases suggest that even when the LLM respects the edit span limitations, the generated code may still be semantically wrong. The Promo interpreter performs better on these cases (3/10 and 10/10 respectively), likely because it provides runtime context that helps the LLM understand what code is actually needed.

The retry mechanism adds little overhead in most cases. Among 18 test cases, only 3 required any retries at all. For most programs, the LLM generated code with position well controlled on the first attempt. This means the enforcement mechanism does not significantly slow down the workflow.

One limitation is that the retry mechanism has a maximum limit. If the LLM fails to maintain position control beyond this limit, the interpreter will accept the last generated code whether or not position control is maintained. In our experiments, this limit is reached only in a minority of cases.

4.3 Runtime Information

The Promo interpreter uses runtime information to improve the performance of code generation, while typical inline AI tools use the static text only. This section evaluates whether runtime context actually helps the model generate correct Racket code.

4.3.1 Setup

As introduced in Section 4.1, we use the same dataset for Sections 4.2 and 4.3. It contains 48 Promo programs, and each program has one or more bracketed NL instructions and a test suite.

To evaluate the effectiveness of providing runtime information, we compare the accuracy of the generated code with and without runtime information. We execute Promo programs with the Promo interpreter and check whether they run successfully. Here, by “successful” we mean the program executes without runtime error and all test cases pass. The 2 configurations here are differentiated by whether the runtime information is provided or not. For each Promo program under each configuration, we execute the program 10 times and count the success ratio.

Because LLM outputs may vary, each program is executed 10 times under each configuration. In total, this yields 480 executions with runtime information and 480 executions without. We report success rates as “x/10” for each program and aggregated success rates over all runs.

4.3.2 Results

Table 4.3 summarizes the success ratio of each program with and without runtime information. Overall, providing runtime information improves the aggregate success rate from 67.7% (325/480) to 81.0% (389/480). Runtime context yields 64 more successful executions in 480 runs, and a +13.3 percentage-point gain.

Case	With runtime	Without runtime	Change
<code>chinese_zodiac</code>	100% (10/10)	0% (0/10)	+100 pts
<code>matrix_inverse</code>	100% (10/10)	0% (0/10)	+100 pts
<code>case_conversion_suite</code>	100% (10/10)	0% (0/10)	+100 pts
<code>avro_codec_tests</code>	100% (10/10)	0% (0/10)	+100 pts
<code>poly_struct</code>	80% (8/10)	20% (2/10)	+60 pts
<code>knn_classifier</code>	100% (10/10)	40% (4/10)	+60 pts
<code>elliptic_curve_multiply</code>	60% (6/10)	0% (0/10)	+60 pts
<code>bitcoin_address_validation</code>	40% (4/10)	0% (0/10)	+40 pts
<code>dna_base_count</code>	40% (4/10)	0% (0/10)	+40 pts
<code>closest_pair</code>	30% (3/10)	0% (0/10)	+30 pts
<code>adapton_testing_tools</code>	20% (2/10)	0% (0/10)	+20 pts
<code>doubly_linked_list_ops</code>	100% (10/10)	90% (9/10)	+10 pts
... (26 cases)	+0 pts
<code>max_ascending_sum</code>	90% (9/10)	100% (10/10)	-10 pts
<code>sexp_to_latex</code>	90% (9/10)	100% (10/10)	-10 pts
<code>lemonade_change</code>	80% (8/10)	100% (10/10)	-20 pts
<code>shunting_yard</code>	60% (6/10)	100% (10/10)	-40 pts

Table 4.3. Success rates with and without runtime information, sorted by improvement.

4.3.3 Analysis

While the improvement is significant, most programs are unaffected; many programs are 10/10 in both settings. Several programs improve a lot when runtime information is provided. There are four cases that go from 0/10 to 10/10: `chinese_zodiac`, `matrix_inverse`, `case_conversion_suite`, and `avro_codec_tests`. There are also a few downgraded cases, such as `shunting_yard` with a success ratio from 10/10 down to 6/10.

Runtime information helps most when the instruction depends on values that are hard to infer statically. The evidence is the group of programs that completely fail without runtime context but become reliable with it like the cases `chinese_zodiac` and `avro_codec_tests`. In these cases, the static code around the instruction is not enough for the model to confidently choose the right transformation. Concrete runtime values and their shapes remove a lot of possible guesses.

The benefit is concentrated in a minority of programs. Out of 48 programs, runtime information improves 12 cases, worsens 4, and leaves 32 unchanged. This distribution matches the intuition that runtime context is not universally necessary: when the instruction is self-contained (or the surrounding code already pins down the intent), the model succeeds with or without additional dynamic state.

Runtime information can also degrade accuracy. Four cases regress when runtime values are included, with `shunting_yard` showing the largest drop (-40 points). A likely explanation is that concrete runtime values can bias the model toward overly specific code (or a shortcut that fits the observed snapshot) instead of the general algorithm the tests require. Another possibility is

that the runtime-variable block adds distracting context, especially when the relevant variables are not clearly distinguished from irrelevant ones. This motivates tighter variable selection and formatting and suggests that providing more information does not guarantee an improvement in accuracy.

Finally, in the runtime-enabled setting, the variable-selection component reduces prompt length. Across the dataset, the selector removes 2.55 variables on average from the runtime snapshot before code generation. Although this experiment does not isolate selection as a separate factor, it indicates that the interpreter rarely needs to expose the full environment to get the observed gains.

4.4 Ambiguity Checker

This section evaluates the ambiguity checking feature of the Promo language server. We examine whether the warnings and errors produced by the language server correlate with the actual success rate of execution by the interpreter.

4.4.1 Setup

We conduct an experiment on the ambiguity checking feature of the Promo language server. The dataset we use is the same as in Section 4.3. Each program contains one or more bracketed natural language instructions along with test cases. We conduct ambiguity checking on 48 Promo programs with the language server. Then the 48 programs are classified based on the results of the ambiguity checking. They are grouped into 3 classes: programs that have no warnings or errors for the natural language instructions; programs that contain no errors but at least one warning; and programs that contain at least one error. Then we compare the grouping result with the success ratio in Section 4.3. By comparing the success ratio of each class, we evaluate the alignment between the Promo language server and the Promo interpreter.

We classify each program into one of three categories. “Pass” means that the program contains no NL instruction flagged as a warning or error. “Warning” means that the program contains at least one NL instruction flagged as a warning, but no errors. “Error” means that the program contains at least one NL instruction flagged as an error.

4.4.2 Results

The classification result shows that most programs fall into the “no warning” category, while only a few trigger warnings or errors from the language server.

Category	Programs	Total Runs	Successful Runs	Success Rate
No warning/error	40	400	327	81.75%
Warning (no error)	6	60	46	76.7%
Error	2	20	16	80.0%

Table 4.4. Success rates grouped by ambiguity checker classification.

The “no warning” group has the highest success rate at 81.75%. Programs with warnings have a lower success rate of 76.7%. The success rate of the “error” group is 80%, although the group has a very small number of programs.

4.4.3 Analysis

The results show a gap between programs with no warnings and those with warnings. The difference in success rate suggests that the language server’s warning detection captures part of the ambiguous natural language instructions. However, the relationship is not strong enough. There are several possible factors causing this. The sample size is relatively small in warning/error classes. There are only 6 programs that trigger warnings and 2 that trigger errors. The results of these groups are very vulnerable to noise. A single program or instruction can affect the percentages significantly.

Also, an ambiguous instruction can still be interpreted correctly by the LLM, especially considering the advancement of current LLMs. The ambiguity checker detects potential ambiguity rather than guaranteeing failures. The 76.7% success rate in the warning class means that most programs still work when the instruction is flagged.

The two programs classified as having errors achieved 80% success, which is higher than the warning group. Looking at these specific cases, the “errors” flagged by the language server appear to be about other issues rather than ambiguity.

Runtime information compensates for ambiguity. Since these execution results use the runtime-enabled configuration, the additional context may help the model to clarify the instructions that are unclear. This could explain why instructions with warnings or errors often succeed. The interpreter sometimes handles these by inferring the missing information from runtime context.

Another observation is that the language server’s classification is based on a single static analysis pass, while the interpreter runs each program 10 times with potentially different LLM outputs. An instruction flagged as ambiguous might fail in some runs but succeed in others. This depends on how the model interprets it on that particular call.

4.4.4 Limitations

The current experiment has some limitations. Most programs, 40 out of 48, have no warnings. This makes the proportion and size of the “warning” and “error” groups very small. A more balanced or larger dataset with more ambiguous instructions may give clearer and more convincing results. Second, we only compare success/failure execution results. A deeper analysis could look at whether warnings correlate with specific failure modes. This could lead to a clearer conclusion about whether the language server catches the real reason for execution failures. Third, the language server uses an LLM (Gemini 3 Flash) different from the interpreter, which uses Claude Sonnet 4.5, which introduces a potential mismatch between different models. Despite these limitations, the warning group’s lower success rate is consistent with the intended purpose of the ambiguity checking feature. It could give programmers early feedback about problems before executing the code.

Chapter 5

Conclusion and Future Work

5.1 Summary

This thesis presented Promo, a system that embeds natural-language instructions directly into the Racket programming language. Developers write instructions enclosed in square brackets and the Promo interpreter translates these instructions into executable Racket code at runtime with the help of a large language model.

We identified three practical challenges in existing AI-assisted coding tools. First, current tools often fail to respect the boundaries of the region a developer intends to modify. Second, these tools rely only on static source code and cannot access runtime values that may be essential for generating correct code. Third, ambiguous natural language instructions lead to misalignment between developer intent and generated code, but developers receive no early warning about such ambiguity.

Promo tries to address these challenges through three features. Position control ensures that the LLM modifies only the bracketed parts of a top-level expression by either verifying that non-bracketed parts remain unchanged or by constructing the final expression from a template that never exposes the surrounding code to modification. Runtime information provides the LLM with formatted descriptions of tracked variables in scope at the point of generation, enabling it to understand data structures and available helper functions that may not be inferred from static text alone. The ambiguity checker inspects each bracketed instruction and emits warnings or errors to the editor when an instruction is diagnosed as vague.

Experiments on a dataset of 48 Promo programs assessed these features. Position control achieved adherence to edit boundaries, whereas the same programs processed through Cursor’s inline edit feature frequently violated position constraints. Providing runtime information improved the overall success rate. Several programs that completely failed without runtime context become reliable when runtime values were included. The ambiguity checker showed a correlation between instructions flagged “warning” and lower execution success rates. This suggests that the diagnostics capture part of the ambiguity that leads to generation failures.

5.2 Future Work

Several directions remain open for future development. One natural extension is to enhance the language server with interactive features or quick-fix support. The current ambiguity checker reports diagnostics but leaves the manual refinement work to the developer. A more helpful workflow would offer concrete suggestions for clarifying an ambiguous instruction, such as proposing additional constraints or asking questions that the developer can answer with a single click. The Language Server Protocol already supports code actions that can insert or replace text in the editor.

Another important direction is to add inspection and restoration mechanisms for incorrectly generated code. At present, when the interpreter produces code that fails at runtime or does not pass tests, the developer may either re-run the program or manually edit the instruction. A more robust approach would let developers inspect the generated code before or after execution, and compare multiple candidate generations side by side. This could be integrated into the language server to display the generated Racket code corresponding to each bracketed instruction. Another idea is to support feedback loops where execution errors are automatically fed back to the LLM for a corrected generation attempt.

Beyond these directions, future work could explore extending the approach to other programming languages. The core ideas of position control and runtime context are not specific to Racket. Improving variable selection through more sophisticated relevance scoring or learning from developer feedback is also worth investigating.

Publications and Research Activities

- (1) Yugu Xie, Feng Dai, Tetsuro Yamazaki, Shigeru Chiba. An Attempt of Combining Natural Language Instructions with Traditional Programming. 第27回プログラミングおよびプログラミング言語ワークショップ, 2025.3.5-7.
- (2) Yugu Xie, Feng Dai, Tetsuro Yamazaki, Shigeru Chiba. Towards Combining Natural-language Instructions With Programming. 日本ソフトウェア科学会第 34 回大会, 2025.9.3-5.

References

- [1] Cursor Documentation. Inline edit. <https://cursor.com/docs/editor/inline-edit>. Accessed: 2026-01.
- [2] Cursor Documentation. Keyboard shortcuts. <https://cursor.com/docs/editor/keyboard-shortcuts>. Accessed: 2026-01.
- [3] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. The impact of ai on developer productivity: Evidence from github copilot, 2023.
- [4] Joel Becker, Nate Rush, Elizabeth Barnes, and David Rein. Measuring the impact of early-2025 ai on experienced open-source developer productivity, 2025.
- [5] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI EA '22, New York, NY, USA, 2022. Association for Computing Machinery.
- [6] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of github copilot’s code contributions. *Commun. ACM*, 68(2):96–105, January 2025.
- [7] Katsumi Okuda and Saman Amarasinghe. Askit: Unified programming interface for programming with large language models. In *Proceedings of the 2024 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '24, page 41–54. IEEE Press, 2024.
- [8] Honghua Dong, Qidong Su, Yubo Gao, Zhaoyu Li, Yangjun Ruan, Gennady Pekhimenko, Chris J. Maddison, and Xujie Si. APPL: A prompt programming language for harmonious integration of programs and large language model prompts. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1243–1266, Vienna, Austria, July 2025. Association for Computational Linguistics.
- [9] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. Prompting is programming: A query language for large language models. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023.
- [10] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather

44 References

- Miller, Matei Zaharia, and Christopher Potts. Dspy: Compiling declarative language model calls into self-improving pipelines, 2023.
- [11] G. Poesia et al. Synchromesh: Reliable code generation from pre-trained language models. In *International Conference on Learning Representations*, 2022.
- [12] Ansong Ni, Srini Iyer, Dragomir Radev, Ves Stoyanov, Wen-tau Yih, Sida I. Wang, and Xi Victoria Lin. Lever: learning to verify language-to-code generation with execution. In *Proceedings of the 40th International Conference on Machine Learning, ICML'23*. JMLR.org, 2023.
- [13] Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2019.
- [14] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d’Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.
- [15] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests, 2022.
- [16] Zongjie Li, Chaozheng Wang, Zhibo Liu, Haoxuan Wang, Dong Chen, Shuai Wang, and Cuiyun Gao. Cctest: Testing and repairing code completion systems. In *Proceedings of the 45th International Conference on Software Engineering, ICSE '23*, page 1238–1250. IEEE Press, 2023.
- [17] Zhongjun Ding, Yin Lin, and Tianjing Zeng. Ambisql: Interactive ambiguity detection and resolution for text-to-sql, 2025.
- [18] S. Jia et al. Automated repair of ambiguous problem descriptions for llm-based code generation, 2025.
- [19] Haau-Sing (Xiaocheng) Li, Mohsen Mesgar, André Martins, and Iryna Gurevych. Python code generation by asking clarification questions. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14287–14306, Toronto, Canada, July 2023. Association for Computational Linguistics.
- [20] Aditey Nandan and Viraj Kumar. Ambiguity resolution with human feedback for code writing tasks, 2025.
- [21] Jane Yen, Tamás Lévai, Qinyuan Ye, Xiang Ren, Ramesh Govindan, and Barath Raghavan. Semi-automated protocol disambiguation and code generation. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference, SIGCOMM '21*, page 272–286, New York, NY, USA, 2021. Association for Computing Machinery.

- [22] Weiwei Sun, Lingyong Yan, Xinyu Ma, Shuaiqiang Wang, Pengjie Ren, Zhumin Chen, Dawei Yin, and Zhaochun Ren. Is chatgpt good at search? investigating large language models as re-ranking agents, 2024.
- [23] Microsoft. Language server protocol specification. <https://microsoft.github.io/language-server-protocol/>. Accessed: 2026-01.
- [24] Rosetta Code. <https://rosettacode.org/wiki/Category:Racket>.
- [25] secp256k1. <https://github.com/MohamedLEGH/secp256k1>.
- [26] racket-avro: A racket implementation of the apache avro serialization format. <https://github.com/Bogdanp/racket-avro>.
- [27] racket-ulid: Universally unique lexicographically sortable identifiers for racket. <https://github.com/Bogdanp/racket-ulid>.
- [28] racket-kafka: A kafka client for racket. <https://github.com/Bogdanp/racket-kafka>.
- [29] simple-polynomial. <https://bitbucket.org/derend/simple-polynomial>.
- [30] adventofcode. <https://github.com/yangh/adventofcode>.
- [31] competitive programming. <https://github.com/usefulmove/cp>.
- [32] Exercises in programming style. <https://github.com/cndreisbach/exercises-in-programming-style>.
- [33] Aoc-2024-in-racket. <https://github.com/swishtail/AoC-2024-in-Racket>.
- [34] brainfuck-interpreter. <https://github.com/swishtail/brainfuck-interpreter>.
- [35] rkt-llm-cli: Yet another llm repl in racket. <https://pkgs.racket-lang.org/package/simple-polynomial>.
- [36] elementary-cellular-automata. <https://github.com/swishtail/elementary-cellular-automata>.
- [37] Sat.rkt: A simple sat solver based on dpll written in racket. <https://github.com/Kraks/SAT.rkt>.
- [38] mboxrd-read. <https://github.com/jbclements/mboxrd-read>.
- [39] array: Generic arrays for racket. <https://github.com/rvs314/array>.
- [40] behavior: Behavioral models for racket. <https://github.com/johnstonskj/behavior>.
- [41] racket-scrapyard: Web scraping (caching and parsing) helper library. <https://github.com/lassik/racket-scrapyard>.

46 References

- [42] roxy-equation: Convert racket expression to latex equations. <https://github.com/chemPolonium-443/roxy-equation>.
- [43] Xrom. <https://github.com/seanwevans/XORM>.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Shigeru Chiba, for his gracious and patient guidance. I would also like to thank Dr. Yamazaki, Dr. Dai and Mr. Li for their kind help and advice. Finally, thanks to all the others who have supported me.

