

Towards Combining Natural-language Instructions With Programming

Yugu Xie, Feng Dai, Tetsuro Yamazaki, Shigeru Chiba

Large Language Models (LLMs) can now draft sizable chunks of code, yet today’s assistants still (i) operate at fixed or uncontrolled granularity, (ii) rely on static prompt context, and (iii) sometimes drift from the developer’s intent. We propose a paradigm that treats natural-language (NL) instructions as first-class code. Our prototype, implemented as a Racket extension, lets developers embed NL instructions inside ordinary code. Any expression wrapped in square brackets, for example, `(+ 1 2 [sum remaining numbers to 10])`, is translated on demand by an LLM into executable Racket code at runtime. The interpreter captures its dynamic environment, compresses that snapshot into a model-friendly string, inserts the string into a prompt, queries the LLM, then splices the reply and evaluates it in the original namespace. To mitigate misalignment, we implement a code inspection mechanism for NL instructions. An evaluation is given to verify the viability of the proposal.

1 Introduction

Large language model (LLM) assistants have made remarkable progress in generating code, yet today’s tools remain challenging: they operate at fixed or hard-to-control granularity, rely solely on static source context, and often drift from the developer’s true intent. This paper proposes PROMO, a Racket extension that treats natural-language (NL) instructions as first-class code. Any expression wrapped in square brackets, such as `(+ 1 2 [sum remaining numbers to 10])`, is translated on demand by an LLM into executable Racket code at runtime. PROMO lets developers specify exactly which sub-expression, statement, or function should be generated. PROMO captures the live runtime environment and injects it into the prompt, enabling context-aware synthesis. PROMO can mitigate misalignment through an automatic NL-inspection mechanism that flags ambiguous instructions. An experiment on 42 open-source Racket programs is planned to verify whether runtime-aware generation improves syn-

tactic correctness and semantic preservation over a static baseline.

2 Challenges in LLM-based Code Generation

The application of LLMs has seen significant progress in software engineering. Among the various AI-powered tools, code generation assistants represent one of the most popular and rapidly advancing fields. Traditionally, code generation from natural-language specifications is approached in two main ways. The first is through in-IDE code completion, where the prompt context is implicitly derived from the surrounding code. The second is through chat-based interfaces such as ChatGPT, where users explicitly define the prompt, offering greater flexibility but also imposing a higher cognitive load.

However, recent advancements have led to a magnificent fusion of two approaches within modern AI-driven IDEs. A typical workflow now involves developers selecting a code snippet or positioning the cursor and then providing a natural-language instruction. The IDE combines this instruction with the relevant code context to form a rich prompt for LLMs. Although such new tools have overcome many previous limitations, significant challenges re-

Yugu Xie, Feng Dai, Tetsuro Yamazaki, Shigeru Chiba, Graduate School of Information Science and Technology, The University of Tokyo.

```
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:],
```

Fig. 1 A try with the JetBrains AI Assistant

```
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]
```

Fig. 2 Modified code by JetBrains AI Assistant

main.

First, the granularity of code generation is often fixed or difficult to control. Developers may want to generate a small expression, a single statement, or an entire function block. However, current assistants typically do not offer strict control of the position and level of the generated code, which may disrupt the existing code structure. There is a lack of fine-grained controllability that allows developers to seamlessly specify exactly where and how much code should be generated. This leads to manual adjustments or refactoring after generation. This lack of fine-grained control is illustrated by a simple experiment with JetBrains AI Assistant (version 251.23774). As shown in Fig. 1, we have a function that returns a tuple of three elements. Our 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 Fig. 2, 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 instance exemplifies the problem in current tools: the AI’s operational granularity does not match the user’s specific selection, forcing the generation that often deviates from the user’s initial intention.

Second, these tools are based on static information. They rely only on static context of source code and the prompt and have no access to the runtime environment of the program, such as dynamic values of variables and current states of data

structures. This static-only view limits their ability to generate context-aware code. For instance, a developer’s instruction might depend on the content of a list that is only determined at runtime, a scenario that is completely invisible to today’s assistants. In the example shown in Fig. 1 and Fig. 2, the developer’s intention is to sort the list. However, the correct sorting logic depends on the actual content of that file at runtime. For example, if a column in the file contains numeric strings such as 10 or 2, it should be sorted numerically. Current AI assistants based on static context have no way to know this.

Finally, the generated code often suffers from misalignment with the developer’s true intention. This happens because the translation from a high-level, sometimes ambiguous, natural-language instruction to a precise and formal code fragment is 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, defeating the purpose of using an AI assistant for efficiency. The same example in Fig. 1 also highlights this problem of misalignment. The user’s prompt, “sorted rows”, is ambiguous. It does not specify the sorting key or the order. The LLM is forced to make an implicit assumption. Since the prompt is ambiguous, the LLM might guess and sort the list by the first column, alphabetically, or numerically. The LLM implicitly makes the decision for the user, which may lead to an unintended result. In cases where the generated code is longer, there could be more unseen and unintended decisions automatically made by LLMs.

3 PROMO: natural-language Instructions as First-Class Racket Code

Given the challenges outlined in Section 2, developers still lack fine-grained, context-aware control over AI-driven code generation. Our extended Racket prototype, PROMO, bridges this gap by treating natural-language instructions as first-class citizens of the code. Racket language is a modern dialect of Lisp and a descendant of Scheme. We choose Racket to extend for its S-expression and wide usage. Any expression enclosed in square brackets, for example, (+ 1 2

Listing 1 A code example of Promo

```
(define (read-data-file filename)
  (with-input-from-file filename
    (lambda () (read))))

(define data (read-data-file ".../data.rkt"))

(print
 [the name of the item most
  semantically similar to dog in data])
```

Listing 2 First 10 lines of data.rkt used in Listing 1

```
((apple (0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 0.1))
 (pig (0.4 0.4 0.2 0.6 0.8 0.3 0.5 0.7 0.1 0.9))
 (dog (0.5 0.5 0.7 0.2 0.4 0.8 0.6 0.3 0.9 0.1))
 (cat (0.3 0.7 0.1 0.5 0.9 0.2 0.4 0.6 0.8 0.3))
 (bird (0.8 0.1 0.6 0.3 0.7 0.9 0.2 0.5 0.4 0.8))
 (fish (0.2 0.9 0.4 0.8 0.1 0.5 0.3 0.7 0.6 0.2))
 (horse (0.6 0.3 0.9 0.1 0.5 0.7 0.8 0.2 0.4 0.6))
 (cow (0.7 0.8 0.2 0.4 0.6 0.1 0.9 0.3 0.5 0.7))
 (sheep (0.9 0.2 0.5 0.7 0.3 0.4 0.1 0.8 0.6 0.9))
 (rabbit (0.1 0.6 0.8 0.9 0.2 0.3 0.5 0.4 0.7 0.1))
```

[sum remaining numbers to 10]), is translated by an LLM into concrete Racket code at runtime, with full access to the runtime environment. PROMO consists of an interpreter and a language server based on the Language Server Protocol to support NL inspection within the editors. The Language Server Protocol is a protocol for communication between the editor and the server that provides language-specific support. An editor extension implemented following the Language Server Protocol could provide language-specific features such as code inspection, auto completion suggestions, quick fixes, and code actions.

3.1 Syntax

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 these 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.

Listing 1 presents an example of PROMO, demon-

strating how a developer can find the item most semantically similar to a specific query within a dataset. The code snippet consists of three main parts: a helper function to load data, a statement that loads a vector dataset into the `data` variable (with its content shown in Listing 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 a dog in data], is a PROMO expression. At runtime, the PROMO interpreter recognizes this expression and sends its text content with other information (including the value of `data` from Listing 2) to an LLM. The LLM then returns executable Racket code, which the interpreter splices back into the program for immediate evaluation.

3.2 Core Mechanism of PROMO

3.2.1 Fine-Grained Granularity Control

Natural language fragments become more powerful when developers can decide exactly where they appear. With PROMO, the mere act of wrapping code in a pair of square brackets is enough to mark

```

1 (define (read-data-file filename)
2   (with-input-from-file filename
3     (lambda () (read))))
4
5 (define test-data (read-data-file "../data.rkt"))
6
7 (print )

```

the name of the item most semantically similar to dog in data

Fig. 3 Using an existing AI tool

the region that should be written in NL and synthesized by LLMs; nothing else about the surrounding program has to be changed. Because brackets are just another expression form, their extent is chosen by only selecting a span of text, typing [, writing the instruction, and closing with]. This seemingly minor convenience frees the programmer from extra prompt input or manual modification.

The contrast with existing tools is illustrated by Fig. 3. Fig. 3 shows an attempt to use the NL instruction in Listing 1 as the prompt for JetBrains AI Assistant (version 251.23774). As shown in Fig. 3, we place the cursor after `print`, invoked the IDE’s “generate code” command, and entered the prompt “the name of the item most semantically similar to dog in data”. Against the user’s intent, which is to write a single expression as the argument for `print`, the assistant inserts a function call to `(find-closest-to 'dog data)` inside the `print` function. It also appends new top-level function definitions of `find-closest-to` and its helper function `distance`.

PROMO gives users precise control over where AI assistance begins and ends. When the interpreter encounters an NL instruction, it constructs a clear, contextual prompt, including a brief instruction prefix and the NL fragment embedded in the surrounding program text with the insertion region marked. This improves code generation quality and ensures the intervention is tightly scoped and executed within the intended context.

3.2.2 Runtime-Aware Code Generation

In many tasks a useful NL instruction depends not only on static code but on values that exist only at runtime. Unlike conventional code generation assistants, which only see the source text, PROMO makes those dynamic values available to the LLM automatically.

Because the interpreter serializes the dynamic environment before every LLM call, each bracketed instruction can freely mention variables, and functions that are in scope at the moment of evaluation. This capability operates on two levels. The user may explicitly reference external definitions. In Listing 1, the instruction states “the item that is most semantically similar to dog in `data`”, where `data` is the list data loaded by the first statement. The interpreter serializes the runtime binding of `data` (a list consisting of 50 pairs) into the prompt, so the model has all information necessary to compute Euclidean distance and cosine similarity in ten dimensions. If the user has defined helper functions such as `cosine-distance`, these too are serialized so that the generated code may invoke them with or without explicit instruction.

A current limitation of this runtime-aware approach is its naive strategy. Our prototype currently serializes all user-defined bindings available in the current namespace and the full body of every function then appends them to the prompt. This inclusion can lead to unnecessarily large prompts, increasing latency and cost, and may even introduce noise that degrades the quality of the code generation. A direction for future work is to develop a more intelligent context filtering mechanism. Instead of serializing the entire namespace, an intelligent filter could employ techniques like semantic retrieval or a light LLM to identify and provide only the bindings that are relevant to the user’s NL instruction. This would improve the efficiency and precision of the runtime-aware approach.

3.2.3 Mitigating Misalignment

We propose the PROMO language server that provides inspection support for NL instructions. We implement an inspection prototype for Emacs following the Language Server Protocol (LSP).

Fig. 4 shows an example of this mechanism, the inspection output shows one warning and one error for the two NL instructions. Our PROMO language server is an intelligent Language Server Protocol implementation that automatically analyzes natural-language instructions embedded within the PROMO code.

It provides real-time feedback through three classification levels: “pass” for clear instructions, “warning” for ambiguous but interpretable content, and “error” for meaningless or irrelevant descrip-

```
(define (to-roman num)
!  [Convert it])
(define (calculate-area shape)
!! [sort the input list])
```

Error: " sort the input list " - The natural language 'sort the input list' is irrelevant to the function name

Fig. 4 An example of our NL inspection

```
##### Context
Natural language content extracted:
\|\|
{natural_language_content}
\|\|

### Source Code
\|\|
{source_code}
\|\|

### Task
Analyze the above natural language content and its associated source code. Provide
detailed feedback:
1. Is the natural language description sufficiently clear to infer the corresponding
logic in the source code?
2. If unclear, suggest improvements.
3. Classify the clarity as 'pass', 'warning', or 'error', and provide a very very short
and extracted explanation for your classification. If the content is ambiguous but the
intent may be inferred from the source code, make a 'warning' classification. If the
content is meaningless or irrelevant, make an 'error' classification.
***
```

Fig. 5 Prompt for NL instruction inspection

tions.

When issues are detected, the system can generate diagnostic messages, suggest improved replacements, and even provide interactive code actions to help developers clarify their natural-language instructions. The NL instruction inspection forms a mechanism that makes the LLM towards faithfully implement the programmer’s intent.

For the implementation of the inspection mechanism, the system scans for text enclosed in square brackets and uses LLM to evaluate the clarity and precision of these natural-language descriptions against their corresponding source code context. The system extracts every NL instruction and stores their text content and position for further processing and diagnosis. Fig. 5 shows the prompt that the server sends to the LLM. The `natural_language_content` is expanded to the specific NL content extracted to inspect, and the `source_code` is expanded to the whole program source code including the original NL instructions. The LLM is tasked to classify the clarity of the instruction and suggest improvements if necessary. We limit the output format to a JSON object that contains two `string` properties: the diagno-

sis `type` and `message`. Google’s Gemini 2.5 Flash Lite model is used to balance precision and response speed.

3.3 Implementation Details

The PROMO interpreter is implemented mainly in Python, with the Racket interpreter as its backend. Racket provides a C interface of its runtime system, which is originally responsible for the implementation of Racket. It also makes it possible to embed the Racket runtime into a program, such as our PROMO interpreter. The integration between Python and Racket is facilitated through a ctypes-based interface. We write a C program to initialize a Racket runtime instance and provide functions wrapping the original API. For instance, `scheme_namespace_require` is used to load modules, and `scheme_eval_string` is used to evaluate the expressions. Racket C-APIs are exported by using ctypes to load a shared library compiled and linked from the C program that consists of wrapped APIs. PROMO’s interpreter uses a Racket 8.13 instance.

The streamline overview of the PROMO interpreter is demonstrated in Fig. 6. First, the bracketed NL expressions are transformed into ordinary Racket. The transformed program are then split into top-level expressions. Each NL instruction embraced by square brackets is replaced by a Racket expression which assembles a prompt and calls the LLM. This Racket expression also uses the `eval` function to execute the LLM-generated Racket code. The split top-level Racket expressions will be evaluated in order utilizing the Racket C-API `scheme_eval_string`. The Racket expression transformed from the NL instruction calls two Racket functions defined by us: `collect-user-variables-as-string` and `get-user-defined-procedure-names`. They pro-

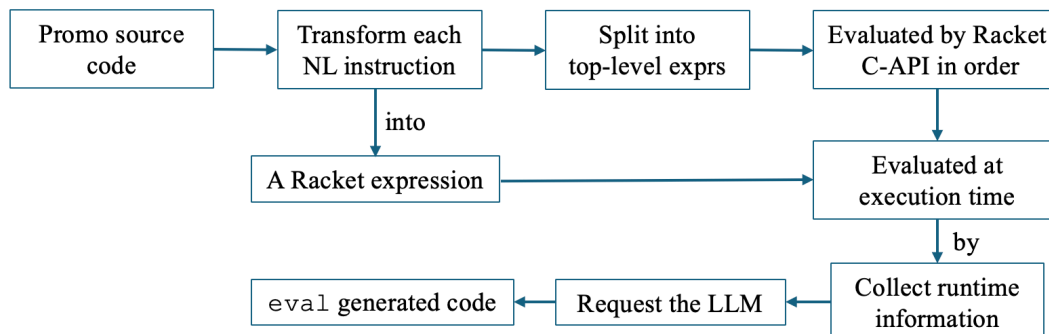


Fig. 6 An overview of the Promo interpreter

vide the runtime information of user-defined variables and functions. The formation of the prompt and the request to the LLM are conducted by a Python script `request_llm`. We invoke the Python script using Racket’s `system` function. The arguments of this Python script are: the text content of the NL instruction, the whole text of the source PROMO program, the string results of function `collect-user-variables-as-string` and function `get-user-defined-procedure-names`. The prompt concatenates an instruction prefix that explains the expected behavior of the LLM, the user’s NL fragment along with the source code and the serialized environment. For the Listing 1 example, the variable `data` and its value are transformed into an easy-to-read format, `read-data-file` and its function body are extracted. All of this information would be included in the prompt whether or not they are relevant to the NL instruction. The LLM is asked to generate Racket code that could replace the original NL instruction directly.

For the prompt, serializable values (numbers, booleans, strings, lists, vectors) are structured into understandable formats; function bodies are provided. The mechanism for providing runtime information operates through an introspection system. The approach involves creating a baseline reference by instantiating a fresh base namespace containing only built-in symbols, then comparing it against the current working namespace to identify user-defined entities. To provide comprehensive information of variables, we write Racket functions to convert data values into human-readable string representa-

tions. These formatting functions can recursively traverse data structures including lists, vectors, hashes, and custom structs, provide detailed string representations of values while preventing infinite recursion through depth limiting. To extract bodies of user-defined functions, we use Racket function `get-user-defined-procedure-names` to extract names of all user-defined functions. The extracted names are passed as a string argument to the Python script `request_llm`. As the PROMO source program is also provided as an argument, before generating Racket code, the Python script makes an LLM request to extract the bodies of user-defined functions, with the prompt including the source programs and the names of user-defined functions. The response would then be added to the code generation prompt.

Calls to the LLM are made through Anthropic’s API using the model Claude Sonnet 4. The implementation currently makes repeated calls to the LLM for repeating execution for the same NL instruction. An NL instruction could be interpreted multiple times like an NL instruction inside a reused function. Considering efficiency and consistency, a cache mechanism for the corresponding generated code would be important. However, direct reuse of generated code would lose the capability of handling data polymorphism. We here propose two possible approaches of future extension. A “lazy” approach is to invoke a new code generation call only when the cached code fails. A more positive approach is to decide whether to generate new code by comparing data changes between

two evaluations of NL instructions. For the positive approach, the LLM will be asked to append the variables used by generated code to the generated code. The data structures of these variables would be stored. Before the next execution of the same NL instruction, the stored data structures will be compared to the current data structures of these variables. If there is any difference, there will be a new code generation request to the LLM. If not, the system will try to reuse the cached code.

4 Evaluation Plan

We plan to evaluate how much providing runtime information contributes to generating correct code. We will compare the PROMO interpreter and the baseline, a modified PROMO prototype that does not provide runtime information to the LLM. When calling the LLM, the modified version of the interpreter forms the prompt based on static source text. The prototype proposed to evaluate is described in Section 3, which captures the runtime information in the prompt before each LLM call. Both variant systems query the same model with identical parameters.

We plan to use a self-collected dataset to drive our experiments. In total, we select 42 open-source Racket projects. They include 9 multi-file programs and 33 individual-file programs. For all 42 projects, the median Lines of Code (LoC) is 87.5 and the average is 214.5.

The programs in the dataset will be transformed by partly replacing Racket code with equivalent NL instructions. We randomly replace expressions or statement blocks with semantically equivalent NL instructions. The NL instruction will be handwritten or LLM-synthesized. We will give each program a comprehensive set of test cases.

We would compare the proposal and the baseline by the semantic preservation rate. It assesses whether PROMO programs (contains one or more NL instructions) could be executed without error and produce the same output as the original program by the interpreter. A run of a program is considered semantically preserving if the system loads and executes on all test cases without throwing any syntax or runtime errors, and for all test cases, the final output is identical to the output of the original Racket program.

36 of the programs (85.7%) already include their own example usage code. The other 6 programs have no test cases. For 2 complex applications, they already have their own comprehensive test suites. For other programs, we construct a set of input/output test cases. For each target function or primary entry point, we define a number of inputs and their expected outputs. The original usage of the target functions will be removed when testing. When the original repository contains test cases, we will remain them. If a project does not contain a test case or its test cases do not provide sufficient coverage, we will use the LLM to generate test cases. The capability of LLMs to generate test cases has been proven [1].

5 Related Work

Potriasaeva et al. proposed a low-code IDE plugin that allows novice programmers to solve exercises by describing the desired algorithm in NL and iteratively refining their prompt until the LLM-generated draft passes the built-in tests [4]. This work applies a similar fusion of natural-language text and traditional source code and adopts the idea of prompt refinement. However, this work makes traditional code an optional part of the source code, and the natural-language descriptions are mandatory. PROMO allows users to decide the level of natural-language instructions freely. While this work provides generated code for users to refine the prompt, PROMO provides direct explanation and suggestion by LLMs for users to refine the natural-language instructions.

Okuda et al. introduced *AskIt*, a domain-specific language that extends TypeScript and Python with two primitives, `ask` and `define`, to interact with LLMs [3]. *AskIt* provides an embedded DSL for a whole-function generation and direct task execution using LLMs. While it supports invoking LLMs at runtime, these calls are limited for direct task execution. The function generations are still finished at *AskIt*'s compile time, which provides a more efficient execution while has no runtime information for code generation compared to PROMO interpreter.

A frequently referenced work is *LMQL* [2], a query language that embeds logical constraints in-

side prompts. Although LMQL tries to combine natural-language text and traditional code seamlessly, it does not generate executable host-language code. Although both PROMO and LMQL interpret NL within a structured context, their implementation and application domains are different. PROMO is a tool for programming with natural-language, while LMQL is a tool for querying with natural-language.

6 Conclusion

PROMO demonstrates that embedding natural language instructions in code may overcome limitations of existing LLM-based code generation. Future work includes extending the implementation of NL instructions by non-code-generation ways and refining the inspection mechanism to provide interactive repair suggestions.

Disclosure

This paper contains content generated by Large Language Models, specifically OpenAI's O3 and

Google's Gemini 2.5 Pro.

References

- [1] Dilhara, M., Bellur, A., Bryksin, T., and Dig, D.: Unprecedented Code Change Automation: The Fusion of LLMs and Transformation by Example, *Proc. ACM Softw. Eng.*, Vol. 1, No. FSE(2024).
- [2] Kellner, L. B., Fischer, M., and Vechev, M.: Prompting is Programming: A Query Language for Large Language Models, *Proc. ACM Program. Lang.*, Vol. 7, No. PLDI(2023), pp. 1–28.
- [3] Okuda, K. and Amarasinghe, S.: AskIt: Unified Programming Interface for Programming with Large Language Models, *Proc. IEEE/ACM Int. Symp. on Code Generation and Optimization (CGO 2024)*, IEEE/ACM, 2024, pp. 41–54.
- [4] Potriasaeva, A., Dzialets, K., Golubev, Y., and Birillo, A.: Using a Low-Code Environment to Teach Programming in the Era of LLMs, *Proc. 2024 ACM Int. Computing Education Research V.2 (ICER '24 Vol. 2)*, New York, NY, USA, ACM, 2024, pp. 1–2.