東京大学
情報理工学系研究科 創造情報学専攻
修士論文

# In-Context Learning プロンプトの自動生成によって コード補完 LLM を新言語に対応させる手法

A Method for Adapting Code Completion LLMs to New Languages by Automatically Generating In-Context Learning Prompt

大渕 雄生
Obuchi Yuki

指導教員　千葉 滋 教授

2025 年 1 月

# 概要

大規模言語モデル（LLM）が十分な学習データを持たない新規プログラミング言語やドメイン固有言語（DSL）でも、LLM を用いたコード補完を実現する手法を本論文は提案する。従来の方法としては、特定タスクに LLM を対応させるためには、モデル全体を再学習するファインチューニングが一般的であるが、大量のコード例と計算資源を要するため、新言語の普及初期段階では適用が難しい状況が多い。加えて、LLM が生成したコードへのパーサーのエラーメッセージをプロンプトとしてフィードバックすることを繰り返すフィードバックループ方式では、誤生成を逐一修正しながら精度を高められる反面、プロンプトが肥大化して推論時間やトークン消費量が増大し、リアルタイムでの活用が難しくなる課題がある。

そこで本論文では、In-Context Learning（ICL）の枠組みを活用し、新言語の文法ドキュメントやサンプルコード、パーサが検出したエラーメッセージをフィードバックループ方式で蓄積して LLM を用いて圧縮する手法を提案する。具体的には、修正履歴を「なぜエラーが起き、どのように修正したか」という観点で要約し、簡潔化されたプロンプト（ICL プロンプト）としてモデルに提示する。圧縮されたプロンプトは新言語の注意点をまとめるため、初回推論から適切な補完を誘導しやすくなり、フィードバックループを多回繰り返すことなく精度を向上させることができると期待される。

複数の未知言語や DSL を対象とした実験では、エラー修正履歴を圧縮して組み込んだ ICL プロンプトを利用することで、文法正解率や BLEU スコアなどの精度指標がフィードバックループを利用するのと同程度に向上することが観察された。また、フィードバックループを用いる場合と比べてトークン数と推論時間を大幅に削減できることも示された。

# Abstract

This paper proposes a method for enabling code completion with large language models (LLMs) even for emerging programming languages or domain-specific languages (DSLs) that lack sufficient training data. Traditionally, to adapt an LLM to a specific task, fine-tuning the entire model has been common practice, but it often requires a large amount of code examples and computational resources. As a result, in the early stages of a new language' s adoption, it can be difficult to apply such an approach. Moreover, in the feedback loop method—where parser error messages generated from LLM output are fed back to the LLM as prompts—accuracy can be improved by iteratively correcting errors. However, these prompts can become excessively large, increasing inference time and token consumption, which poses challenges for real-time use.

In response, this paper leverages the In-Context Learning (ICL) framework and proposes a technique that accumulates grammar documentation, sample code, and parser error messages for the new language in a feedback loop, then uses the LLM to compress this information. Specifically, the correction history is summarized from the perspective of "why an error occurred and how it was corrected," and then provided to the model as a concise prompt (an ICL prompt). Because this compressed prompt consolidates key points about the new language, it helps guide the model toward proper completion from the first inference, potentially improving accuracy without repeatedly relying on a multi-iteration feedback loop.

Experiments conducted on several unknown languages and DSLs demonstrate that incorporating a compressed ICL prompt of the error-correction history improves accuracy metrics such as grammar correctness and BLEU score to levels comparable to those achieved by the feedback loop method. Furthermore, this approach significantly reduces both token count and inference time compared to using a feedback loop.

# 目次

# 第 1 章

# 新言語のための LLM コード補完に向けた手法

ソフトウェア開発の規模拡大と多様化が進む現代において、プログラミング言語に対する要求はますます複雑になっている。既存のメジャー言語では対処しづらい特殊な機能や表現力を必要とする要求に応えるために、ドメイン固有言語 (Domain-Specific Language: DSL) を開発し、利用することは一般的な手法となっている。

独自言語を設計・実装し、開発者コミュニティに普及させるためには、言語仕様の策定やコンパイラの開発、ドキュメント整備など多岐にわたる作業が必要となる。IDE（統合開発環境）での支援機能やコード補完の充実は言語利用者にとって欠かせない要素であるにもかかわらず、その開発には多くの労力と時間を要する。こうした状況を背景に、言語開発における労力を削減する手段として、Language Workbench のような統合ツール群が注目を集めてきた。Language Workbench は、新言語の文法や意味論を定義するメタモデルからエディタやパーサ、コンパイラなどを自動生成・半自動生成する枠組みを提供し、従来よりも迅速に DSL を開発することができる。

深層学習や大規模言語モデル（LLM）の急速な進歩により、自然言語とプログラミング言語の両面で高度な自動補完やコード生成が実用的になりつつある。代表的な例としては、GitHub Copilot に代表されるような IDE 統合型のコード補完システムが挙げられ、すでに Python や Java といったメジャー言語では開発効率を大きく向上させる成果を上げている。

既存の Language Workbench は LLM を新言語に対応させ、IDE でのコード補完を提供するための機能を備えていない。多くの LLM は広く使われる言語に対する学習データを大量に含む一方、利用者数が極端に少ない新言語や DSL のような「Low-Resource 言語」については十分なトレーニングがなされていない。LLM のコード補完機能をそのまま適用しても、文法エラーや不適切な補完が頻発し、必ずしも開発者にとって有用ではないことが問題となる。

LLM を特定タスクに応用するアプローチとしては、追加のファインチューニングを行う方法が挙げられる。いずれの手法においても新言語向けに十分なコードサンプルを集める必要があり、研究初期の段階やコミュニティがまだ小さい言語ではデータ不足が深刻化する。ファインチューニング自体が計算資源とコストを要する点も導入の障壁となっている。IDE での実

用的なコード補完機能を実現するには、ただ構文ルールを学習するだけでなく、実際の文法エラーや修正事例といった知識も取り込む必要がある。

本研究では In-Context Learning（ICL）の枠組みを新言語のコード補完に適用し、さらに多回に及ぶフィードバックループで得られるエラー修正の履歴を圧縮する「Compressed Prompt Generation」手法を提案する。

従来の Fine-tuning やプロンプト内での複数回のフィードバックループを用いて補完精度を高める ICL を用いた方法が試みられているものの、それぞれに次の問題がある。Fine-Tuning は新言語のように学習データが不足している場合や、大規模な計算資源が得られない場合、適応が難しく、フィードバックループを多数回行うと、推論履歴が蓄積しトークン数が加速度的に増大し、IDE 上で使うには現実的でない推論時間や API コストをもたらす。Compressed Prompt Generation ではフィードバック履歴を要約・統合して短いプロンプトに落とし込み、新言語の要点やエラー回避パターンを簡潔に提示することで、コストを抑えながらコード補完の精度向上を目指す。

本研究の狙いは、大規模言語モデル (LLM) が学習データを十分に含んでいない新規プログラミング言語やドメイン固有言語 (DSL) に対しても、実用的なコード補完を提供する仕組みを構築する点である。本手法では次の手順で補完を行う。

1. 言語作成者が、新言語ドキュメントやコード例、パーサを組み合わせ、エラー修正のフィードバックループを回して修正履歴を蓄積する。
2. 集めた事例やエラーメッセージを LLM に要約させ、ICL (In-Context Learning) プロンプトとして圧縮する。
3. 生成した ICL プロンプトを IDE に組み込み、初回推論でも低いコストで高精度のコード補完を実現する。

本論文は、未知言語に対するコード補完に着目し、次の点に主眼を置いている。

1. 従来のフィードバックループ手法が持つ高精度化の利点を活かしつつ、プロンプトの肥大化を回避する方法として、フィードバック履歴の圧縮がどの程度有効かを定量的に検証する。
2. 圧縮後のプロンプト（ICL プロンプト）を使用してもなお修正が必要なケースに対して、少数回の追加フィードバックを組み合わせることでどれほど高い精度を得られるかを評価する。
3. 複数の DSL や新言語を対象に実験を行い、提案手法が一般的な Low-Resource 言語への適用事例として有効性を持つかを示す。

実験では、自然言語ドキュメントのみを提示したベースラインは文法正解率や BLEU スコアが低めとなり、置換トークン数が増えるにつれ精度も低下する結果になった。一方で、フィードバックループにより精度が著しく向上し、その履歴を圧縮した ICL プロンプトを用いることで、最初の推論時から低コストかつ高い精度を確保しやすい事実が確認された。さらに ICL プロンプトにフィードバックを追加する手法では、若干の推論コスト増を伴いながら

も補完精度をさらに高められることがわかった。

# 第 2 章

# プログラミング言語開発の背景と課題

## 2.1　新しいプログラミング言語を取り巻く課題

### 2.1.1　新言語設計の背景

プログラミング言語の世界では、特定の領域に最適化した新しい言語を設計・実装する試みが数多く行われている [1]。並列分散処理を専門に扱う研究者が、高度な並列制御文法を持つ新しい言語を設計 [2][3] したり、IoT 機器向けの組み込みシステムといった特定の用途 [4] や、ヒューリスティック計算などの特定の計算に特化した言語 [5] を開発するなどの事例が挙げられる。このように、既存言語では表現や最適化が難しい問題領域を解決するために独自の言語をゼロから開発するケースは少なくない。

これらの新言語の背後には、既存言語や既存ツールチェーンでは十分に対処しきれない要求を満たす必要があるという動機がある。一方で、新しいプログラミング言語を一から作り上げるには膨大な労力を要するのも事実である。言語仕様の策定はもちろん、コンパイラやインタプリタの開発、ランタイムの整備、標準ライブラリの設計など、多岐にわたる技術的課題を乗り越えなければならない。さらに、言語のドキュメントやチュートリアルの整備、開発者コミュニティの構築といった周辺活動も欠かせず、これらを総合的に実施しなければ新しい言語が実用的な水準に到達してユーザを獲得することは難しい。

### 2.1.2　新言語に不可欠なツールチェーン

新言語を実用化し、開発者コミュニティに広げるためには、単に文法や意味論の仕様を定義し、コンパイラやインタプリタを提供するだけでは十分ではない。多くのプログラマーはコードを記述する際、統合開発環境 (IDE) のコード補完、シンタックスハイライト、エラーチェック機能、リファクタリングなどを活用するからである。近年では大規模プロジェクトが増えて

```
stencil upsample_x_even(vector#2 float X) {
    return (3*X + X@[-1,0]) / 4;
}
stencil upsample_x_odd(vector#2 float X){
    return (X@[1,0] + X*3) / 4;
}

out = (@[(0,2),0] = upsample_x_even(in);
       @[(1,2),0] = upsample_x_odd(in); );
```

Forma言語

```
hs = selection simple_random
ac = acceptance nonimproving-with_ probability 0.5

initialize solution

loop 100% {
    h1 = next hs
    s = first solution
    n1 = apply hi s
    h2 = next hs
    n2 = apply h2 n1
    check ac n2 replace s
}
```

HH-DSL

```
composition Capture
  device A = "Alice s Raspberry Pi"
  device B = "Bob s Raspberry Pi"
  device C = "Charlie s laptop"
  service motion = "motion sensor" on A
  service cam = "camera" on B
  service storage = "storage" on C
  synthesized service capture
    out notif photo(Image)
  when notif move() from motion do
    send req take_photo() to cam
    receive resp photo(var img) from cam
    send cmd store(img) to storage
    send notif photo(img) from capture
```

ComPOS言語

**図 2.1.** DSL の例: Forma 言語 [2]、ComPOS 言語 [4]、HH-DSL の一部構文例 [5]

おり、ソースコードを一元管理して効率的に編集・リファクタリングする仕組みが一段と重要になっている。

新言語を一般的な IDE で扱うためには、言語仕様を IDE に認識させるプラグインや拡張モジュールの開発が欠かせない。IDE が内部で用いる抽象構文木 (AST) やシンボルテーブルなどの分析情報を正しく生成・管理する仕組みを整える必要があるからである。言語仕様が複雑化するとプラグインは大型化し、実装・保守の負担が大きくなる。また、型推論機能のある言語や高度な型システムを備えた言語では、補完やエラーチェック機能に同様のロジックを盛り込み、開発コストがさらに増す。特に DSL では汎用言語とは異なる文法や意味論を持つことも多く、既存の IDE やツールチェーンをそのまま流用するのが難しい場合もある。図 2.1 に示すような DSL の構文例では、既存の文法解析手法を当てはめにくいケースが見られる。

このような IDE 機能を十分に整備しないまま新言語を公開してしまうと、利用者は不便を感じやすく、言語の普及や開発者コミュニティの形成が滞る要因となりかねない。

## 2.2 Language Workbench の概要

### 2.2.1 Language Workbench の概念

Martin Fowler によって提唱された“Language Workbench”は、プログラミング言語の開発・利用に必要なエディタやパーサ、コンパイラなどのツールチェーンを統合的に生成・管理する仕組みを提供する枠組みである [6]。Language Workbench を利用すると、新言語固有の

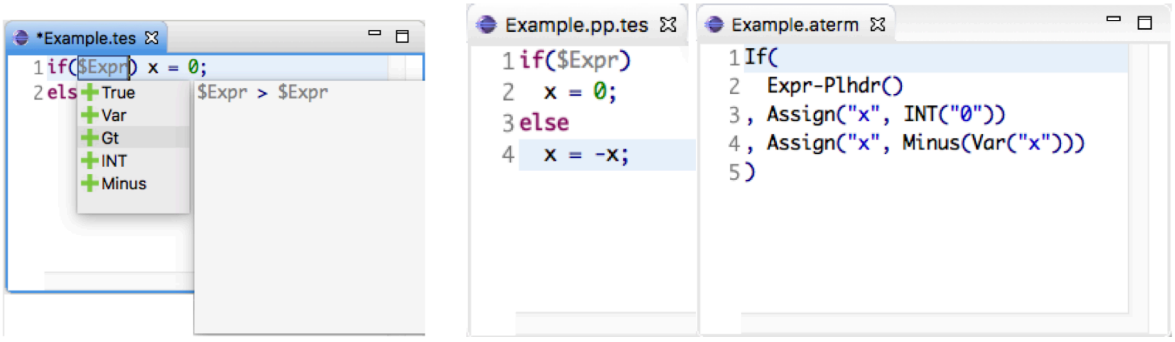



**図** 2.2. Spoofax でのコード補完、コード変換機能の実行例 ([8] のサイト上画像から引用)

補完やハイライトなどのエディタ支援機能、パーサ・AST 生成器、意味解析器、コンパイラやインタプリタといった複数のツールをまとめて整備しやすくなり、手作業でプラグインを構築する場合と比べて新言語の開発負担を大幅に削減できる可能性がある。図 2.2 は典型的な Language Workbench の 1 つである Spoofax [7] を用いたコード補完やコード変換機能の実行例である。

典型的な Language Workbench では、言語仕様をメタモデルやメタ言語と呼ばれる形式で定義し、その仕様に従って自動生成・半自動生成される形で各ツールが構築される。言語仕様を更新すれば、エディタ機能やコンパイラに即座に反映されるような仕組みが用意されており、新言語の改良や試行錯誤を繰り返しやすい点が特徴である。

### 2.2.2　Language Workbench の具体例

Language Workbench の代表例としては、Spoofax [7] や Rascal [9] などが知られている。それぞれの特徴や適用事例を概説する。

- **Spoofax [7]:** Spoofax は Eclipse プラットフォーム上で動作する Language Workbench であり、主にドメイン固有言語 (DSL) の開発を強力に支援する。SDF(Scanning and Parsing Definition Formalism) による文法定義と、Stratego と呼ばれる変換言語を用いた意味解析やコード変換の仕組みを提供している。図 2.3 に、Spoofax での文法定義、セマンティックス定義、コード変換機能の例を示す。構文や型システムを変更した際、コード補完やエラーチェックなどのエディタ機能も即座に更新できる点が特徴で、設計段階での試行錯誤を円滑に行いながら常に最新の開発サポート環境をユーザに提供可能である。
- **Rascal [9]:** Rascal はソースコード解析と変換を軸としたメタプログラミング言語であり、SDF を活用して Python や Java などの既存言語に対するコード生成や変換機

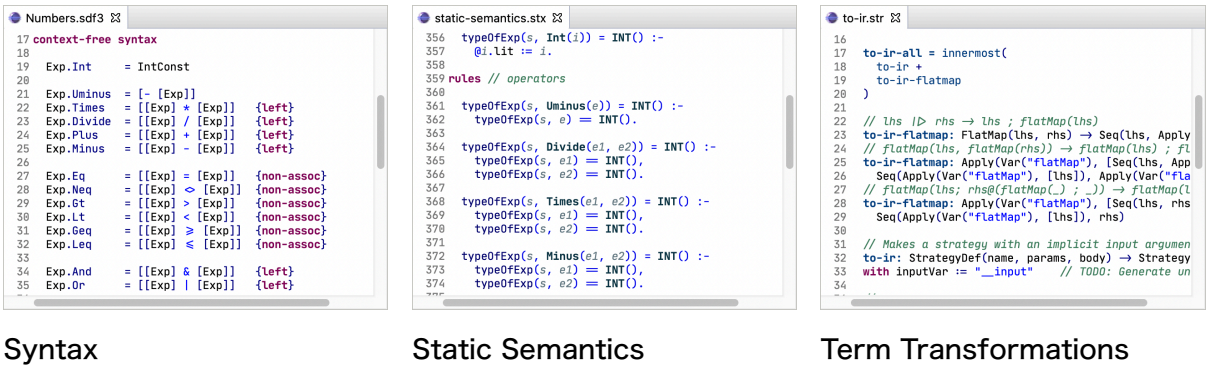



**図 2.3.** Spoofax での文法定義・セマンティックス定義・コード変換機能の定義例 ([8] のサイト上画像から引用)

能を実現している。リファクタリングなど多様なメタプログラミングを行いやすく、Rascal 自体も Eclipse ベースの Language Workbench として利用可能である。

### 2.2.3 Language Workbench の限界と新たな要求

Language Workbench を導入することで、プラグインを手作りする場合よりも言語実装者の負担は大幅に軽くなる。しかし、近年のソフトウェア開発では、IDE の支援機能として一段上のコード補完や自動生成を求める声が強まっている。具体的には、ソースコードを構文・型レベルで補完するだけでなく、設計意図や大規模コードベースの知見を踏まえた自動生成が期待されている。

## 2.3 IDE と LLM の統合が進む現状

### 2.3.1 大規模言語モデルによるコード補完

従来の IDE では、ユーザが数文字入力すると、言語の文法規則や静的解析結果をもとに、適切なキーワードや識別子を提案するコード補完機能が広く用いられてきた。深層学習技術の飛躍的な進歩と大規模データの活用によって、自然言語やプログラミング言語を扱う大規模言語モデル (LLM) が急速に発展している。2022 年に公開された Copilot for VSCode[10] や、2023 年に登場した Cursor[11]、2024 年に発表された Apple Intelligence for Xcode[12] などは、LLM を本格的に IDE に統合する事例として注目を集めた。

LLM を活用した補完機能では、大量の公開リポジトリ上のコードを学習したモデルが自然言語としてのコンテキストをある程度理解するため、より高レベルな提案が実現される [13]。単なるキーワードの補完にとどまらず、関数の名称や型情報から関数本体を自動生成する、テ

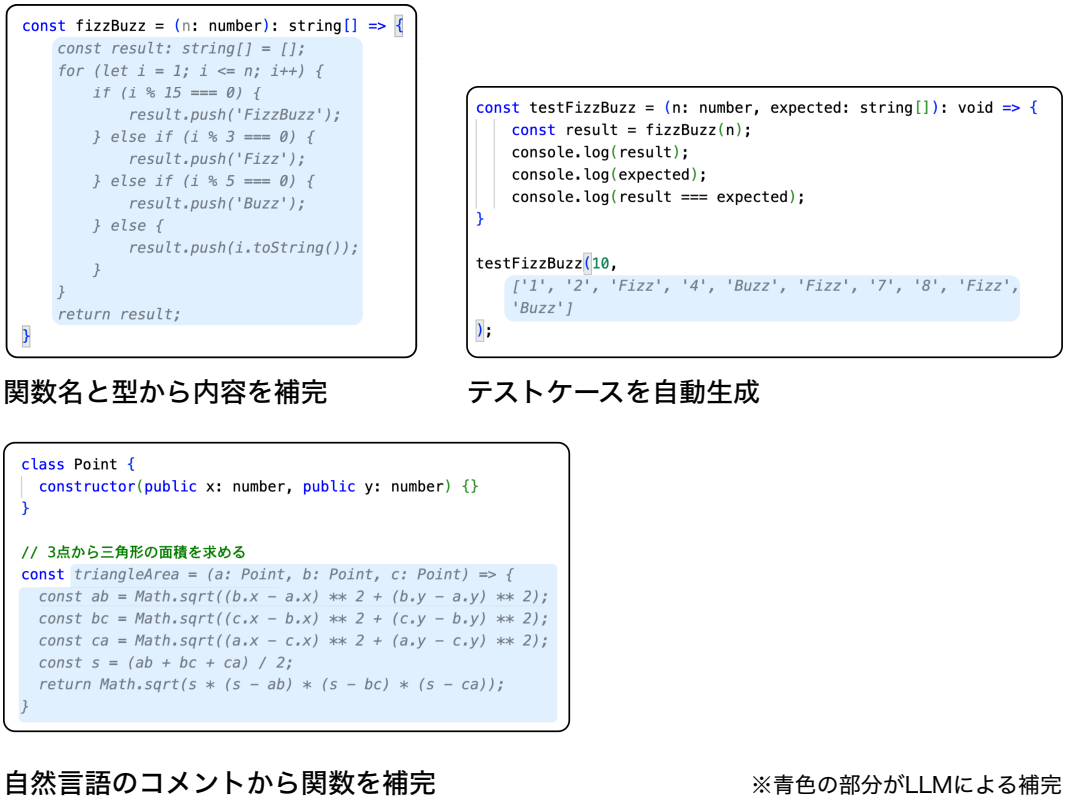



**図 2.4.** 大規模言語モデルによるコード補完、コード生成の例 [10]

ストケースを生成するといった高度なコード補完が既に実用段階に入りつつある [13]。自然言語形式のコメントを入力し、モデルが対応するコード断片を生成する手法も広く普及し始めている。図 2.4 に、Copilot for VSCode におけるコード補完やコード生成の例を示す。

### 2.3.2　LLM 活用による利点と限界

LLM による IDE 機能は、開発者の生産性向上につながると報告されている [13]。ライブラリやフレームワークの探索が効率化され、モデルが蓄積した実装パターンをもとに提案が行われるためである。自然言語のコメントや質問を与えると、雑多なタスクの自動化やサンプルコードの生成を行いやすくなる点も開発効率を底上げする要因となっている。

一方で、学習データセットに含まれていない言語や、使用頻度が低い言語、あるいは独自開発の新言語などについては、十分な恩恵を受けにくい。言語仕様や文法規則、慣用表現をモデルが獲得するにはある程度以上の規模のサンプルコードが必要となるため、データセットの少ない新規に作られた言語や、特定領域専用の DSL などに対しては生成品質の精度が低下しがちである。このような Python や Java などの高リソース言語に比べて、対応データセットが

少ない言語を「Low-resource 言語」と呼ぶ [14]。Low-resource 言語における LLM 活用には、以下のような課題あげられる [14]。

1. **データ不足:** Rust や R のようにユーザが増えつつある言語でさえも、大規模言語モデルの学習で採用されるデータセットに含まれる量は、Python や Java などの高リソース言語に比べてきわめて少ない。ましてや特定領域専用の DSL となると、公開されたコード例が存在しない場合も珍しくない。
2. **ドメイン特化言語への対応:** DSL は領域固有の文法構造や厳密な仕様をもつ場合が多く、一般的なプログラミング言語とは大きく異なる規則を持ちうる。モデルの既存知識だけでは対応しきれないケースが多々あり、高精度な補完には追加の工夫が必要となる。
3. **評価手法の不足:** Python などには多彩な公開ベンチマークがあり、モデルを多方面から検証できる。一方、Low-resource 言語には利用可能なベンチマークが少なく、文法も用途も言語ごとに異なるため、複数言語を横断的に比較する共通基盤が不足している。
4. **データのコンタミネーション:** コミュニティや企業内に閉じたコードしか存在しない言語だと、学習用データと評価用データが重複しやすくなり、正確な性能評価を阻害する。

このように、Low-resource 言語ではデータセットの多い言語に比べ、LLM の活用を阻む要素が多い。新しい言語でコード補完やコード生成を行うためには、その言語仕様を熟知した LLM が必要であるものの、LLM を最初から作成するのは困難を伴う。GPT 系モデルなどの LLM は数百億から数千億にも及ぶパラメータを持ち、膨大なテキスト・コードデータを用いて事前学習されており [15]、GPT-3 では 1750 億パラメータ・570GB のコーパスが用いられたことが報告されている [16]。

個人や小規模研究グループがこれほどの大規模学習を行うのは、計算資源や人材面で非現実的である。また、新言語の普及初期では、学習に十分な量のコード例やプロジェクトがまだ存在しないため、データ不足という本質的な問題も生じてしまう。

## 2.4 Low-resource 言語への LLM 適用の具体的手法

Low-resource な言語へ LLM を適用する試みは、近年多くの研究で進められている。研究動向を大きく分類すると、以下のとおりである [14]。

1. **Fine tuning:** 既存の大規模言語モデルの重みを Low-resource 言語用に微調整し、文法や慣用表現を学習させる Fine tuning と呼ばれるアプローチである。ベースモデルに比べ、対象言語に対する性能が大幅に向上する場合が多い [17, 18, 19, 20, 21, 22, 23]。
2. **In-Context Learning と反復学習:** Zero-shot や Few-shot といった従来の手法に加え、Chain-of-Thought や Context 内のデータを活用してモデルの思考プロセスや役割・文脈を指定する In-Context Learning(ICL) という方法が提案されている [24, 25]。

コンパイルエラーやテスト結果をフィードバックし、複数回の再生成を行う「反復的デバッグ」は有効であり、Verilog や Rust など構文エラーを起こしやすい言語の精度改善に利用されている [21, 26, 27]。

3. **Retrieval-Augmented Generation（RAG）:** DSL のようにコード例やドキュメントが限られた言語では、モデル内部のパラメータだけでなく、外部のソースコードやマニュアルを検索・参照して活用する手法が試みられている。必要とする文法要素をドキュメントから抽出し、モデルに提示することで、より適切なコード生成を行おうとするアプローチである [28, 29, 30]。
4. **他の手法:** 言語に特化した比較損失を導入して小規模モデルを学習する [31]、生成時のデコード手順に文法制約を組み込み生成結果を制御する [32, 33]、LLM が扱いやすい小さな DSL を複数作って組み合わせる [34, 35] など、多様なアプローチも検討されている。

### 2.4.1　Low-resource 言語の LLM 対応 - Fine-tuning

Low-resource 言語への LLM 対応の主要な方法の 1 つとしてファインチューニング (Fine-tuning) と呼ばれる事前学習済み LLM に追加の学習を行い、特定のタスクに最適化する方法がある [36, 37]。Fine-tuning は LLM を一から作成する場合と比べて、小さいデータセットと少ない計算資源で特定のタスクへの対応を行うことができるが、それでも新言語開発者が用意するには大きすぎるデータセットと計算資源を要求するという問題がある。

事前学習済みモデルに対して行うファインチューニングには、次のようにいくつかの形態が存在する。

1. **Full Fine-Tuning:** モデル全パラメータを再学習可能とする方法である。数十億から数千億にも及ぶパラメータをすべて微調整する必要があるため、大規模な計算資源とデータが欠かせない。一方で、タスクに合わせてモデルを最大限に適応させることができる [37, 38]。
2. **Partial Fine-Tuning:** モデルの特定層のみを学習対象とする方法であり、残りの層は固定したまま微調整を行う。Full Fine-Tuning に比べて計算負荷は抑えられるが、依然としてそれなりのデータセットと計算資源が必要になる [39]。
3. **Parameter-Efficient Fine-Tuning (PEFT):** Adapter [40] や LoRA(Low-Rank Adaptation) [41] などの軽量学習技術を活用し、モデル全体のパラメータを固定して一部に挿入した小規模な学習可能パラメータ（小さな MLP や低ランク行列など）だけを更新する方式である。学習コストを大幅に抑制できるが、元のモデルが対象言語を十分にカバーしていない場合は精度向上が限定的にとどまることもある [42]。

#### Full/Partial Fine-Tuning の実用例

Full Fine-Tuning や Partial Fine-Tuning をプログラミング言語タスクに応用する研究としては、Code Llama[43] や Carllm[44] が挙げられる。これらは、人気のある言語や用途向けの大規模ソースコードを追加学習することで、実用的なコード生成性能を実現している。

- **Code Llama:** Llama 2 モデルをベースに、コード生成・理解に特化した Full Fine-Tuning を施した LLM である。Python 向けの Code Llama - Python や、ユーザの指示に応じやすい Code Llama - Instruct など多様なバリエーションが用意されており、HumanEval や MBPP などのベンチマークでも高いスコアを示している。
- **Carllm:** GitHub 上の膨大なコードレビュー履歴を学習し、レビューコメントを生成する機能を強化した LLM である。多数のオープンソースプロジェクトからレビューコメントと修正履歴を収集し、Partial Fine-Tuning を行うことでコードの欠陥指摘や改善案の提案精度を高めている。

#### Full/Partial Fine-Tuning の新言語への適用が難しい理由

上記研究はもともと利用者が多く、豊富なコードベースが存在する言語を対象にしたものだからこそ、高い精度が得られている点に留意する必要がある。たとえば Code Llama - Python は 1000 億トークン (100B token) 以上の Python コードを追加学習しており、Carllm も GitHub から 1,793 件のプロジェクトと 19,456 件のファイルを収集するといった、大規模なデータセットを構築している。Rust や R のようにユーザコミュニティが拡大しつつある言語であれば、同規模のデータを集めることは可能かもしれない。しかし誕生したばかりの新言語では、これほど膨大なコードを準備できない場合が多く、Full Fine-Tuning や Partial Fine-Tuning は現実的ではないと考えられる。

### 2.4.2 Parameter-Efficient Fine-Tuning(PEFT)

Full Fine-Tuning や Partial Fine-Tuning よりも計算負荷を軽減しやすい代替策として、Parameter-Efficient Fine-Tuning (PEFT) が注目されている [42, 22, 23]。PEFT では、モデル本体の重みを固定し、アダプタ層や低ランク行列などの部分的なパラメータのみを学習可能とする。Adapter や LoRA が代表例である。

- **Adapter:** Transformer ブロック間に小規模な MLP などを挿入し、そこだけ学習可能にする技術である。メモリや計算量を大幅に節約できる一方で、追加された Adapter 層を使って対象言語固有の知識を補う。
- **LoRA:** モデル内部の重み行列の一部を低ランク行列分解した形で再パラメータ化し、その分解先の行列を学習する方式である。Adapter よりもさらに少ない追加パラメー

タで済む場合が多く、大規模モデルであっても比較的少ない計算コストでタスクや言語への特化を図りやすい [45]。

#### PEFT の限界

PEFT 手法によって、比較的少量のデータセットでもある程度の精度向上が見込める。しかし、ほとんどコードサンプルが存在しない新言語では、ベースモデルがそもそも該当言語の知識を持たないという根本的な問題が解決しにくい。PEFT は既存のモデルが持つ知識を活かして微調整するための手法であり、完全に未知の文法や構文要素を学習させるには十分なサンプルコードが不可欠である。

さらに、PEFT の実装においては数千から数万規模のサンプルが必要になることも報告されている。Adapter のトレーニングを行う先行研究では、数千件のサンプルを用いる例が少なくない。LoRA [41] では LLaMA-13B や BLOOMz-7B、GPT-J-6B に対し、

- GSM8K: トレーニング 8,800 件、テスト 1,319 件
- AQuA: トレーニング 100,000 件、テスト 254 件
- Math10K: トレーニング 10,000 件

といった大規模なセットを用いている。LLM-Adapters [40] では GPT-3 175B を

- MNLI: 433,000 文
- WikiSQ: 87,726 の SQL クエリ

などで学習している。これらの数字は、Full Fine-Tuning と比較すれば小規模かもしれないが、新言語では文字通り「1 からコードを書く」必要があるため、言語開発者にとっては依然として高いハードルである。

### 2.4.3　Low-resource 言語の LLM 対応 - In-Context Learning

大量のデータを用いて、大量のパラメータを更新する必要がある Fine-Tuning とは別のアプローチとして、プロンプトの設計や反復学習を通じて精度を高める In-Context Learning(ICL) が知られている [46]。ICL では、LLM そのもののパラメータを更新せず、推論時のプロンプトに具体的な例や文脈情報を付与することで、その場でタスクに合わせた出力を導く。ICL は新言語のように非常にデータセットが少ない状況で有用であるが、プロンプトが長くなることで推論時間や API コストが増大するといった問題がある。ICL は以下のように区分される [46]。

1. **Few-shot In-Context Learning:** 数個の「入力→出力」例（例：「hello → bonjour」「thank you → merci」など）を提示して入出力関係を示し、未知の入力に対応した出力を推定させる方式である。従来は Few-Shot[16] と呼ばれていたが、ICL の一部と位

置づけられる。

2. **Instruction / Role-based ICL:** 指示文や役割指定による ICL であり、「英語をフランス語に翻訳してください」「フランス語翻訳の専門家として回答してください」などと記述するだけで、モデルの振る舞いを切り替える方法を指す。従来の few-shot のように具体例を提示しなくても、「あなたは○○の専門家です」という役割付与だけでタスクに対応できるケースもある。
3. **Unsupervised / Explanation-based ICL:** ラベルなしの事例や説明文から推論する ICL である。入力と正解ラベルが対になっていない状況や、類似問題の解答例のみが存在するケースでも、タスクや文脈を推定して精度を向上させることが可能であると報告されている [**?**]。
4. **Time-series Extrapolation ICL:** 時系列データをもとに外挿を行う ICL である。売上や株価といった時系列情報を文脈として与え、先の値を予測する場合などが典型例で、データの傾向を文脈から推測する仕組みを ICL とみなせる。
5. **Meta-ICL:** 階層構造を持つタスクを段階的に学習する ICL である。文脈内で複数のタスクを提示し、「タスク A → タスク B → タスク C……」と進むうちに、後段のタスクほど少ない例示で素早く適応できる。問題→誤り指摘→正解→次の問題、というプロセスで精度を高めることが報告されている。
6. **Basic ICL in NLP:** 自然言語処理の基本タスクを ICL として捉える考え方である。代名詞の照応先や文構造、単語の語義などを文脈情報から類推する場合、ICL が持つ仕組みと類似した推論が行われるとされる。

### In-Context Learning の実用例

ICL を活用した研究としては、Many-Shot In-Context Learning(MSICL)[47] や UniLog[48] などが代表的である。

- **Many-Shot In-Context Learning:** MSICL では、Unsupervised ICL を用いて解説や解答そのものを提示しなくとも、問題だけを多数例示することでモデルがタスクを理解できるかを検証している。タスク関連の知識を事前学習がすでに内在化している場合、few-shot を超える性能向上が得られることが示唆されている。many-shot で大量にサンプルを提示することで事前学習で形成されたバイアスを上書きできる可能性があり、従来は Fine-tuning でしか達成できなかったレベルの学習性能が many-shot ICL だけで得られる場合もあると報告されている。
- **UniLog:** UniLog では、プログラムにログ出力コードを挿入する位置やログレベルの判断、ログメッセージの生成などを LLM に行わせる際、数百件のサンプルをプロンプトとして与える形で ICL を実装している。これにより、モデル側のパラメータは更新せずに、推論時のプロンプト設計だけでタスクの精度向上を狙う方法を示している。

ICL は、大規模な追加学習を伴わずに LLM 内部の言語知識を再利用できる点が利点として挙げられる。大規模なデータセットや計算資源を用意しにくい新言語にとっては、コード補完やコード生成を LLM で行う際の有力な手法となり得る。

#### Low-Resource 言語での ICL 応用の問題

比較的ニッチな言語や用途でも LLM を活用し、出力結果を人間やツールで検証したうえでモデルにフィードバックすることによって ICL の精度を段階的に高める研究も存在する [21][26][27]。こうしたフィードバックループはコード精度を上げる効果がある一方で、オーバーヘッドが増すことが指摘されている [14]。

Low-resource 言語向けの LLM 活用では、生成したコードをツールや開発者が検証して再度修正を行う「フィードバックループ」を繰り返す方法が広く試みられている [21][26][27]。このアプローチは、モデル出力の精度を少しずつ高める利点がある一方で、以下のようなオーバーヘッドが発生しやすい点が問題視されている。

- **プロンプトの肥大化:** フィードバック履歴を蓄積してプロンプトに反映するたびに、入力プロンプトが膨らみやすい。必要トークン数が増すと、推論時間と API コストが加速度的に増大する。
- **リアルタイム性の低下:** 長大なプロンプトはモデル側の推論時間を押し上げ、ネットワーク越しの API であれば通信待ちも増す。インタラクティブにコードを編集する IDE 環境で、こうした遅延はユーザ体験を著しく損なう要因となる。
- **継続的な API コストの増大:** トークンに応じて課金される API を利用している場合、フィードバック回数に比例して料金が跳ね上がる。頻繁なコード補完やデバッグが求められる現場ほど、このコストは無視できない水準に達する可能性がある。

新言語にフィードバックループを導入するには、IDE との統合が前提となるが、オーバーヘッドが大きくなるリスクが高い。新言語ではコード自体が少なく、モデルが文法を誤解する頻度も高まりやすいため、修正作業が繰り返されてプロンプトや問い合わせ数が膨らむ恐れがある。開発サイクルの効率をかえって悪化させ、API コストを高騰させる可能性があり、実運用に向けた解決策が必要である。

### 2.4.4　Low-resource 言語の LLM 対応 - RAG や他の手法

Low-Resource な言語への LLM 応用に向けた大きなデータセットを必要としない他のアプローチとして、LLM へ渡すプロンプトに必要なデータを検索して追加する RAG[28, 29, 30] やデコーダー層で LLM の出力を制限する [32, 33] などがある。これらのアプローチも有用ではあるが、RAG は検索に用いることができる言語の Document やソースコードを必要とし、デコーダーで LLM の出力を制限すると推論精度が大幅に落ちることが報告されている [49]。

## 2.5 新言語への LLM 応用への問題

近年に新しいプログラミング言語を開発しようとした場合、LLM 補完への対応は大きな課題となっている。新しいプログラミング言語は、その言語仕様からコンパイラ、インタプリタ、ランタイム、IDE 連携など幅広い要素が必要となり、すべてを一から整備するには莫大なコストがかかる。Language Workbench と呼ばれる支援環境が存在するものの、近年の IDE で一般的となっている LLM を用いた高度なコード補完や自動生成機能には対応できていない。

LLM を新言語に対応させるアプローチの 1 つとして、Full Fine-Tuning や Partial Fine-Tuning、PEFT といったファインチューニング手法があるが、新言語のように学習すべきサンプルコードが極端に少ない場合には応用が難しい。In-Context Learning(ICL) という追加学習不要のアプローチもあるが、Low-Resource での ICL で主流である文法誤解やエラーの修正を繰り返すフィードバックループという方式には、大きなオーバーヘッドが発生するといった問題がある。

# 第 3 章

# Compressed Prompt Generation

## 3.1 提案の目的

前章までの課題を解消するため、Language Workbench のコンポーネントとして新言語用のコード補完タスク向けのプロンプトを生成する「Compressed Prompt Generation」というシステムを提案する。Compressed Prompt Generation はフィードバック付き In-Context Learning と同等の精度を持ちながら、フィードバックがない In-Context Learning と同等の速度で補完を行うことを目指す。本手法ではフィードバック付き In-Context Learning で得られたプロンプトを LLM で圧縮してプロンプト（ICL プロンプト）を生成する。補完時は生成した ICL プロンプトとユーザーのコードを LLM に与え、フィードバックを行わずに補完を行う。

本手法では次の 3 種類の入力から ICL プロンプトを生成する。いずれも開発者に大きな追加負担を強いることなく用意できる点が利点である。

1. **新言語の自然言語ドキュメント:** 言語仕様書やリファレンスガイドなどを自然言語形式で LLM に与える。
2. **新言語の具体的コード例:** 各言語 10〜15 例程度用意する。
3. **新言語を解析できるパーサ:** Language Workbench の別コンポーネントで文法規則より生成する。

これら 3 つの入力から、要点だけを抽出した「ICL プロンプト」を生成することで、長大なプロンプトを常時保持しなくても新言語向けの高精度な補完を行うことが期待できる。前処理ではフィードバックループを通して補完エラーや修正内容の履歴を蓄積・抽出して圧縮し、ICL プロンプトとして再構成する。圧縮によって得られたプロンプトには、エラーや修正の事例から抽出された主要情報のみが含まれることになる。このようにフィードバックループの利点を活かしながら、プロンプトの肥大化を軽減できる可能性がある。

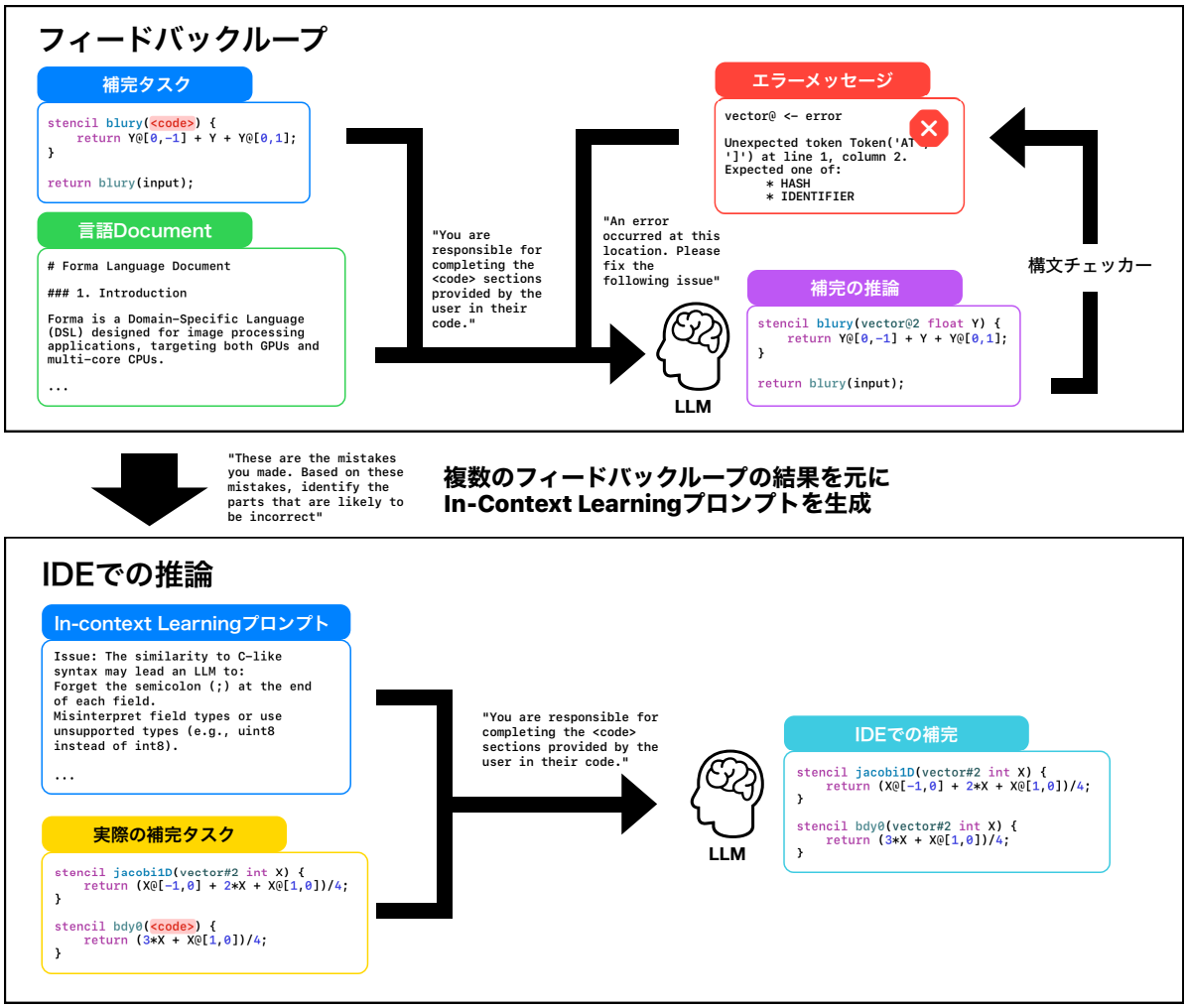



図 3.1. アルゴリズムの全体構造（Compressed Prompt Generation）

## 3.2 提案システムのアルゴリズム

図 3.1 は提案手法の全体的なアルゴリズムを示している。流れを要約すると以下のとおりである。

1. **コード補完問題の生成:** 新言語で書かれたコード例を部分的に置換文字列に変換し、ユーザーの IDE での補完を模した補完問題を作成する。
2. **LLM への解答要求とエラー検出:** 新言語ドキュメントと補完問題を組み合わせ、システムプロンプトとともに LLM に入力してコードを出力させる。生成されたコードをパーサで検証し、エラーがあればそのメッセージを再度 LLM へ渡して修正を要求する。この作業をエラーが解消されるまで繰り返す。

3. **学習結果プロンプト断片の作成:** 個々の補完問題に対して、誤答やエラーメッセージ、最終的な正解コードなどをセットで保存する。この段階で「なぜエラーが起き、どう修正したか」という情報が蓄積される。
4. **全問題に対する圧縮:** 得られた履歴のすべてと新言語ドキュメントをまとめ、LLM に対して「圧縮指示」を与える。モデルは重複や不要部分を削ぎ落としてエラー修正に必須なポイントを要約し、簡潔なテキストを生成する。これが「ICL プロンプト」に相当する。
5. **IDE での補完:** ユーザが新言語でコードを編集するとき、あらかじめ作成しておいた ICL プロンプトと補完用プロンプトを LLM に提示する。エラー修正の知見が集約されているため、長大な履歴を維持せずとも新言語向けの高精度補完が行いやすい。

### アルゴリズムの詳細

提案手法のアルゴリズムは次のようなステップで構成される。補完問題を $x^i$ $(i = 1, 2, \ldots, k)$ とし、LLM への問い合わせを $\mathcal{M}(x)$、パーサによる検証を $\mathcal{P}(y)$、文字列連結を $||$ で表記する。

1. **初回のコード生成:** システムプロンプト $p_{\text{sys}}$、新言語ドキュメント $p_{\text{doc}}$、補完問題 $x^i$ を連結してモデルに入力し、補完コード $y_0^i$ を生成する。

$$y_0^i = \mathcal{M}(p_{\text{sys}} \,||\, p_{\text{doc}} \,||\, x^i)$$

2. **パーサによるエラー検出:** $y_0^i$ をパーサ $\mathcal{P}$ で検証し、エラーメッセージ $e_0^i$ を得る。

$$e_0^i = \mathcal{P}(y_0^i)$$

3. **エラー修正要求:** エラーがある場合は、修正指示 $p_{\text{fix}}$ とエラーメッセージ $e_n^i$ を連結し LLM に再入力して修正版コードを生成する。エラーが解消しない場合は繰り返し実行し、パーサを再度通して検証を行う。

$$y_{n+1}^i = \mathcal{M}(\ldots \,||\, p_{\text{fix}} \,||\, e_n^i)$$

$$e_{n+1}^i = \mathcal{P}(y_{n+1}^i)$$

4. **学習結果プロンプト断片の作成:** 問題 $x^i$、誤答の履歴、エラーメッセージ列 $e_j^i$、そして合格コード $y_n^i$ を統合し、$R^i$ として保存する。
5. **圧縮:** すべての $R^i$ を連結し、新言語ドキュメント $p_{\text{doc}}$ も付加したうえで圧縮指示 $p_{\text{compress}}$ を加え、LLM へ入力する。すると、モデルは重複や不要要素を排除し、エラー修正に要するポイントだけをまとめた ICL プロンプト $p_{\text{icl}}$ を出力する。

$$p_{\text{icl}} = \mathcal{M}\big((p_{\text{doc}} \,||\, R^1 \,||\, \cdots || \, R^k) \,||\, p_{\text{compress}}\big)$$

6. **IDE での補完:** 実運用時は、生成済みの ICL プロンプト $p_{\text{icl}}$ と補完指示プロンプト $p_{\text{comp}}$、ユーザ入力のコード断片 $p_{\text{user}}$ を連結し、LLM に問い合わせて最終的な補完結果 comp を得る。

$$\text{comp} = \mathcal{M}(p_{\text{icl}} \,||\, p_{\text{comp}} \,||\, p_{\text{user}})$$

アルゴリズム 1 とアルゴリズム 2 に疑似コードを示す。

**Algorithm 1** ICL プロンプト生成アルゴリズム

**Require:** $p_{\text{sys}}$: システムプロンプト, $p_{\text{doc}}$: 新言語ドキュメント, $p_{\text{fix}}$: 修正指示, $p_{\text{compress}}$: 圧縮指示, $\{x^i\}_{i=1}^{k}$: 補完問題

**Ensure:** 圧縮済プロンプト $p_{\text{icl}}$

**for** $i = 1$ to $k$ **do**
  **(1) 初回のコード生成:**
  $y_0^i \leftarrow \mathcal{M}(p_{\text{sys}} \,||\, p_{\text{doc}} \,||\, x^i)$
  **(2) パーサによるエラー検出:**
  $e_0^i \leftarrow \mathcal{P}(y_0^i)$
  **(3) エラー修正要求:**
  $n \leftarrow 0$
  **while** エラーメッセージ $e_n^i$ が存在し、回数制限に達していない **do**
    $y_{n+1}^i \leftarrow \mathcal{M}(\ldots \,||\, p_{\text{fix}} \,||\, e_n^i)$
    $e_{n+1}^i \leftarrow \mathcal{P}(y_{n+1}^i)$
    $n \leftarrow n + 1$
  **end while**
  **(4) 学習結果プロンプト断片の作成:**
  $R^i \leftarrow (\, x^i, \{ \text{誤答}, e_0^i, e_1^i, \ldots \}, \, y_n^i \,)$
**end for**
**(5) 圧縮:**
$p_{\text{icl}} \leftarrow \mathcal{M}\big((p_{\text{doc}} \,||\, R^1 \,||\, \ldots \,||\, R^k) \,||\, p_{\text{compress}}\big)$
**return** $p_{\text{icl}}$

**Algorithm 2** ICL プロンプトを利用した IDE でのコード補完

**Require:** $p_{\text{icl}}$: 圧縮 ICL プロンプト, $p_{\text{comp}}$: 補完指示プロンプト, $p_{\text{user}}$: ユーザ入力のコード断片

**Ensure:** comp: 補完結果

comp $\leftarrow \mathcal{M}(p_{\text{icl}} \,||\, p_{\text{comp}} \,||\, p_{\text{user}})$
**return** comp

アルゴリズム 1 とアルゴリズム 2 が示すように、新言語のドキュメントや補完問題、修正履歴などを圧縮して最適化されたシステムプロンプトを構築する点が本手法の重要なポイントで

ある。

## 3.3　提案手法の利点

提案手法は以下の利点をもたらすと考えられる。

- **低オーバーヘッド:** 圧縮後の ICL プロンプトを利用するため、フィードバック履歴を逐次付加し続ける必要がなく、推論時のトークン消費や API コストを大きく抑制できる。
- **容易な導入:** 自然言語ドキュメント・少数のコード例・パーサがあれば開始できるため、新言語の開発初期でも導入しやすい。言語仕様が変更された場合でも圧縮過程を繰り返すだけで対応可能である。
- **配布のしやすさ:** 圧縮済みプロンプトを配布すれば、新言語に詳しくないユーザでも高精度補完を活用できる。大規模なモデルや膨大なプロンプト履歴を共有する必要がなく、新言語の普及を促進しやすい。

# 第 4 章

# 実験

本章では、前章で提案した Compressed Prompt Generation 手法の実効性を検証するために、複数の Low-Resource なドメイン固有言語 (DSL) を用いて実験を行う。既存の大規模言語モデル (LLM) は、Python や C/C++ などのメジャー言語については十分な学習を受けている可能性が高いが、新たに開発された DSL などについては学習データがほぼ存在しない場合が多い。

そのような未知言語に対するコード補完能力を高める方法として、本研究では In-Context Learning(ICL) を用いたフィードバックループ、そして最終的にプロンプトを圧縮する本手法 (Compressed Prompt Generation) がどの程度有効かを検証する。

本章では、実験の全体像や対象とする言語、具体的なタスク生成手順、評価する指標と実験手順のフローを示したのちに実験の結果を記述する。

## 4.1 実験の目的と概要

本研究の目標は、LLM が事前学習時に十分な量のサンプルを含んでいないであろう未知言語に対して、提案するプロンプト圧縮手法がどれほど効果的に補完タスクの精度を向上させるか、そして推論時のコストをどの程度削減できるかを評価する点である。具体的には、以下の 2 点に着目する。

1. **コード補完精度の向上:** 新言語のコードをある程度の水準まで補完可能にすることは、言語の利用者が実際に開発を進めるうえで欠かせない。未知言語の文法や仕様がメジャー言語とは大きく異なる場合、LLM が誤った推測をしやすいことが懸念される。本実験では、提案手法によるフィードバックループと ICL プロンプトの活用が、この補完の正確性と完結性をどこまで高めるかを検証する。
2. **推論時コストの削減:** フィードバックループでエラー修正を行う場合、長い履歴が蓄積されるため、その履歴を毎回 LLM に提示することになる。このとき、トークン数が大きくなり推論時間や API 利用料金が増加する問題がある。圧縮したプロンプト (Compressed Prompt) を用いることで、補完精度を保ちながらトークン消費量や推論

時間を削減できるかを確かめる。

上記 2 点は相互にトレードオフの関係にある場合も多い。すなわち、高い精度を得るためには大量の文脈情報が必要となる一方、プロンプトが肥大化すると推論コストが増大する。しかし本手法では、フィードバックループで得たエラー修正ノウハウを圧縮して再利用することで、精度とコストのバランスを改善することを狙いとしている。

## 4.2　新言語と LLM の未知性の定義

### 4.2.1　未知言語の選定基準

本研究における「未知言語」とは、LLM が事前学習データとして十分な分量を含んでいないと推測される言語であり、以下の基準によって判定した。

- **LLM への直接問い合わせでの回答精度:** 事前に「本言語の構文を教えてほしい」「この DSL で書かれた関数の例を示してほしい」などの単純な問い合わせを行い、それに対して LLM が正確かつ具体的な回答をできなかった場合を未知言語とみなす。回答が曖昧であったり、明らかに他のメジャー言語の情報を混入させたりする場合は、未知言語の要件を満たす。
- **文法や仕様に関する混同:** たとえば特定の DSL の並列制御構文を尋ねても、LLM が C 言語や Python などの制御文を出力してしまう場合が多いなど、構文的に全く異なる記述を混同し続けるかどうかを観察する。

これらの観点に加えて、内部 DSL のようにホスト言語の構文上で定義されるものは、ある程度ホスト言語の知識だけで補完が可能な場合があるこのため、本研究では外部 DSL か、内部 DSL でもホスト言語と切り離し可能であり、かつホスト言語の構文と大きく異なるものを対象とする。

これらの観点を踏まえ、未知言語と判断される DSL を合計 11 種選定した。選定された言語はいずれも研究用あるいは企業内で限定的に利用されているもので、公開されているサンプルが非常に少ないか、もしくは一般的にはほとんど知られていないものを中心とする。LLM にとっては十分な学習サンプルが存在しないと想定でき、問い合わせ時に大幅なハルシネーションや回答の拒否が生じることを確認している。

### 4.2.2　具体的な言語例

未知言語として採用した DSL を以下に列挙する。言語の説明にあたっては各言語について次の点に着目している。

- **DSL としての主要機能や構文要素:** 並列制御構文、画像処理用のステンシルオペレーション、金融計算用の特殊文法など。
- **言語のパラダイム:** 手続き型・関数型・オブジェクト指向など、いずれのパラダイムに

近いのか、あるいは複合的か。

- **従来言語との相違点やメリット:** 既存言語ではできなかったどのような処理が可能になるのか、またそのためにどのような制約があるのか。
- **LLM がハルシネーションを起こしやすいと思われる文法要素:** 多段ネスト・独自の制御フロー・記号リテラルの特殊な扱いなど、LLM が混同しやすい要素の例。

それぞれの言語の特徴は以下の通りである。

- **Forma 言語 [2]:**
  Forma 言語は、画像処理アプリケーションを高水準に記述し、GPU やマルチコア CPU など多様なアーキテクチャ向けに効率的なコードを自動生成するために作られた外部 DSL である。パラダイムは C 言語に近いな手続き型言語である。
  既存言語と比べて、実装の煩雑さを大幅に減らしつつ、ステンシルやパイプライン最適化などの高度な変換をコンパイラが一元的に行うため効率的な開発ができるというメリットがある。
  文法として `vector@[(0,2),(1,2)]` などのスケーリング付きオフセットや `funcname(args)` で呼び出す関数と `funcname<args>`で呼び出すという特殊な仕様がある。
- **ComPOS 言語 [4]:**
  ComPOS 言語は、IoT システムに含まれる複数のサービスを接続・調停し、弱い接続状況 (weak connectivity) に対処しながらシステム全体を構成するために作られた DSL である。SQL に近い宣言型言語と手続き型言語の特徴を併せ持つイベント駆動的な外部 DSL である。
  既存言語と比べて、メッセージフローや反応の流れが明示的に定義され、弱い接続の下でも新しいイベントを優先して取り扱える。古い反応を自動的に打ち切るなどの機能があるため、IoT システムに適した柔軟で効率的な開発ができるというメリットがある。
  文法として、`when notif X from S do ...` や `send req X(...) to S` といったイベント駆動型の構文、並列実行を行う `parallel ... end`、先に完了した分岐のみを採用する `finish first ... end` など、反応シナリオを宣言的に記述できる特殊な仕様がある。
- **FaCT 言語 [50]:**
  FaCT 言語は、暗号処理の実装においてタイミングサイドチャネル攻撃を防ぐために作られた外部 DSL である。パラダイムは C 言語に近い手続き型である。
  既存言語と比べて、秘密情報を表すラベルの明示や自動的な定数時間化が可能なため、安全で可読性の高い暗号実装を簡潔に記述できるというメリットがある。文法として、型宣言で `secret/public` を指定する必要があること、`ctselect(...)`・`assume(...)`・`declassify(...)` などのキーワードを用いるといった特殊な仕様がある。

- **Simplicity 言語 [51]:**
  Simplicity 言語は、暗号通貨やブロックチェーンアプリケーションにおいて、より安全で表現力の高いスマートコントラクトを実装するために作られた外部 DSL である。パラダイムは関数型言語である。
  既存言語と比べて、チューリング完全ではない（ループや再帰を持たない）設計により静的解析が容易で、実行時の計算資源を事前に把握できるというメリットがある。文法として、Gentzen のシーケント計算に基づくコンビネータ群（`iden`, `comp`, `injl`, `injr`, `case`, `pair`, `take`, `drop`）によってプログラムを構成し、変数やループを使わずに分岐やタプルを表現するという特殊な仕様がある。
- **Saiph 言語 [52]:**
  Saiph 言語は、高性能計算環境における大規模な流体力学（CFD）シミュレーションを効率的かつ簡潔に記述し、並列処理や数値解法を自動的に最適化するために作られた外部 DSL である。パラダイムは手続き型言語と関数型の特徴を併せ持つ言語である。
  既存言語と比べて、連立する偏微分方程式 (PDE) や境界条件・メッシュ定義などを数式に近い形で高水準に記述できるため、ユーザが並列化や数値スキーム実装を意識せずに効率的な開発を行えるというメリットがある。文法として、物理量の単位 (Units) を組み込んだ変数宣言、境界条件や差分演算子（`der`、`grad`、`lapla`、`div` など）を直接 DSL 構文として表現できるという特殊な仕様がある。
- **GemRBAC-DSL [53]:**
  GemRBAC-DSL 言語は、ロールベースのアクセス制御 (RBAC) ポリシーを高い表現力で記述し、モデル駆動型アプローチによる整合性チェックや自動実行可能な形式へ変換するために作られた外部 DSL である。パラダイムは宣言型言語（設定記述言語）である。
  既存のアクセス制御言語 (例: XACML, OCL など) と比べて、RBAC の標準的な機能だけでなく前提条件・委任・階層・文脈 (時空間) といった高度なポリシーを網羅的に表現でき、かつ自然言語に近い構文のため扱いやすいというメリットがある。
  文法として、`assign-role admin prerequisite participant;` や `conflicting-roles-activation assistant, admin on-same-object;` のように、RBAC の主要要素 (`role`, `user`, `permission` など) を事前に列挙し、各ポリシーを `assign-role`、`conflicting-roles-activation`、`trigger-role-hierarchy` などのキーワードで宣言する独自仕様がある。
- **HH-DSL [5]:**
  HL-DSL は、ハイフレックス（HyFlex）フレームワーク上でハイパーヒューリスティクスを開発するために作られた外部 DSL である。パラダイムは独自の記法を用いた手続き型言語である。
  既存の Java などの汎用言語 (GPL) と比べて、ハイパーヒューリスティクスの開発者がドメイン固有の記述を簡潔かつ直感的に行え、詳細に煩わされずに研究・実装に集中できるというメリットがある。文法として、`next` の呼び出しだけで次の近似解を取得

したり、`check` で解の受け入れ可否を判定し置き換えるなど、ハイパーヒューリスティクス特有の操作をキーワードベースで記述できる独自構文がある。

- **GraphIt 言語 [54]:**
  GraphIt 言語は、高性能なグラフ計算の実装を効率化し、アルゴリズムと最適化戦略を分離して記述できるようにするために作られた 外部 DSL である。パラダイムは C 言語に近い手続き型言語である。
  既存言語と比べて、グラフアルゴリズムの記述（「何を計算するか」）と最適化の記述（「どのように計算するか」）を分離したうえで、多彩な方向探索や NUMA・キャッシュ最適化など多数の最適化を容易に組み合わせられるというメリットがある。
  文法として、`edgeset.from(frontier).apply(func)` のように辺集合や頂点集合へ宣言的に処理を適用する構文、さらに label を用いてアルゴリズム部分とスケジュール部分を切り替えるといった特殊な仕様がある。
  今回の実験に用いた gpt-4 は解答ができないことを確認しているが、最新の gpt-4o は解答できることを確認している。
- **Saggitarius 言語 [55]:**
  Saggitarius 言語は、多様なテキストフォーマットをとるデータ（例: 日付・住所・電話番号・CSV・XML など）の文法をひとつの「ドメイン（文法集合）」として定義し、その中から実際の入力例に適合する文法を推論・特定するために作られた 外部 DSL である。 パラダイムは、YACC ベースの構文定義に近い。
  既存言語（YACC など）と比べて、多数のバリエーションを含む文法集合（メタ文法）を一度に指定し、少数の正例/負例から自動的に「正しい文法」を推論できるというメリットがある。文法として、`?` 付きの「オプション」ルールや `if (condition) then ...` による条件付き生成ルール、さらに制約・優先度付き選択などの特殊な文法を備えている。
- **Sparrow 言語 [56]:**
  Sparrow 言語は、大規模かつ多様なアクタ間の非同期メッセージの同期や協調を簡潔に表現するために作られた、Elixir 上の内部 DSL である。パラダイムは Elixir に準じた関数型言語である。
  既存言語と比べて、アクタの受信面に高度なジョインパターンや動的なリアクション登録を宣言的に定義できるため、複雑な並行処理をモジュール的かつ可読性高く記述できるというメリットがある。
  文法として `defpattern as` や `count: n、andAfter` といったジョインパターン特有の構文、そして `|> map(...) / |> filter(...)` などのパイプライン式を備えるという特殊な仕様がある。
- **I ❤️ Mesh 言語 [57]:**
  I ❤️ Mesh 言語は、メッシュ構造上の幾何処理を論文などに近い数式的表記で記述し、自動的にコンパイルして実行可能なコードを生成するために作られた外部 DSL である。パラダイムは宣言的かつ関数型に近い。

```
stencil blury(vector#2 float Y) {
    return Y@[0,-1] + Y + Y@[0,1];
}

return blury(input);
```

抽出したサンプルコード

```
stencil blury(vector#2 float Y) {
    return Y@[0,-1] + Y <code>;
}

return blury(input);
```

```
stencil blury(vector#2 float Y) {
    return <code> + Y + Y@[0,1];
}

return blury(input);
```

```
stencil blury(<code>) {
    return Y@[0,-1] + Y + Y@[0,1];
}

return blury(input);
```

補完タスクの例

**図 4.1.** 欠落補完タスクの例

既存言語と比べて、メッシュ要素（頂点・エッジ・面・セル）を型安全に扱え、数式そのままの記法で煩雑な実装を大幅に削減できるというメリットがある。文法として Unicode の $\mathbb{R}$ や $\sum$、$\in$ などの数学記号をそのまま利用できること、集合内包表記を用いた頂点・面・セルの集合定義が可能という特殊な仕様がある。

いずれの DSL も、コード例やドキュメントが非常に限られているため、LLM が既存学習データだけで正しく推測することは難しく、本研究の対象領域として適切であるといえる。

## 4.3　コード補完タスクの生成手順

本研究では、IDE でのコード補完を模擬した評価を行うため、ソースコードの一部を欠落させ、そこを LLM で補完させる形式のタスクを作成する。その際、削除する部分の大きさを複数パターン用意し、実際にどの程度の難易度の補完が必要になるのかを評価する。ここではサンプルコードの抽出方法とプレースホルダ置換、および欠落アルゴリズムについて解説する。

### 4.3.1　サンプルコードの抽出とプレースホルダ置換

各言語について、次のような情報源からサンプルコードを収集した。

1. 公式ドキュメントや言語仕様書に付随する簡易的なサンプル

2. 論文や学会発表スライド上で示されるデモ用コード断片
3. オープンソースリポジトリやコミュニティ内の共有例

得られたサンプルコードをパーサで解析し、複数のノード構造が含まれることを確認してから、プレースホルダ <code> を挿入するための編集を行う。

のようにする。IDE 上では「ここに何を書くか提案してほしい」という状況になるわけである。本研究では欠落させるトークン数を 4 パターン (1,3,5,7) 用意し、それぞれ 10 件ずつ程度のサンプルを作成することで、計 40 件前後のタスクを 1 言語あたり生成する。

### 4.3.2 トークン削除アルゴリズム

コードから欠落を作り出す方法を単にランダムに実行すると、実際の IDE での補完と異なる可能性がある。そこで本研究では、アルゴリズム 3 のような段階的手法を採用している。アルゴリズムの手順は以下の通りである。

1. **AST を作成:** ソースコードをパーサで解析し、抽象構文木 (AST) を構築する。
2. **ノードのランダム選択:** 削除数 $m$ 以上のトークンによって構成されるノードをランダムで 1 つを選ぶ。こうすることで、文法的に意味のある単位 (たとえば条件式や引数リスト) を欠落対象にしやすい。
3. **末尾側からトークンを削除してプレースホルダ化:** 実運用の IDE では、ユーザが途中まで入力し、残りを補完に任せるケースが多いことを想定し、ノードを構成するトークンを出現逆順に $m$ つ削除し、 <code> に置き換える。

このようにして得られた欠落タスクは、IDE での自然な入力シチュエーションに近いと考えられる。具体例としては、関数定義の引数リスト後半が欠落した状態や、多重ループの内側で更新式が消えている状態など、DSL の特徴的な文法構造を直に補完させるタスクが作成される。図 4.1 に、この手法で生成された欠落補完タスクの例を示す。

## 4.4 実験設定の詳細

次に、本研究での具体的な実験条件やパラメータの設定について述べる。LLM のバージョンやデータセットの分割方法、フィードバックループの回数制限などを定義し、後述の評価手法 (§??) と組み合わせて結果を検証する。

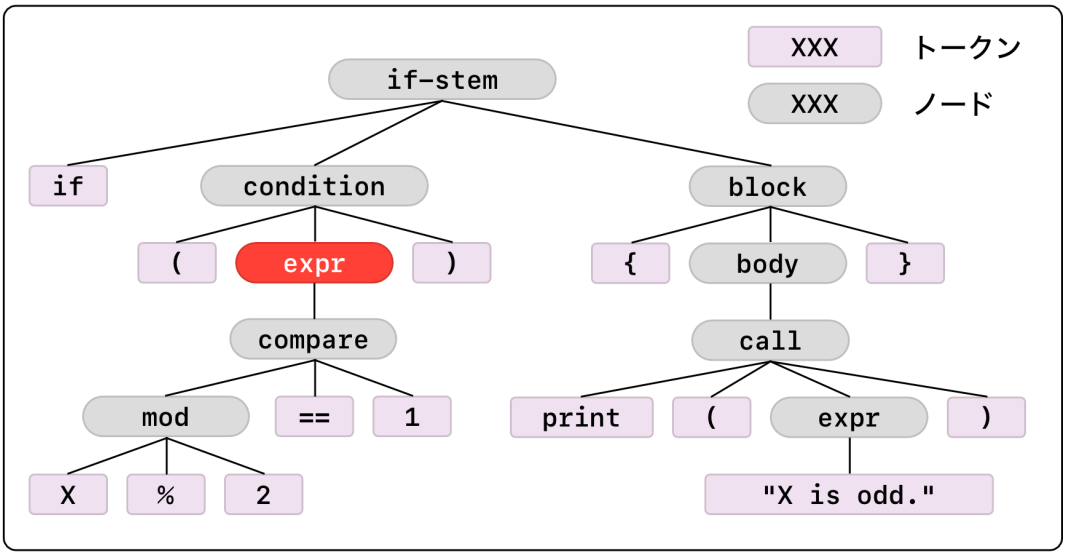
### 4.4.1 LLM のバージョンと利用形態

本研究では、OpenAI 社の gpt-4-turbo (2024-04-09 API) を用いて実験を行う。

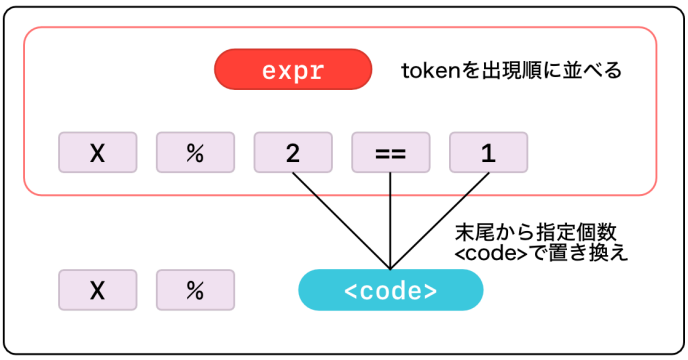
- **Temperature や Top-p はデフォルト:** 特定の最適化を施すことなく、Temperature(生成の多様性を調整) や Top-p(確率質量削減) は既定値のまま利用し、Structured-Output などの機能は用いない。これにより、現実の開発現場に近い形で補完タスクを

```
if (x % 2 == 1) {
    print("X is odd.");
}
```

元のコード




ASTを構築してランダムなノードを選択




<code>への置換

```
if (x % <code>) {
    print("X is odd.");
}
```

補完タスク

図 4.2. トークン削除アルゴリズムの例

**Algorithm 3** トークン削除アルゴリズム

**Require:** ソースコード $S$, トークン削除数 $m$

**Ensure:** 欠落補完タスク用コード $S'$

$A \leftarrow \text{Parser}(S)$ {AST を生成 }
候補ノード集合 $\leftarrow \{\}$
**for all** ノード $n$ in $A$ **do**
  $t_n \leftarrow$ ノード $n$ が含むトークン列
  **if** $\text{length}(t_n) \geq m$ **then**
    候補ノード集合 $\leftarrow$ 候補ノード集合 $\cup \{n\}$
  **end if**
**end for**
$n_{\text{chosen}} \leftarrow$ 候補ノード集合からランダム選択
$t \leftarrow$ 末尾 $m$ 個のトークンを削除 $(t_{n_{\text{chosen}}})$
$t_{\text{replaced}} \leftarrow t + \text{code}$ { 末尾にプレースホルダを追加 }
$S' \leftarrow$ 元のソースコード中で $n_{\text{chosen}}$の部分を $t_{\text{replaced}}$に置換
**return** $S'$

実施することができる。

- **連続会話形式での利用:** フィードバックループを通じ、誤った補完に対してエラーメッセージを再度モデルに送る手順を実装するため、実験環境でセッションを保持してやり取りを行う。これにより、会話履歴が大きくなるとトークン数が増加する様子を観察できる。

なお、ChatGPT-4 が元々持つ言語知識は非常に豊富ではあるが、未知言語に関するサンプルがほとんど含まれていないと推測されるため、本研究の狙いである「Low-Resource 言語に対する補完性能の検証」には適切と判断した。

### 4.4.2 データセットと分割

- **学習用セット (約 70%):** 取得したサンプルコードのうち、約 70% をフィードバックループでの学習プロセス (プロンプト生成・圧縮の工程) に使用する。ここでは欠落補完タスクを作成し、LLM と繰り返しやり取りすることで、最終的に正しい補完が得られるまでのプロセスを記録・圧縮する。
- **評価用セット (約 30%):** 残りの約 30% は評価専用とし、(a) フィードバックループを一切行わない状態と、(b) すでに得られた圧縮プロンプトを適用した状態での補完精度・推論コストを比較する。評価用セットを分離することで、未見のコード例に対する汎化性能を測定しやすくなる。

各 DSL ごとに、公式サンプルやコミュニティ例などを合計 10〜15 件程度確保できる場合

が多かった。これらのファイルから上記のように学習用と評価用に分割し、1 つのファイルから複数の欠落タスクを作り出して合計タスク数を増やしている。最終的には 1 言語あたり 30 件前後のタスクを学習用に、10〜15 件を評価用に用意する形となった。

### 4.4.3　フィードバックループと実行回数

- **最大 10 回のフィードバック:** 学習用セットの欠落タスクに対し、1 度 LLM から回答を得た後、パーサで検証してエラーがあればエラーメッセージを再度 LLM へ返す。これを最大 10 回繰り返し、それでも正しくコンパイル可能なコードが得られない場合は打ち切る。エラーには文法的なものから型エラー、予約語の誤使用など様々あるが、いずれもパーサから返されるメッセージを LLM へ渡す手順は同じである。
- **繰り返し試行:** 同じ欠落タスクを 10 回独立して試行し (ただし履歴は共有しない)、確率的な出力の揺らぎが全体の成功率にどう影響するかを観測する。1 言語につき 300 回程度のフィードバックループを実施し、応答のトークン数や推論時間、エラーが消えるまでの平均反復回数などを統計的に記録する。

これにより、フィードバックループを深く回した場合の精度向上と、履歴の増大に伴うトークンコスト上昇の具体的数値を把握できる。その後、この履歴を圧縮して ICL プロンプトを生成し、圧縮後プロンプトで推論を行った場合の速度向上やトークン削減率を比較検証する計画である。

## 4.5　実験手順のフロー

提案手法の有効性を確かめるため、本研究では以下の 4 段階ステップを順番に実行し、各段階でのコード補完精度や推論に要するリソースの変化を測定する。

1. **自然言語ドキュメントのみを使用 (ベースライン):** 新言語ドキュメント (言語仕様書のサマリや主要な予約語の説明など) を LLM へ提示し、補完タスクに対して一度だけ応答してもらう。ここではフィードバックループを全く導入しないため、事前知識がないと LLM がハルシネーションを起こしやすい状況を確認し、ベースラインとしての正解率や文法エラー率を取得する。
2. **自然言語ドキュメント ＋ フィードバックループ:** 次に、(1) と同じタスクを与えるが、文法エラーが検出された場合はエラーメッセージを LLM へ提示し、修正を繰り返して最終的に正解コードが得られるかを確認する。フィードバックループによる補正効果がどの程度存在するか、また会話履歴が大きくなるにつれ推論コスト (トークン数) がどう増加するかを計測する。
3. **自然言語ドキュメント ＋ フィードバックループの最終プロンプト (最終プロンプト)** (2) で得られた複数のタスクにおける、ループ最後のプロンプトを結合して LLM に提

示し、フィードバックループは行わずに補完を行わせる。フィードバックループを用いない修正履歴による精度向上を観測する。ループ最後のプロンプトはタスクから全種類のサンプルコードが含まれるようにランダムに選んだ。

4. **履歴圧縮と ICL プロンプトの生成:** (2) で複数の誤答や修正手順、正答例が得られるため、それらを連結して LLM に要約・圧縮させる。圧縮にあたっては、エラー原因と修正方法に関わる核心的な情報だけを抽出するよう依頼する。この圧縮後の ICL プロンプトを「新言語の利用におけるエラー対処ノウハウ集」と位置づけ、(4) 以降はこれを既定のベースプロンプトとして活用する。
5. **ICL プロンプト + 追加的なフィードバックループ:** 圧縮された ICL プロンプトを提示した状態で、(2) と同じ補完タスクに再挑戦する。今度はフィードバックループがなくてもエラーの多くが最初から解消されるか、あるいは必要なフィードバック回数が激減するかをチェックする。また、それでも解消されないエラーが発生した場合にフィードバックループを少数回だけ導入した時の精度向上も合わせて評価する。ここで、推論時に必要なトークンがどの程度削減されるかを測定する。

この 4 段階フローを実施することで、(1) ベースラインの精度とコスト、(2) フィードバックループ導入による精度改善とコスト増大、(3) 修正履歴による精度向上、(4) 圧縮プロンプトの生成手順、(5) 圧縮後プロンプトを適用した際の精度・コスト評価という一連の流れを把握できる。最終的には、提案手法 (Compressed Prompt Generation) がどのようにコード補完精度を向上させ、かつ推論コストを削減するかが体系的に明らかになる。

## 4.6 評価手法

本節では、実験結果を評価するために用いる指標や手法を示す。フィードバックループの有無、圧縮の有無といった実験条件ごとに以下の項目を定量化し、比較を行う。

1. **文法正解率 (Grammar Accuracy):** パーサでエラーが発生せずにコンパイル可能となる割合を測定する。コード補完において最も基本となる指標であり、未知言語への対応力を直接示す。特に DSL では予約語や特殊構文が非常に重要なので、文法エラーをどれだけ減らせるかが焦点となる。
2. **内容一致度 (BLEU スコアによる類似度):** 欠落部に対して本来意図していた正解コードとの類似度を、BLEU スコアという自動翻訳品質評価指標を用いて数値化する。比較は欠落させた部分と LLM によって補完された部分のみに関して行った。BLEU スコアは 4-gram のものを用い、Add-One Smoothing を利用した。
3. **推論時間・トークン消費量:** API 経由で行う各問い合わせ (初回の補完やエラー修正依頼など) に要した推論時間と、消費した入力 + 出力トークン数を記録し、合計や平均を算出する。フィードバックループによる会話履歴の肥大化がどの程度のオーバーヘッド

をもたらすか、圧縮後はどれだけ緩和されるかを比較する上で重要な指標となる。

4. **フィードバックループの回数:** エラーを修正して正解に辿り着くまでに必要な回数や、打ち切りとなるケースの割合を測定する。提案手法導入前後で大きく変化する可能性があり、圧縮プロンプトの効果を定量的に示すデータとして活用する。

LLM が補完タスクを誤解し、<code>と書いた部分以外を書き換えてしまう場合がある。このような場合は補完失敗とし、文法は不正解、BLEU スコアは 0 とした。

これらの指標を総合的に評価することで、未知言語に対するコード補完タスクでの精度向上・コスト抑制・回数削減の度合いがどのように変化するかを分析できる。特に、文法的正解率やトークン消費量の推移を詳細に観測することで、Compressed Prompt Generation 手法の有効性を具体的に測定できる見込みである。

以上が、本実験で実施する全体的な流れ、そして評価のための準備・手法となる。本章で述べたように、本研究が取り扱う DSL はいずれも LLM が事前学習で充分にカバーしていない未知領域であり、その意味で本実験の結果は Low-Resource な言語への適用事例として一定の一般化可能性を持つと考えている。次章では、これらの実験を通して得られた結果を分析し、提案手法が未知言語のコード補完タスクにおいてどの程度貢献できるかを評価する。

## 4.7　実験結果

本節では、前節までに述べた実験設定に基づき得られた実験結果を示し、文法正解率や BLEU スコアといった精度指標、ならびに推論時のコストに関わる指標を総合的に評価する。さらにそれらを踏まえた考察を述べる。

### 4.7.1　文法正解率と BLEU スコア

表 4.1 および表 4.2 に、代表として欠損トークン数が 5 の場合の各言語の文法正解率と BLEU スコアを示す。また、表 4.3 および表 4.4 には、欠損トークン数別の全言語の欠損トークン数ごとの平均を示す。

まず、ベースラインにあたる「自然言語ドキュメントのみを提示した場合」の結果を見ると、置換（欠落）トークン数が増えるにつれ、文法正解率と BLEU スコアの両方が徐々に低下していることが分かる。1 トークン欠落では文法正解率がおよそ 73% を示すのに対し、7 トークン欠落になると 54% 程度まで低下する（表 4.3）。BLEU スコアも同様に、1 トークン欠落のときは 0.46 程度であるのに対し、7 トークン欠落で 0.30 近くまで落ち込んでいる（表 4.4）。

これは未知言語に対する LLM の初期理解が不十分であることを示唆しており、欠落部分が増えると、自然言語ドキュメントだけでは十分に正確なコードを再現できないことを意味する。

フィードバックループを導入することで文法正解率は平均して 5〜6% ほど上昇し、特に 3〜5 トークン欠落のように中規模の欠落量がある場合で顕著な効果が確認できる。これは、初回出力で誤った構文が生成されたとしても、パーサからのエラーメッセージを反復して提示す

| 言語 | Document のみ | フィードバック | 最終プロンプト | ICL プロンプト | ICL+ フィードバック |
|---|---|---|---|---|---|
| ComPOS 言語 | 62% | 72%(+**10%**) | 70%(+**8%**) | 70%(+**8%**) | 72%(+**10%**) |
| FaCT 言語 | 54% | 61%(+**7%**) | 57%(+**3%**) | 60%(+**6%**) | 64%(+**10%**) |
| Forma 言語 | 46% | 56%(+**10%**) | 58%(+**12%**) | 57%(+**11%**) | 57%(+**11%**) |
| GemRBAC-DSL | 41% | 48%(+**7%**) | 42%(+**1%**) | 46%(+**5%**) | 46%(+**5%**) |
| GraphIt 言語 | 77% | 81%(+**4%**) | 78%(+**1%**) | 79%(+**2%**) | 82%(+**5%**) |
| HH-DSL | 61% | 66%(+**5%**) | 72%(+**11%**) | 65%(+**4%**) | 66%(+**5%**) |
| I ❤️ Mesh 言語 | 30% | 36%(+**6%**) | 32%(+**2%**) | 34%(+**4%**) | 37%(+**7%**) |
| Saggitarius 言語 | 81% | 86%(+**5%**) | 88%(+**7%**) | 85%(+**4%**) | 86%(+**5%**) |
| Saiph 言語 | 58% | 61%(+**3%**) | 57%(**-1%**) | 61%(+**3%**) | 64%(+**6%**) |
| Simplicity 言語 | 48% | 54%(+**6%**) | 52%(+**4%**) | 53%(+**5%**) | 57%(+**9%**) |
| Sparrow 言語 | 58% | 64%(+**6%**) | 62%(+**4%**) | 62%(+**4%**) | 62%(+**4%**) |
| 平均 | 56% | 62%(+**6%**) | 60%(+**4%**) | 61%(+**5%**) | 63%(+**7%**) |

**表 4.1.** 文法正解率 - 5 トークン

ることで、正しい予約語や制御フローに到達しやすくなるためと考えられる。

フィードバックループの最終プロンプトのみを用いた場合は、フィードバックループと比べて 2〜3% 程度ほど文法正解率は落ち込み、BLEU スコアも 0.01 程度減少した。また、複数のタスクのフィードバックループ結果を結合した結果プロンプト長および推論時間も Document のみより増加している。

フィードバックループの内容を圧縮して ICL プロンプトに落とし込むと、置換トークンが 3 個以上のケースで 4〜5% 程度の追加的な正解率向上が見られた。1 トークン欠落ではもともと 73% の正解率があるため上昇幅は限定的だが、3〜7 トークン欠落の場合は ICL プロンプトを活用することで最初から適切な構文を提示しやすくなる効果が大きいと考えられる。

BLEU スコアについても概ね同様の傾向が見られ、3 トークン以上の欠落においてはフィードバックループや ICL プロンプト導入で 0.01〜0.02 ほどスコアが上昇している。一方で 1 トークン欠落のケースではもともとのベースラインがある程度高い（0.46）ため、ICL プロンプトでわずかに下がるケースが観測されているが、これは文法的には正しいが実装が異なるという別解が増えた影響とも考えられる。総じて、フィードバックループと ICL プロンプトの組み合わせ（ICL+ フィードバック）が最終的にもっとも高い文法正解率と比較的良好な BLEU スコアを達成することが確認された。

各言語ごとの文法正解率と BLEU スコアの変化を見ると、ベースラインの文法正解率が低いほど ICL プロンプトの効果が強く現れることがわかる。ComPOS 言語や Saggitarius 言語、Simplicity 言語などの文法正解率が高い言語は、フィードバックループや ICL プロンプトの導入による効果が限定的である一方、FaCT 言語や Forma 言語、I ❤️ Mesh 言語などの文

| 言語 | Document のみ | フィードバック | 最終プロンプト | ICL プロンプト | ICL+ フィードバック |
|---|---|---|---|---|---|
| ComPOS 言語 | 0.32 | 0.34(+**0.02**) | 0.30(**-0.02**) | 0.33(+**0.01**) | 0.32(+**0.00**) |
| FaCT 言語 | 0.30 | 0.31(+**0.01**) | 0.30(+**0.00**) | 0.33(+**0.03**) | 0.37(+**0.07**) |
| Forma 言語 | 0.32 | 0.36(+**0.04**) | 0.30(**-0.02**) | 0.37(+**0.05**) | 0.35(+**0.03**) |
| GemRBAC-DSL | 0.30 | 0.33(+**0.03**) | 0.32(+**0.02**) | 0.33(+**0.03**) | 0.32(+**0.02**) |
| GraphIt 言語 | 0.37 | 0.38(+**0.01**) | 0.32(**-0.05**) | 0.38(+**0.01**) | 0.38(+**0.01**) |
| HH-DSL | 0.37 | 0.35(**-0.02**) | 0.37(+**0.00**) | 0.34(**-0.03**) | 0.35(**-0.02**) |
| I ❤️ Mesh 言語 | 0.25 | 0.26(+**0.01**) | 0.26(+**0.01**) | 0.26(+**0.01**) | 0.25(+**0.00**) |
| Saggitarius 言語 | 0.41 | 0.44(+**0.03**) | 0.42(+**0.01**) | 0.44(+**0.03**) | 0.44(+**0.03**) |
| Saiph 言語 | 0.34 | 0.36(+**0.02**) | 0.35(+**0.01**) | 0.35(+**0.01**) | 0.36(+**0.02**) |
| Simplicity 言語 | 0.29 | 0.30(+**0.01**) | 0.29(+**0.00**) | 0.33(+**0.04**) | 0.34(+**0.05**) |
| Sparrow 言語 | 0.33 | 0.30(**-0.03**) | 0.30(**-0.03**) | 0.32(**-0.01**) | 0.32(**-0.01**) |
| 平均 | 0.327 | 0.339(+**0.012**) | 0.321(**-0.006**) | 0.344(+**0.016**) | 0.345(+**0.018**) |

**表 4.2.** BLEU - 5 トークン

| Token | Document のみ | フィードバック | 最終プロンプト | ICL プロンプト | ICL+ フィードバック |
|---|---|---|---|---|---|
| 1 | 73% | 79%(+**6%**) | 75%(+**2%**) | 76%(+**3%**) | 78%(+**5%**) |
| 3 | 61% | 66%(+**5%**) | 66%(+**5%**) | 65%(+**5%**) | 67%(+**6%**) |
| 5 | 56% | 62%(+**6%**) | 60%(+**4%**) | 61%(+**5%**) | 63%(+**7%**) |
| 7 | 54% | 60%(+**6%**) | 58%(+**3%**) | 59%(+**4%**) | 61%(+**7%**) |
| 平均 | 61% | 67%(+**5%**) | 64%(+**3%**) | 65%(+**4%**) | 67%(+**6%**) |

**表 4.3.** 文法正解率の全言語平均 (トークン別)

法正解率が低い言語は、特に ICL プロンプトの導入で大きな効果が見られる傾向がある（表 4.1、表 4.2)。文法的に複雑な言語や、LLM が未知の言語構造を理解するのに時間がかかる言語ほど、フィードバックループや ICL プロンプトの効果が顕著に現れるということを示唆している。

### 4.7.2　推論コストとフィードバックループ回数

フィードバックループを導入した場合、表 4.6 のように平均 3.94 回の再生成を行う結果となり、都度履歴が蓄積されるため消費トークン数はベースライン比で約 4 倍程度に増加している（表 4.5)。推論時間も平均 4.88 秒に達しており、軽量な補完を期待するリアルタイムな IDE 環境では負担が大きいと言わざるを得ない。

| トークン | Document のみ | フィードバック | 最終プロンプト | ICL プロンプト | ICL+ フィードバック |
|---|---|---|---|---|---|
| 1 | 0.459 | 0.481(+**0.022**) | 0.464(+**0.005**) | 0.45(**-0.009**) | 0.443(**-0.016**) |
| 3 | 0.347 | 0.369(+**0.022**) | 0.355(+**0.008**) | 0.362(+**0.015**) | 0.365(+**0.018**) |
| 5 | 0.327 | 0.339(+**0.012**) | 0.321(**-0.006**) | 0.344(+**0.016**) | 0.345(+**0.018**) |
| 7 | 0.299 | 0.312(+**0.013**) | 0.306(+**0.007**) | 0.313(+**0.014**) | 0.316(+**0.017**) |
| 平均 | 0.358 | 0.375(+**0.017**) | 0.362(+**0.004**) | 0.367(+**0.009**) | 0.367(+**0.009**) |

**表 4.4.** BLEU の全言語平均（トークン別）

| Token | Document のみ | フィードバック | 最終プロンプト | ICL プロンプト | ICL+ フィードバック |
|---|---|---|---|---|---|
| 1 | 2493 | 12620(+**10127**) | 5553(+**3060**) | 3159(+**666**) | 9485(+**6992**) |
| 3 | 2595 | 12946(+**10352**) | 5988(+**3394**) | 3275(+**681**) | 9787(+**7192**) |
| 5 | 2563 | 12776(+**10213**) | 6030(+**3467**) | 3177(+**614**) | 9710(+**7148**) |
| 7 | 2529 | 12703(+**10174**) | 5979(+**3449**) | 3221(+**691**) | 9708(+**7178**) |
| 平均 | 2545 | 12761(+**10217**) | 5888(+**3343**) | 3208(+**663**) | 9673(+**7128**) |

**表 4.5.** 平均消費トークン数 の全言語平均 (トークン別)

| Document のみ | フィードバック | 最終プロンプト | ICL プロンプト | ICL+ フィードバック |
|---|---|---|---|---|
| 1 (固定) | 3.94(+**2.94**) | 1 (固定) | 1 (固定) | 3.17(+**2.17**) |

**表 4.6.** フィードバックループ回数の全言語平均

| Document のみ | フィードバック | 最終プロンプト | ICL プロンプト | ICL+ フィードバック |
|---|---|---|---|---|
| 1.17s | 4.88s(+**3.71s**) | 2.80s(+**1.63s**) | 1.26s(+**0.09s**) | 4.08s(+**2.91s**) |

**表 4.7.** 推論時間 (秒) の全言語平均

対照的に ICL プロンプトを導入した場合、フィードバックループなしで初回から正しいコードを生成できるケースが増えるため、消費トークン数はベースライン比で +24〜+27% 程度に抑えられている（表 4.5）。推論時間も約 1.26 秒と、Document のみの場合の 1.17 秒と比べてもわずかな増加に留まっている（表 4.7）。

さらに ICL+ フィードバックの形態では、フルフィードバック（未圧縮）ほどの大幅なコスト増加は起こらないものの、やはり問い合わせ回数が 3.17 回ほど必要になるため、最終的に

推論時間は 4.08 秒程度まで上昇する。文法正解率は最も高いが、リアルタイム性との兼ね合いをどう考慮するかはユースケース次第である。

| 言語 | Document のみ | フィードバック | 最終プロンプト | ICL プロンプト | ICL+ フィードバック |
|---|---|---|---|---|---|
| ComPOS 言語 | 3097 | 11982(+**8885**) | 7753(+**4656**) | 3108(+**11**) | 4875(+**1778**) |
| FaCT 言語 | 3086 | 14368(+**11282**) | 4983(+**1897**) | 3167(+**81**) | 11025(+**7939**) |
| Forma 言語 | 2297 | 14833(+**12536**) | 5975(+**3678**) | 3328(+**1031**) | 9613(+**7316**) |
| GemRBAC-DSL | 954 | 7631(+**6677**) | 3976(+**3022**) | 1501(+**547**) | 6304(+**5350**) |
| GraphIt 言語 | 4516 | 13677(+**9161**) | 7865(+**3349**) | 4680(+**164**) | 13930(+**9414**) |
| HH-DSL | 2052 | 6033(+**3981**) | 6127(+**4075**) | 2788(+**736**) | 4832(+**2780**) |
| I ❤️ Mesh 言語 | 4281 | 32107(+**27826**) | 8951(+**4670**) | 6123(+**1842**) | 19136(+**14855**) |
| Saggitarius 言語 | 709 | 2317(+**1608**) | 2117(+**1408**) | 799(+**90**) | 1827(+**1118**) |
| Saiph 言語 | 2894 | 14902(+**12008**) | 6127(+**3233**) | 3391(+**497**) | 11232(+**8338**) |
| Simplicity 言語 | 2064 | 12577(+**10513**) | 6127(+**4063**) | 2972(+**908**) | 14274(+**12210**) |
| Sparrow 言語 | 2241 | 10105(+**7864**) | 6329(+**4088**) | 3093(+**852**) | 9767(+**7526**) |
| 平均 | 2563 | 12776(+**10213**) | 6030(+**3467**) | 3177(+**614**) | 9710(+**7148**) |

**表 4.8.** 平均消費トークン数 - 5 トークン

言語ごとの消費トークンを欠損トークン数が 5 の場合に見ると、ベースライン比で最も大きく増加したのは I ❤️ Mesh 言語である。これは初回の文法正解率が低く、また言語仕様が複雑であるため、フィードバックループを多用することで正しいコードを生成するために多くのトークンを消費する必要があるためと考えられる。一方で、GraphIt 言語や Saggitarius 言語、HH-DSL などの初回の文法正解率が高い言語は、フィードバックループを多用しなくても比較的正確なコードを生成できるため、消費トークン数の増加率が抑えられている（表 4.8）。

### 4.7.3　総合的な考察

1. **フィードバックループの精度向上効果:** 追加的な問い合わせとエラーメッセージの提示により、初回出力が不正でも修正していくことで文法正解率が大きく向上することが確認できた。一方で問い合わせ履歴が大幅に膨らむため、推論時間や消費トークン数は高騰する。
2. **ICL プロンプトによるコスト削減:** フィードバックループから得られたエラー修正情報を圧縮し、要点を ICL プロンプトに組み込むことで、最初から正しい構文を生成しやすくなる。結果としてベースライン (ドキュメントのみ) とほぼ変わらない推論速度と、4 倍程度に膨らんでしまうことのない安定したトークン使用量を両立できる。
3. **ICL+ フィードバックの高精度:** ICL プロンプトに加え、必要に応じて少数回のフィードバックループを行うと、文法正解率や BLEU スコアがさらに向上して最良の結果と

なる反面、トークン消費と推論時間もそれなりに増加する。ある程度複雑なコード補完や高精度を要求する場面なら十分に有用だが、IDE での常時利用には検討が必要となる。

4. **用途に合わせた選択:** 全体的に、フィードバックループを最大限活用すれば高精度に到達できるものの、リアルタイム性が著しく低下する。一方、圧縮 ICL プロンプトのみでは精度が多少落ちるケースもあるが（特に 1 トークン欠落で BLEU スコアが若干低下した例など）、補完を素早く得られる利点が大きい。場面によって両手法を使い分けることが実用的と考えられる。

総じて、LLM に未知な DSL や新言語を扱う状況下においては、最初にフィードバックループによる修正履歴を蓄積し、それを圧縮して ICL プロンプトを整備するアプローチが効果的であるといえる。定常的な利用シナリオでは圧縮 ICL プロンプトを使い、複雑なケースに直面した際には改めてフィードバックループを適用する、という運用形態が最もバランスが良いと推測される。

以上の観点から、提案手法が未知言語におけるコード補完精度を高めつつ、トークンコストと推論時間の増大を抑制するうえで効果的な方策となり得ることが示唆される。

# 第 5 章

# Compressed Prompt Generation の総括と今後の展望

## 5.1 研究成果のまとめ

本研究の狙いは、大規模言語モデル (LLM) が学習データを十分に含んでいない新規プログラミング言語やドメイン固有言語 (DSL) に対しても、実用的なコード補完を提供する仕組みを構築する点にあった。従来は Full Fine-tuning や Partial Fine-tuning、Parameter-Efficient Fine-Tuning、あるいはプロンプト内での複数回のフィードバックループを用いて補完精度を高める方法が試みられているものの、以下の問題が見受けられた。

- 新言語のように学習データが不足している場合や、大規模な計算資源が得られない場合、これらの Fine-tuning を適用することが難しい。
- フィードバックループを多数回行うと、推論履歴が蓄積しトークン数が加速度的に増大する。膨大なトークン数は IDE 上で使うには現実的でない推論時間や API コストをもたらす可能性がある。

## 5.2 提案手法の意義

Compressed Prompt Generation が持つ主な意義は、以下の 4 点である。

1. **不十分な学習データへの対処:**
   - **Low-Resource 言語への対応:** 新しい言語や研究目的の DSL など、十分なコードデータセットを用意しづらい環境でも初期から利用できる。追加のファインチューニングや大規模なデータ収集が難しい場合にも、最低限のコード例と自然言語ドキュメント、パーサという新言語作成に自然に作成するもののみから LLM の新言語対応を行える。
   - **文法エラーを用いた学習:** 文法的に誤った生成を行いがちな未知言語に対し、パー

サから返されるエラーメッセージをもとに修正事例を蓄積・抽出できるため、事前に大規模な正解データがなくてもエラー回避パターンを学習しやすい。

2. **精度とコストの両立:**
   - **トークン消費の大幅な抑制:** フィードバックループを素朴に回すと、累積履歴が肥大化し推論コストが急激に増大するが、本手法では複数回の修正履歴を要約した小規模のプロンプト（ICL プロンプト）をあらかじめ用意できる。フィードバックを回した最後のプロンプトのみを利用する場合と比べても圧縮したプロンプトは小さくなる。ユーザーサイドでの補完フェーズでは短いプロンプトを使うだけで済むため、トークン消費量や推論時間を抑えながら補完精度を維持することが可能である。
   - **リアルタイム性:** IDE での頻繁な補完要求に対して、長大な履歴を都度送信する必要がなくなるため、補完応答までの待ち時間を低減でき、実用的な速度でのコード提案を実現しやすい。
   - **精度:** フィードバックループを回す場合と比べると少し精度は劣るが、Document のみを与えた場合や最後のプロンプトのみを利用する場合を比べて高精度となった。エラー情報を抽出していることによるものと考えられる。
3. **IDE への容易な統合と配布:**
   - **大規模パラメータの再配布が不要:** 通常は未知言語に対応するファインチューニングモデルや追加学習済みパラメータを配布する必要があるが、本手法では圧縮後の「ICL プロンプト」だけを共有すればよい。また、フィードバックの最終プロンプトのみを提供する場合と比べても少ない数キロバイト〜数十キロバイト程度の文字列で済むため、言語コミュニティ内でも導入が容易。
   - **言語仕様変更への追従:** 新言語ではバージョンアップにより文法や予約語が変わるケースがあるが、圧縮プロンプトは比較的手軽に再生成できる。そのため IDE のシステムプロンプトを差し替えるだけで新仕様にも追従しやすい。

## 5.3 制限事項と今後の課題

提案手法には以下のような制限や課題が残されている。

1. **文法エラー中心のフィードバック:**
   - **意味論エラーや実行時エラーへの未対応:** 本研究では主にパーサを用いた文法レベルのエラー修正に重きを置いているが、型推論や依存関係、意味論エラーは扱えていない。これらを検出するために、より高度な静的解析器や単体テストの結果をフィードバックできれば、さらに正確な補完が可能になると想定される。
   - **テスト実行との連携:** 単体テスト・統合テストを自動的に回して失敗情報を LLM に渡す仕組みを組み込めば、ロジックの誤りに対しても修正履歴を蓄積できる可能

性がある。

2. **フィードバックループの限界:**
   - **抽象的なエラーメッセージの難しさ:** DSL によってはパーサやコンパイラのエラーメッセージが極端に抽象的であったり、同じエラーがさまざまな原因で起こる場合がある。LLM が誤って理解し、修正をうまく行えないリスクが依然として残る。
   - **エラーの繰り返し生成:** 同じ文法ミスを繰り返すケースや、複数パターンの誤りが複合して再帰的にエラーが起こるケースがある。フィードバック情報をより構造化してやり取りしない限り、単純なエラーメッセージ再提示では限界がある。
3. **専門知識が必要な分野への対応:**
   - **ドメイン知識の補填不足:** 金融、科学技術計算など、専門的な概念や業務知識が必要な領域の DSL では、文法だけ修正しても正しいコードにならない場合が多い。提案手法の枠組みを超えて、専門家のノウハウやドキュメントをどう圧縮プロンプトに盛り込むかが重要となる。
   - **追加データの統合:** 実務では API リファレンスやドメイン特有のルールが欠かせない。こうした情報源をどのように取り込み、圧縮して提示するかという設計が課題として残る。
4. **圧縮の質:**
   - **圧縮時の情報損失リスク:** LLM による要約・圧縮プロセスでは、重要なエラー修正事例が省略されたり、細かな実装指針が抜け落ちることがある。制御可能な要約アルゴリズムや手動での後工程チェックを組み合わせることでさらなる精度向上を行える可能性がある。
   - **冗長な要約や重複内容:** 圧縮プロンプト生成時に、似たようなエラー修正が重複して記載されてしまい、結果的にプロンプト自体が肥大化するケースがあった。自動要約の安定性や品質管理が技術的なチャレンジとなる。

## 5.4　今後の応用可能性

提案手法は新言語や DSL だけでなく、既存の主要言語における特殊なライブラリやフレームワークに対しても利用できる可能性がある。Python や Java などメジャー言語でも、あまり使われていない書式や構文でエラーが起こりやすい場面が想定される。圧縮 ICL プロンプトを事前に用意し、定型的なエラー修正知識を利用者と共有する形で、コード補完の品質向上と学習コスト低減が見込まれる。

開発プロジェクトの試行錯誤から蓄積されるノウハウを収集し、圧縮してチーム内で共有すれば、継続的なナレッジベースとして活用できる。LLM への依存度が高まる中、大規模履歴そのものを使うのではなく、必要十分な形にまとめる手法はさまざまな分野への適用が期待される。

## 5.5 最終的な展望

Compressed Prompt Generation は、未知言語や Low-Resource 言語における LLM を用いたコード補完を高精度化する有力なアプローチとなり得る。従来のフィードバックループ中心の手法に比べ、圧縮された ICL プロンプトを用いることで、初回推論段階から高い文法正解率を得つつ、推論時間とトークン消費を大幅に抑えられる点が大きな特徴である。ICL プロンプトに少数回の追加フィードバックを組み合わせれば、さらなる補完精度向上も可能な柔軟性が確認された。

新言語や Low-Resource 言語は、学習データやユーザ事例が充実しにくいが、本手法は最小限のコード例と自然言語ドキュメント、そしてパーサさえあれば導入を開始できる。コミュニティの拡大とともにプロンプト圧縮手順を再実行し、より高い精度の補完環境を継続的に提供できる点も魅力である。

複雑な DSL や科学技術向け言語、専門知識を要する分野へ適用範囲を拡大し、LLM へ専門情報を効率よく伝える仕組みを強化することが今後の大きな課題となる。絶えず進化する LLM と連携し、開発者が自然に知見を集約できる環境を目指すことで、圧縮 ICL プロンプトはより幅広い領域に貢献できると考えられる。新言語の普及やイノベーションの加速を支援する手法として、今後さらに発展していくことが期待される。

# 発表文献と研究活動

(1) 大渕 雄生, 千葉 滋 第 26 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（PPL 2024）, 日本ソフトウェア科学会 プログラミング論研究会, 2024 年 3 月 5 日-7 日.

# 参考文献


[1] Ivens Portugal, Paulo Alencar, and Donald Cowan. A survey on domain-specific languages for machine learning in big data, 2016.

[2] Mahesh Ravishankar, Justin Holewinski, and Vinod Grover. Forma: a dsl for image processing applications to target gpus and multi-core cpus. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, GPGPU-8, p. 109–120, New York, NY, USA, 2015. Association for Computing Machinery.

[3] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.*, Vol. 48, No. 6, p. 519–530, June 2013.

[4] Alfred Åkesson, Görel Hedin, and Niklas Fors. Compos: A dsl for composing iot systems with weak connectivity. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, REBLS 2023, p. 31–42, New York, NY, USA, 2023. Association for Computing Machinery.

[5] Hilal Kevser Cora, H. Turgut Uyar, and A. Şima Etaner-Uyar. Hh-dsl: a domain specific language for selection hyper-heuristics. In *Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '13 Companion, p. 1317–1324, New York, NY, USA, 2013. Association for Computing Machinery.

[6] Martin Fowler. Language workbenches: The killer-app for domain specific languages? 2005.

[7] Lennart C.L. Kats and Eelco Visser. The spoofax language workbench: rules for declarative specification of languages and ides. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, p. 444–463, New York, NY, USA, 2010. Association for Computing Machinery.

[8] Spoofax Team. Spoofax - Spoofax: The Language Designer's Workbench — spoofax.dev. `https://spoofax.dev`. [Accessed 08-01-2025].

[9] Tijs van Der Storm. The Rascal Language Workbench. Research report, 2011.

[10] GitHub Copilot overview — code.visualstudio.com. `https://code.visualstudio.`

com/docs/copilot/overview. [Accessed 07-01-2025].

[11] Cursor - The AI Code Editor — cursor.com. https://www.cursor.com. [Accessed 07-01-2025].

[12] Apple. Apple Intelligence. https://developer.apple.com/jp/apple-intelligence/. [Accessed 07-01-2025].

[13] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. Large language models for software engineering: Survey and open problems, 2023.

[14] Sathvik Joel, Jie JW Wu, and Fatemeh H. Fard. A survey on llm-based code generation for low-resource and domain-specific programming languages, 2024.

[15] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer

McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. Gpt-4 technical report, 2024.

[16] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[17] Kaiyan Chang, Kun Wang, Nan Yang, Ying Wang, Dantong Jin, Wenlong Zhu, Zhirong Chen, Cangyuan Li, Hao Yan, Yunhao Zhou, Zhuoliang Zhao, Yuan Cheng, Yudong Pan, Yiqi Liu, Mengdi Wang, Shengwen Liang, Yinhe Han, Huawei Li, and Xiaowei Li. Data is all you need: Finetuning llms for chip design via an automated design-data augmentation framework. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, p. 1–6. ACM, June 2024.

[18] Ayush Agrawal, Siddhartha Gadgil, Navin Goyal, Ashvni Narayanan, and Anand Tadipatri. Towards a mathematics formalisation assistant using large language models, 2022.

[19] Luming Lu, Jiyuan An, Yujie Wang, Liner yang, Cunliang Kong, Zhenghao Liu, Shuo Wang, Haozhe Lin, Mingwei Fang, Yaping Huang, and Erhong Yang. From text to cql: Bridging natural language and corpus search engine, 2024.

[20] Pietro Liguori, Christian Marescalco, Roberto Natella, Vittorio Orbinato, and Luciano Pianese. The power of words: Generating powershell attacks from natural language, 2024.

[21] Mohamad Fakih, Rahul Dharmaji, Yasamin Moghaddas, Gustavo Quiros, Oluwatosin Ogundare, and Mohammad Abdullah Al Faruque. Llm4plc: Harnessing large language models for verifiable programming of plcs in industrial control systems. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, p. 192–203. ACM, April 2024.

[22] Peng Di, Jianguo Li, Hang Yu, Wei Jiang, Wenting Cai, Yang Cao, Chaoyu Chen, Dajun Chen, Hongwei Chen, Liang Chen, Gang Fan, Jie Gong, Zi Gong, Wen Hu, Tingting Guo, Zhichao Lei, Ting Li, Zheng Li, Ming Liang, Cong Liao, Bingchang Liu, Jiachen Liu, Zhiwei Liu, Shaojun Lu, Min Shen, Guangpei Wang, Huan Wang, Zhi Wang, Zhaogui Xu, Jiawei Yang, Qing Ye, Gehao Zhang, Yu Zhang, Zelin Zhao, Xunjin Zheng, Hailian Zhou, Lifu Zhu, and Xianying Zhu. Codefuse-13b: A pretrained multi-lingual code large language model. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, p. 418–429. ACM, April 2024.

[23] Mingzhe Gao, Jieru Zhao, Zhe Lin, Wenchao Ding, Xiaofeng Hou, Yu Feng, Chao Li, and Minyi Guo. Autovcoder: A systematic framework for automated verilog code generation using llms, 2024.

[24] Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. Evaluating llms for hardware design and test. In *2024 IEEE LLM Aided Design Workshop (LAD)*, p. 1–6. IEEE, June 2024.

[25] Houxing Ren, Mingjie Zhan, Zhongyuan Wu, Aojun Zhou, Junting Pan, and Hongsheng Li. Reflectioncoder: Learning from reflection sequence for enhanced one-off code generation, 2024.

[26] Paola Vitolo, George Psaltakis, Michael Tomlinson, Gian Domenico Licciardo, and Andreas G. Andreou. Natural language to verilog: Design of a recurrent spiking neural network using large language models and chatgpt. In *2024 International Conference on Neuromorphic Systems (ICONS)*, p. 110–116. IEEE, July 2024.

[27] Jianan Yao, Ziqiao Zhou, Weiteng Chen, and Weidong Cui. Leveraging large language models for automated proof synthesis in rust, 2023.

[28] Shuyan Zhou, Uri Alon, Frank F. Xu, Zhiruo Wang, Zhengbao Jiang, and Graham Neubig. Docprompting: Generating code by retrieving the docs, 2023.

[29] Saikat Chakraborty, Gabriel Ebner, Siddharth Bhat, Sarah Fakhoury, Sakina Fatima, Shuvendu Lahiri, and Nikhil Swamy. Towards neural synthesis for smt-assisted proof-oriented programming, 2024.

[30] Sameer Pimparkhede, Mehant Kammakomati, Srikanth Tamilselvam, Prince Kumar, Ashok Pon Kumar, and Pushpak Bhattacharyya. Doccgen: Document-based controlled code generation, 2024.

[31] Shang Liu, Wenji Fang, Yao Lu, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution. In *2024 IEEE LLM Aided Design Workshop (LAD)*, pp. 1–5, 2024.

[32] Lakshya A Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K. Lahiri, and Sriram K. Rajamani. Guiding language models of code with global context using monitors, 2023.

[33] Matthew DeLorenzo, Animesh Basak Chowdhury, Vasudev Gohil, Shailja Thakur, Ramesh Karri, Siddharth Garg, and Jeyavijayan Rajendran. Make every move count: Llm-based high-quality rtl code generation using mcts, 2024.

[34] Wamiq Reyaz Para, Shariq Farooq Bhat, Paul Guerrero, Tom Kelly, Niloy Mitra, Leonidas Guibas, and Peter Wonka. Sketchgen: Generating constrained cad sketches, 2021.

[35] Akifa Nabil Ufairah and Yani Widyani. Experiment on utilizing gpt-3.5 language model to generate dsl of data management for web application development using qore-base platform. In *2023 IEEE International Conference on Data and Software Engineering (ICoDSE)*, pp. 108–113, 2023.

[36] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[37] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

[38] Kai Lv, Yuqing Yang, Tengxiao Liu, Qinghui Gao, Qipeng Guo, and Xipeng Qiu. Full parameter fine-tuning for large language models with limited resources, 2024.

[39] Peng Ye, Yongqi Huang, Chongjun Tu, Minglei Li, Tao Chen, Tong He, and Wanli Ouyang. Partial fine-tuning: A successor to full fine-tuning for vision transformers, 2023.

[40] Zhiqiang Hu, Lei Wang, Yihuai Lan, Wanyu Xu, Ee-Peng Lim, Lidong Bing, Xing Xu, Soujanya Poria, and Roy Ka-Wei Lee. Llm-adapters: An adapter family for parameter-efficient fine-tuning of large language models, 2023.

[41] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language

models, 2021.

[42] Lingling Xu, Haoran Xie, Si-Zhao Joe Qin, Xiaohui Tao, and Fu Lee Wang. Parameter-efficient fine-tuning methods for pretrained language models: A critical review and assessment, 2023.

[43] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024.

[44] Yongda Yu, Guoping Rong, Haifeng Shen, He Zhang, Dong Shao, Min Wang, Zhao Wei, Yong Xu, and Juhong Wang. Fine-tuning large language models to improve accuracy and comprehensibility of automated code review. *ACM Trans. Softw. Eng. Methodol.*, Vol. 34, No. 1, December 2024.

[45] Justin Zhao, Timothy Wang, Wael Abid, Geoffrey Angus, Arnav Garg, Jeffery Kinnison, Alex Sherstinsky, Piero Molino, Travis Addair, and Devvret Rishi. Lora land: 310 fine-tuned llms that rival gpt-4, a technical report, 2024.

[46] Andrew Kyle Lampinen, Stephanie C. Y. Chan, Aaditya K. Singh, and Murray Shanahan. The broader spectrum of in-context learning, 2024.

[47] Rishabh Agarwal, Avi Singh, Lei M. Zhang, Bernd Bohnet, Luis Rosias, Stephanie Chan, Biao Zhang, Ankesh Anand, Zaheer Abbas, Azade Nova, John D. Co-Reyes, Eric Chu, Feryal Behbahani, Aleksandra Faust, and Hugo Larochelle. Many-shot in-context learning, 2024.

[48] Junjielong Xu, Ziang Cui, Yuan Zhao, Xu Zhang, Shilin He, Pinjia He, Liqun Li, Yu Kang, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. Unilog: Automatic logging via llm and in-context learning. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[49] Zhi Rui Tam, Cheng-Kuang Wu, Yi-Lin Tsai, Chieh-Yen Lin, Hung yi Lee, and Yun-Nung Chen. Let me speak freely? a study on the impact of format restrictions on performance of large language models, 2024.

[50] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. Fact: a dsl for timing-sensitive computation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, p. 174–189, New York, NY, USA, 2019. Association for Computing Machinery.

[51] Russell O'Connor. Simplicity: A new language for blockchains. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, PLAS '17, p.

107–120, New York, NY, USA, 2017. Association for Computing Machinery.

[52] Sandra Macià, Sergi Mateo, Pedro J. Martínez-Ferrer, Vicenç Beltran, Daniel Mira, and Eduard Ayguadé. Saiph: Towards a dsl for high-performance computational fluid dynamics. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, RWDSL2018, New York, NY, USA, 2018. Association for Computing Machinery.

[53] Ameni Ben Fadhel, Domenico Bianculli, and Lionel Briand. Gemrbac-dsl: A high-level specification language for role-based access control policies. In *Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies*, SACMAT '16, p. 179–190, New York, NY, USA, 2016. Association for Computing Machinery.

[54] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: a high-performance graph dsl. *Proc. ACM Program. Lang.*, Vol. 2, No. OOPSLA, October 2018.

[55] Anders Miltner, Devon Loehr, Arnold Mong, Kathleen Fisher, and David Walker. Saggitarius: A dsl for specifying grammatical domains. *Proc. ACM Program. Lang.*, Vol. 7, No. OOPSLA2, October 2023.

[56] Humberto Rodriguez Avila, Joeri De Koster, and Wolfgang De Meuter. Sparrow: a dsl for coordinating large groups of heterogeneous actors. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE 2017, p. 31–40, New York, NY, USA, 2017. Association for Computing Machinery.

[57] Yong Li, Shoaib Kamil, Keenan Crane, Alec Jacobson, and Yotam Gingold. Imesh: A dsl for mesh processing. *ACM Trans. Graph.*, Vol. 43, No. 5, June 2024.

# 謝辞

本研究を進めるにあたり、指導教員である東京大学 情報理工学系研究科 創造情報学専攻 千葉滋研究室の千葉 滋教授には、週にわたる定期的なミーティングを通じて、研究の方向性から実験の設計に至るまで丁寧かつ行き届いたご指導を賜りました。

また、本研究の進捗に際し、本論の方向性の検討や実験手法の提案などにおいて多大な助言をいただきました東京大学 情報理工学系研究科 創造情報学専攻の山崎 徹郎助教にも、厚く御礼申し上げます。

これらのご指導・ご支援がなければ、本研究の成果は得られなかったと実感しております。改めまして、心より感謝申し上げます。