大規模言語モデルを用いたプログラミングを支援する言語機構

奥田 勝己　Saman Amarasinghe

本論文では，大規模言語モデル (LLM) を用いたプログラムの作成を支援する言語拡張を提案する．創発的能力を有する LLM は様々なタスクへの応用が期待される．しかし，LLM を用いたプログラムを作成するための統一的なプログラミングインタフェースは存在しない．このため，ユーザはタスクごとに LLM へのプロンプトを自然言語で記述し，自然言語で返される応答を構文解析する必要があった．そこで本論文では，ユーザによるタスクごとのプロンプトエンジニアリングの一部や構文解析を不要化するためのプログラミングインタフェースを提案する．本プログラミングインタフェースでは,（1）型を用いた出力制御,（2）プロンプトテンプレートによる関数の定義,（3）LLM を用いたコード生成をサポートする．既存プロンプトを提案インタフェースで置き換えた実験では自然言語による出力フォーマットの指定を型指定に置き換えることで 16.1%削減できることを確認した.

1　はじめに

大規模言語モデル（Large Language Model: LLM）はモデルのスケールアップに伴って様々な能力を発現することが知られている [5]．これらの能力には算術演算，質問応答，テキストの要約，言語の翻訳，コード生成，創造的なテキスト作成などの多岐にわたるタスクが含まれる．また，これらは，明示的に訓練されたものではなく，訓練フェーズ中の大量の自然言語データから自動的に獲得されたものである．この現象は「創発的能力」として知られ，LLM の特徴的なものとなっている．

このような自動的に獲得される創発的能力の登場は，ソフトウェア開発の新しいパラダイムをもたらす可能性がある．ソフトウェア開発者は，質問応答，テキストの要約，言語の翻訳などのタスクを実行するために，LLM をアプリケーションの一部として使用することができる．また，ソフトウェア開発者は，LLM に代わりにコーディングを行わせることもできる．LLM を用いたツールである Jigsaw [3] や Codex/Copilot [2] は，自然言語の記述からコードを生成することができる．また，専用ツールを用いる以外にもソフトウェア開発者は ChatGPT，BingAI，Bard などの汎用的な LLM のチャットモデルを使用して，自然言語を用いてコードを生成することができる．

しかし，現状では LLM を用いた統一的なプログラミングインタフェースは存在しない．たとえば，開発者は，プログラミングタスクが直接 LLM を用いて実行すべきタスクか，LLM を用いてコード生成すべきタスクかによって異なる手順を踏む必要がある．また，LLM を用いる際には，アプリケーションごとにテーラーメイドのプロンプトを作成し，LLM の応答を抽出して処理する必要がある．たとえば，アプリケーションが必要とするデータフォーマットを指定することや，LLM の応答から必要な情報を抽出するための構文解析が必要となる．さらに，LLM の性能を引き出すためには，LLM の特徴を理解し，適切なプロンプトを作成する必要がある．たとえば，単純に答えを要求するだけでなく，その理由を説明させたり，step-by-step で応答を返させたりすることで，より良

* This is an unrefereed paper. Copyrights belong to the Authors.

Katsumi Okuda, マサチューセッツ工科大学 / 三菱電機株式会社, Massachusetts Institute of Technology / Mitsubishi Electric Corporation.

Saman Amarasinghe, マサチューセッツ工科大学, Massachusetts Institute of Technology.

い応答を得ることができる．

本論文では，これらの課題を解決するための統一的なプログラミングインタフェースを提案する．提案するインタフェースは，(1) 型を用いた出力制御，(2) プロンプトテンプレートによる関数の定義，(3) LLM を用いたコード生成をサポートする．型を用いた出力制御では，ユーザは自然言語で出力フォーマットを指定する代わりに，習熟したプログラミング言語の型を用いて出力フォーマットを指定することができる．プロンプトテンプレートによる関数の定義では，プロンプトから関数を生成することができる．関数の戻り値は，指定した型となるため，ユーザ自身による構文解析を不要化できる．また，LLM を用いたコード生成では，定義した関数を LLM に直接実行させる代わりに，LLM にコード生成されることができる．この時 LLM が直接実行する場合であっても，LLM にコード生成させる場合でもプログラミングインタフェースは共通である．TypeScript 上に実装した提案プログラミングインタフェースを用いた実験では，50 個の既存のオープンソースのプロンプトから出力フォーマットの指定を削除し，型を用いた出力制御に置き換えた結果，プロンプトの長さを 16.1% 短縮できることを確認した．

2 プログラミングにおける LLM の利用方法

LLM を用いたプログラミングにおける課題を明らかにするため，本章では LLM を使用した 2 つのシナリオを検討し，LLM を用いたタスクの分類を行う．最初のシナリオはテキストで与えられた商品レビューの感情分析を行う例である．感情分析は従来自然言語処理のパイプラインや専用の機械学習モデルで対応していたタスクである．しかし，適切なプロンプトを提供することで，GPT-4 のような汎用の LLM を使用し，与えられたレビューの背後の感情を解釈することが可能である．

2 つ目のシナリオでは感情分析の結果をファイルに格納するケースを考える．従来は人手でファイルにアクセスするプログラムを書いていた．しかし，プログラムの仕様を LLM に与えることで，人手に代わって LLM がファイルにアクセスするためのコードを生成することができる．

2.1 アプリケーションの一部としての LLM

例として，LLM を用いて商品レビューの感情分析をする場合を考える．この場合のプログラムは以下の疑似コードで表すことができる．なお，# から始まる部分はコメントである．

```
1 review = "この商品は使いやすくてとても気に入っています．"
2 prompt = "What is the sentiment of this review: " + review
3 prompt += "\nFinal answer should be enclosed with [ and ] like [negative]."
4 response = LLM.predict(prompt) # response: "The sentiment of the review is positive. [positive]"
5 sentiment = parse_sentiment(response)  # sentiment: "positive"
```

1 行目は商品のレビューである．この例ではハードコードされているが，現実的にはレビューはデータベースや他のデータソースから取得される． 2 行目は，LLM へのプロンプトである．プロンプトでは，レビューの感情を予測するために LLM に与えられる情報を指定する．この例では，レビューの感情を予測するために，レビューのテキストをプロンプトに埋め込んでいる．3 行目は，LLM の応答を生成するために最終結果を [と] で囲むように指示している．このような応答フォーマットの指定は，結果の抽出を容易にするための定石である．4 行目では，LLM に対してプロンプトを与えている．5 行目では，LLM の応答から感情を抽出している．開発者は LLM の応答から [と] で囲まれた部分を抽出することで，レビューの感情を取得することができる．

4 行目と 5 行目は，LLM の用途ごとにプロンプトエンジニアリングが必要となる部分である．LLM の応答が異なれば出力フォーマットの指定や結果の抽出方法も異なるため，プロンプトエンジニアリングは LLM の用途ごとに必要となる．感情分析では出力が単一の単語（positive や negative）のような簡単なものであったが，タスクによっては複数の数値や単語のリストなどの複雑なものもある．また，問題がさらに複雑である場合，LLM から良好な応答を得

るためには，開発者は Chain of Thought (CoT) [4] や Few-shot Learning [1] などのテクニックを利用する必要がある．たとえば，CoT を利用するためには，プロンプトに"Let's think step-by-step"のような文言を追加する必要がある．これらのテクニックを利用するためには，開発者は LLM の機能を理解し，プロンプトを適切に設計する必要がある．

2.2　コード生成器としての LLM

前節の感情分析タスクに続けて，感情分析の結果をローカルの CSV ファイルに書き込むタスクを考える．開発者は Copilot などの専用ツールを用いることもできるが，開発者は ChatGPT，Bard，BingAI のような汎用的なチャット用の LLM を利用してこのタスクのコードを生成することができる．ユーザは LLM に自然言語による使用記述を与えることでコードを生成する．本シナリオでは，開発者は ChatGPT に，感情分析の結果をローカルファイルに保存する Python コードを生成するように求める．コード生成プロセスは，対話的な画面で行われる．たとえば，図 1 は ChatGPT にコーディングを指示している様子である．本シナリオでは，ChatGPT に CSV ファイルの末尾にレビューと感情を追加するコードを生成するように求めている．また，ChatGPT による応答画面には，正しくコード断片が実装されている．このコードはローカルのファイルシステムにアクセスする必要があるため，LLM によって直接実行することはできない．開発者は，ChatGPT によって生成されたコード断片をコピーして，自身のソフトウェア開発環境に貼り付ける必要がある．

このコード生成プロセスは，開発者がコードを生成するために LLM を使用する場合には一般的である．しかし，このプロセスは，開発者が生成されたコードを自身のソフトウェア開発環境に貼り付ける必要がある点が不便である．我々が提案するプログラミングインタフェースでは，統一的なプログラミングインタフェースから本節で紹介した 2 種類のタスクに対応することができる．

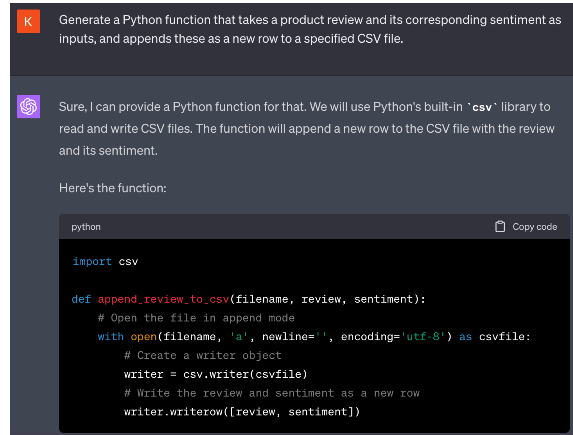

図 1: LLM （ChatGPT）を用いたコード生成の例

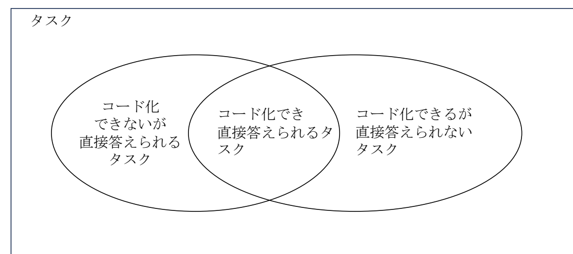

図 2: Classification of Tasks

2.3　LLM を用いたタスクの分類

前節で示した 2 つのシナリオは，LLM を用いたプログラミングの 2 つの利用法を示している．1 つはアプリケーションの一部として LLM を直接用いる方法であり，他方は LLM にコードを生成させる方法である．すなわち，タスクには LLM を直接用いることができるものと，コード生成を用いることができるものがある．また，どちらの方法でも対応できるタスクも存在する．このような利用法を元に，LLM を用いたタスクは，以下の 3 つのカテゴリに分類することができる．

コード化できないが直接答えられるタスク

これらのタスクは自然言語の理解と生成に関わり，LLM で直接対応できるが，コーディングには適していない．文脈，セマンティクス，言語のニュアンスを理解する必要があり，これらの理解のための処理をコーディングするためのアルゴリズムやルールは存

在しない．このようなタスクの例には，テキストの要約，感情分析，言語翻訳，抽象的な概念の説明，文脈が与えられた後のテキストの予測，言い換えが含まれる．

交差タスク - コード化でき直接答えられる

これらのタスクはLLMと伝統的なプログラミング実装の両方で扱うことができる．これらはしばしば決定論的であり，ルールに基づいているため，LLMとコード化されたソリューションの両方に適している．たとえば，数学の問題は数学のルールに従って解決でき，文字列の操作には事前定義された操作が含まれる．また，正規表現のマッチングもルールに基づいており，ソートや検索の問題は明確に定義されたアルゴリズムを使用して対応できる．LLMを直接用いるのかコーディングを行うかの選択は，特定の文脈や要件に依存する．

コード化できるが直接答えられないタスク

これらのタスクはコーディングに適しているが，追加のリソースやデータへのアクセスが必要であるため，LLMでは直接解決できない．これらのタスクは，外部のシステムとの相互作用を必要とするか，リアルタイムやユーザー固有のデータに依存することが多い．このようなタスクの例として，Webからのリアルタイムデータの取得，特定のシステムに格納されたユーザー固有のデータの操作，データベースとのデータの取得，挿入，更新，または削除のための相互作用，特定の計算リソースやライブラリが必要な高度な計算やアルゴリズムが含まれる．LLMはこれらのタスクの指示やコードを提供できるが，それらを実行するためにはプログラミング環境や計算資源が必要である．

従来，LLMを用いたプログラミングでは，ユーザは暗黙的にこれらのタスクの分類を行い，適切なプロンプトを作成する必要があった．また，交差タスクについてはLLMに直接実行させるのかコード生成させるのかの選択を行う必要もあった．交差タスクの境界は曖昧であり，現状ではコード化できないタスクであっても将来的にコード化できる可能性がある．前節で見たように両者では扱いが異なるため，一方から他方に移行する場合，はじめからやり直しになるかもしれない．このため，これらを違いを意識せず統一的に扱うことができるプログラミングインタフェースが必要である．

3　提案インタフェース

3.1　ユースケース

本章ではLLMを利用するための統一的なプログラミングインタフェースを提案する．提案インタフェースは，直接答えられるがコード化できないタスク，交差タスク，コード化できるが直接答えられないタスクなど，多岐にわたるタスクに対応することができる．本インタフェースでは，LLMの応答を型で指定し，プロンプトをテンプレートで指定することで，LLMを用いたタスクを実行することができる．また，同様のインタフェースでコード生成を行うことも可能である．

コード化できないが直接答えられるタスクの一例として，レビューの感情を取得するタスクがある．提案インタフェースによる型を指定を用いるとこのタスクは以下のようにプログラミングすることができる．

```
1 let sentiment = llm<"positive"|"negative">(’What
      is the sentiment of ”私はこの商品が気に入り
     ました.”?’);
```

ここで，`llm`は提案インタフェースの1つであり，プロンプトを引数とする関数である．また，<と>で囲まれた部分は戻り値の型を指定するための型パラメータである．`llm`の型パラメータには，LLMの応答の型を指定することができる．ここでは，文字列リテラル型の”positive”または”negative”を戻り値として指定している．このように型パラメータを用いることで，LLMの応答の型を指定することができる．このため，プロンプト内では出力のフォーマットを指定する必要はない．たとえば，”Please answer with [positive] or [negative]”のような文言をプロンプトに埋め込む必要はない．

LLMの引数であるプロンプトには一部をパラメータ化したプロンプトテンプレートを指定することもできる．プロンプトテンプレートではパラメータを{{と}}で囲めば良い．たとえば，レビューの感情を取

得するタスクでは，レビューのテキストをプロンプトに埋め込む必要があるが，これはパラメータを用いることで実現できる．

```
let sentiment = llm<"positive"|"negative">('What is the sentiment of {{review}}?');
```

先述の例で具体的なレビューのテキストが本例ではreviewで置き換わっている．

さらにテンプレートプロンプトと型パラメータを用いて以下のように関数を定義することも可能である．

```
let getSentiment = define<"positive"|"negative">('What is the sentiment of {{review}}?')
```

提案インタフェースの一部であるdefineは引数のテンプレートプロンプトから関数を生成する．テンプレートのパラメータは自動的に関数のパラメータに変換される．この例ではgetSentimentを定義しており，これは以下のように呼び出すことができる．

```
let sentiment = getSentiment({review: "私はこの商品が気に入りました．"});
```

コード化できるが直接答えられないタスクも同様に扱うことができる．以下は，レビューの感情を取得するタスクをコード化した例である．

```
llm<string[]>('ファイル名が{{filename}}のファイルから{{regex}}と一致するすべての行を選択せよ');
```

ここで，regexは正規表現型の変数で，filenameは文字列変数である．また，llmに続く<string[]>は戻り値の方として文字列型の配列を指定している．このコードを用いると簡単な grep コマンドを実装することができる．grep コマンドは，ローカルファイルシステムへのアクセスが必要なため，LLM が実行することができないタスクである．それでも，同じインタフェースを用いて以下のように実装することが可能である．

```
const filename = sys.argv[2];
const regex = new ReGex(sys.argv[3]);
for (let line of llm<string[]>('ファイル名が{{filename}}のファイルから{{regex}}と一致するすべての行を選択せよ')) {
  console.log(line);
}
```

ここで，filenameとregexはコマンドラインの引数である．このコードは，ファイルfilename内の正規表現regexと一致するすべての行を出力する．llmに続く<string[]>は，結果の型が文字列の配列であることを示しており，for...ofループで使用することができる．

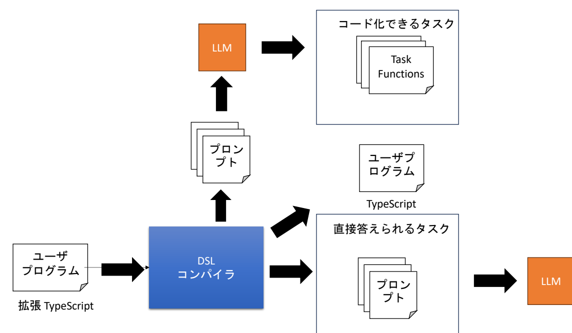

図 3: 提案インタフェースをサポートする DSL コンパイラの概要

3.2 提案インタフェースの実装

本節では，提案インタフェースを実現する DSL コンパイラについて説明する．本 DSL コンパイラは，llmとdefineを TypeScript の構文として解釈し，戻り値の型パラメータを用いて LLM 向けのプロンプトを作成する．また，コード化できるタスクについては，自動的に LLM を用いてコード生成を行う．コード生成を行なった結果，llmの呼び出しは，実行時には LLM の呼び出しではなく，生成された TypeScript 関数への呼び出しとなる．

図 3 に提案インタフェースを含むプログラムのコンパイルの概要を示す．DSL コンパイラは，TypeScript コンパイラプラグインとして実装され，TypeScript コンパイラがソースコードをコンパイルするときに実行される．TypeScript コンパイラは，提案インタフェースで拡張された TypeScript で書かれたソースコードを解析し，ソースコードの抽象構文木（AST）を生成する．DSL コンパイラは AST をトラバースし，llmおよびdefineの呼び出しを変換する．この時，llmに対しては，与えられたプロンプトを元にコード生成を行い，生成されたコードに置換する．コード生成できない場合，LLM の呼び出し関数への

置換を行う．コード生成を行う場合も直接 LLM を用いる場合も，`llm`および `define`の型パラメータは，LLM の応答の型を指定するためにプロンプトに自動的に反映される．また，これらの型は LLM からの応答を解析するためにも用いられる．

4 実験結果

型による出力制御の効果を確認するため，既存のプロンプトを提案インタフェースを用いたプロンプトに変換し，元のプロンプトと提案インタフェースのプロンプトの長さを比較した．これらのプロンプトは OpenAI Evals †1 から取得したものである．OpenAI Evals リポジトリには， LLM の実世界の使用例を表す 300 以上のベンチマークが含まれる．

リポジトリ内の各ベンチマークは，複数のテストケースから構成される．各テストケースは，プロンプトと期待する応答を含む．本実験では，OpenAI Evals の最初の 50 個のベンチマークに焦点を絞った．また，特定のベンチマーク内のすべてのテストケースが似た型を持つが，入力が異なるため，各ベンチマークからの最初のテストケースのみを実験に用いた．

プロンプトの修正過程では，提案インタフェースでは不要となる情報を削除した．これには，LLM の応答形式を指示するフレーズや，LLM にその答えの根拠を明らかにするよう促すフレーズが含まれる．たとえば，ベンチマーク *2d_movement.dev.v0* の元のプロンプトには以下のような英語による指示が含まれる．

> Please note: In the following EXERCISE, it is essential that you only respond with a single line in the format (x, y).

このような指示は LLM の出力の型で置き換えることで不要となる．この例では，出力の型として `{ x: int, y: int }`を指定した．

なお，OpenAI Evals には現状の GPT-3.5 や GPT-4 が解決できないベンチマークを中心に登録されているため，修正されたプロンプトがテストケースで詳述されている LLM の期待される応答と出力の形式が一致していることのみを確認し，LLM の応答が正しいかどうかは確認していない．

†1 https://github.com/openai/evals

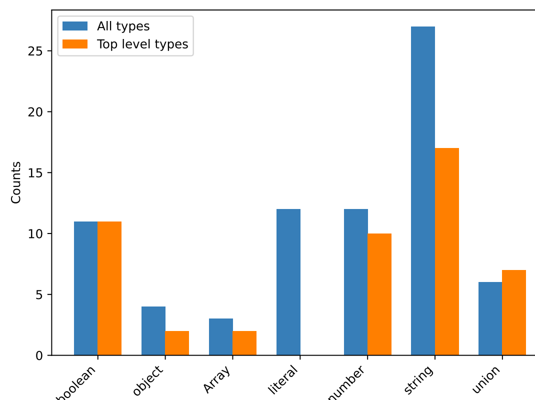

図 4: プロンプト書き換えにおける型ごとの使用回数

表 1 に実験結果の詳細を示す．最初の列はベンチマークの名前を示し，2 番目の列は戻り値の型を示す．3 番目の列は元のプロンプトの長さであり，4 番目は書き換え後のプロンプトの長さである．元のプロンプトからの文字数の平均削減量は 16.1%である．

なお，2 番目の列にはベンチマークの戻り値のトップレベルの型のみを示した．たとえば，戻り値の型が `Array(int)`であれば，テーブル内で単に `Array`として表示される．図 4 に既存プロンプトの書き換えに用いたすべての型をまとめた．特にリテラル型は表 1 には含まれていなかった型である．これは，リテラル型をユニオン型の一部としてのみ使用したためである．たとえば，四択問題を解決するベンチマークでは，選択肢はリテラル型のユニオンとして詳述されており，そのようなベンチマークの戻り値型は `"A" | "B" | "C" | "D"`として指定した．

5 まとめと今後の課題

本論文では，LLM を用いたプログラミングをサポートするためのプログラミングインタフェースについて提案した．提案インタフェースでは，型を用いた出力制御，プロンプトテンプレートを用いた関数定義，プロンプトからのコード生成をサポートする．既存プロンプトの書き換えによる実験で

表 1: プロンプトの書き換えによる文字数の削減効果

ベンチマーク	戻り値の型	元のプロンプト長	削減後のプロンプト長	削減文字数	削減率
2d_movement.dev.v0	object	506	387	119	23.52
3d_globe_movement.dev.v0	string	633	549	84	13.27
Unfamiliar-Chinese-Character.dev.v0	object	136	136	0	0.00
aba_mrpc_true_false.dev.v0	boolean	233	222	11	4.72
abstract-causal-reasoning-symbolic.dev.v0	union	383	270	113	29.50
abstract2title.test.v1	union	1531	1523	8	0.52
actors-sequence.dev.match-v1	Array	403	304	99	24.57
adultery_state_laws.dev.v0	Array	193	142	51	26.42
afrikaans-lexicon.dev.v0	boolean	130	90	40	30.77
aime_evaluation.dev.v0	number	208	152	56	26.92
algebra-word-problems.s1.simple-v0	number	211	136	75	35.55
allergen-information.dev.v0	union	246	246	0	0.00
alternate_numeral_systems.dev.v0	string	411	211	200	48.66
ambiguous-sentences.dev.v0	string	134	134	0	0.00
anagrams.test.v1	string	74	74	0	0.00
arc.dev.v0	string	1070	1070	0	0.00
arithmetic-expression-meta.dev.v0	string	601	441	160	26.62
arithmetical_puzzles.dev.v0	number	992	850	142	14.31
ascii-digit-recognition.dev.v0	number	496	409	87	17.54
ascii-wordart.dev.v0	string	815	807	8	0.98
asl-classifiers.dev.v0	boolean	403	242	161	39.95
atpl_exams.dev.v0	union	693	625	68	9.81
automata-and-complexity.dev.v0	boolean	283	128	155	54.77
backgammon-illegal-move.dev.v0	boolean	743	621	122	16.42
balance-chemical-equation.dev.v0	union	296	218	78	26.35
base64-decode-simple.dev.v0	string	377	326	51	13.53
beam.analysis.dev.v0	float	234	184	50	21.37
belarusian-grammar.dev.v0	boolean	165	150	15	9.09
belarusian-lexicon.dev.v0	boolean	108	93	15	13.89
belarusian-numerals.dev.v0	number	246	179	67	27.24
belarusian-orthography.dev.v0	string	218	218	0	0.00
belarusian-proverbs.dev.v0	string	240	145	95	39.58
belarusian-rhyme.dev.v0	union	180	159	21	11.67
belarusian-russian-translation.dev.v0	string	347	347	0	0.00
belarusian-syllable-count.dev.v0	number	169	127	42	24.85
belarusian-synonyms.dev.v0	boolean	183	168	15	8.20
benjaminmoore_to_hex.dev.v0	string	66	66	0	0.00
best.dev.v0	string	35	35	0	0.00
bigrams.dev.v0	number	300	300	0	0.00
bitwise.dev.v0	string	502	502	0	0.00
blackfoot-numerals-modern.dev.v0	number	707	305	402	56.86
body-movement.dev.zero_shot_v0	union	510	465	45	8.82
born-first.dev.v0	boolean	91	76	15	16.48
brazilian-lexicon.dev.v0	boolean	138	98	40	28.99
brazilian_laws.test.v1	string	1068	1068	0	0.00
building_floorplan.test.v1	number	960	870	90	9.38
bulgarian-lexicon.dev.v0	boolean	134	94	40	29.85
canto_wu_pronunciation.dev.v0	string	301	301	0	0.00
canto_wu_pronunciation_fewshot.dev.v0	string	445	445	0	0.00

は，プロンプトの長さを 16.1%削減することができた．これは，自然言語による出力フォーマットの指定部分に置き換えることができるためである．今後，実験を拡張し，コード生成による性能向上などを検証する予定である．なお，本論文で提案した言語機構の Python と TypeScript による実装は，それぞれ `https://github.com/katsumiok/pyaskit` と `https://github.com/katsumiok/ts-askit` でオープンソースとして公開している．

参考文献

[1] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Amodei, D., and Mordatch, T.: Language Models are Few-Shot Learners, *arXiv preprint arXiv:2005.14165*, (2020).

[2] Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W.: Evaluating Large Language Models Trained on Code, 2021.

[3] Jain, N., Vaidyanath, S., Iyer, A., Natarajan, N., Parthasarathy, S., Rajamani, S., and Sharma, R.: Jigsaw: Large Language Models Meet Program Synthesis, *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, New York, NY, USA, Association for Computing Machinery, 2022, pp. 1219–1231.

[4] Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., and Iwasawa, Y.: Large Language Models are Zero-Shot Reasoners, *Advances in Neural Information Processing Systems*, Vol. 35, 2022, pp. 22199–22213.

[5] Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., Yogatama, D., Bosma, M., Zhou, D., Metzler, D., Chi, E. H., Hashimoto, T., Vinyals, O., Liang, P., Dean, J., and Fedus, W.: Emergent Abilities of Large Language Models, 2022.