Master's Thesis

A Study on Improving LLM-based Code Completion Using LR Parsing

Department of Artificial Intelligence Convergence

Graduate School, Chonnam National University

ATIQUE, MD MONIR AHAMMOD BIN

August 2025

A Study on Improving LLM-based Code Completion Using LR Parsing

Department of Artificial Intelligence Convergence

Graduate School, Chonnam National University

ATIQUE, MD MONIR AHAMMOD BIN

Supervised by Professor Choi, Kwanghoon

A dissertation submitted in partial fulfillment of the requirements for the **Master of Science in Artificial Intelligence Convergence**

August 2025

Content

List of Figures

List of Tables

A Study on Improving LLM-based Code Completion Using LR Parsing

ATIQUE MD MONIR AHAMMOD BIN

Department of Artificial Intelligence Convergence

Graduate School Chonnam National University

(Supervised by Professor Kwanghoon Choi)

Abstract

Code completion is a crucial feature in modern IDEs, improving programming efficiency. Traditional systems rely on prefix filtering and static ranking but often overwhelm users with lengthy, alphabetically sorted lists. Recent research has introduced LR-parsing-based approaches that derive completion candidates from language syntax and compute their ranks using open source programs; however, these methods only suggest structural candidates, requiring manual refinement into complete code. To address this, we propose a hybrid method that integrates LR parsing with LLMs to enhance accuracy and usability. Our approach refines structural candidates using LR parsing into textual code suggestions via an LLM, referencing a database of ranked candidates from open-source programs. This combines the syntactic precision of LR parsing with the generative capabilities of LLMs. This study examines whether LLMs benefit from LR structural candidates in code completion. By comparing completions with and without these candidates, we assess their impact. Building on prior research, we also explore how leveraging top-ranked structural candidates can effectively enhance LLM-based code completion precision. We also demonstrate our method through VSCode extensions for Microsoft Small Basic and C. As a language-agnostic solution, our system applies to any language with a defined LR grammar. Our findings suggest that integrating LR parsing with LLM-based completion improves both accuracy and usability, paving the way for more effective code completion in modern IDEs. This study also investigates whether incorporating LR grammar-based structural candidates explicitly into LLM-driven code completion improves performance. Experiments using SacreBLEU and SequenceMatcher on Microsoft Small Basic and C show no significant (accuracy) gain, suggesting LLMs may already internalize grammar or need alternative evaluation. However, despite the progress of LLMs, the impact of LLM selection on LR-parsed based syntax-aware code generation remains underexplored. In this study, we also conduct a comparative analysis of the performance and impact of two prominent LLMs, ChatGPT 3.5 and Llama 3, within an LR parsing-based code completion framework. Our experiments, evaluated using SacreBLEU and SequenceMatcher accuracy metrics, reveal that ChatGPT 3.5 achieves higher accuracy than Llama 3, underscoring the importance of selecting an appropriate LLM for enhanced code completion. These findings highlight the role of model selection in LLMbased code completion using LR parsing. Future research could extend this comparative analysis to a broader range of LLMs.

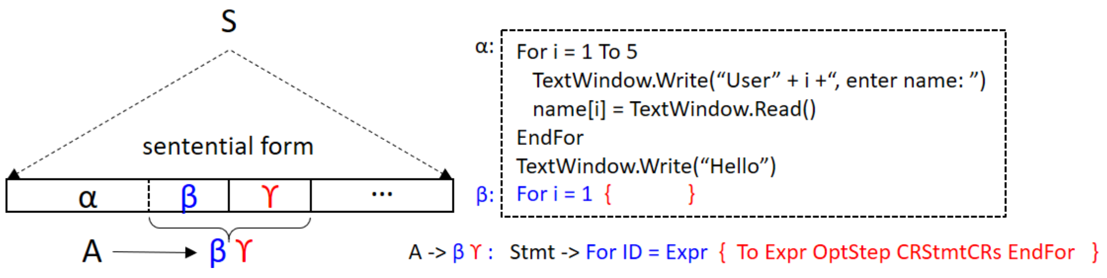
1. Introduction

Many integrated development environments (IDEs), such as VSCode, provide syntax completion features that streamline the editing process for various programming languages. These features typically display pop-up menus as users type code, suggest functions and APIs, assist with function parameters, clarify variable or method names, and predict the next steps to complete code. Today, developers of IDEs should prioritize implementing syntax completion features for every supported language.

Conventional code completion tools in IDEs usually display all possible attributes or methods when a user types a trigger character like '.' or '='. Moreover, without ranking the suggestions, users are forced to scroll through a lengthy, alphabetically sorted list, which can be slower than typing out the method name manually. Research indicates that developers often depend on prefix filtering to narrow down the options [20].

To enhance both efficiency and cost-effectiveness, it is advantageous to tackle this implementation systematically by adhering to a detailed specification. An analytical method rooted in syntax analysis and leveraging the extensively studied LR parsing theory [1] has been proposed. Sasano and Choi [30, 31] introduced a formal definition for code completion candidates γ corresponding to a prefix $\alpha\beta$ of some sentential form. When there exists a production $A \rightarrow \beta\gamma$ in the LR grammar such that $\beta\gamma$ can be reduced to the nonterminal A, γ becomes a code completion candidate, which we call a structural candidate. This concept is depicted in Figure 1.

As an example, consider a code completion request for the prefix 'For i = 1' in Microsoft Small Basic language. Here, β represents 'For ID = Expr' of the production, which corresponds to a sequence of terminal and nonterminal

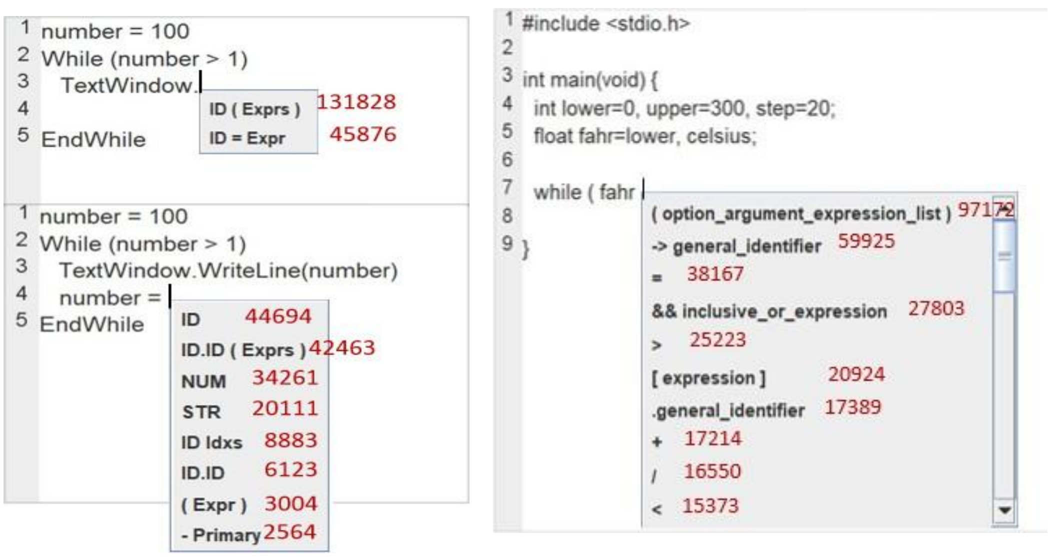

S
sentential form
α
β
Υ
…
A
βΥ
α:
β:
A -> βΥ : Stmt -> For ID = Expr { To Expr OptStep CRStmtCRs EndFor }

```
For i = 1 To 5
  TextWindow.Write("User" + i +", enter name: ")
  name[i] = TextWindow.Read()
EndFor
TextWindow.Write("Hello")
For i = 1 {          }
```

Figure 1: The idea of structural candidates for code completion using LR parsing [30].

symbols describing the initial structure of a for loop. The method developed by Sasano and Choi [30, 31] can automatically generate a completion candidate γ, such as 'To Expr OptStep CRStmtCRs EndFor', which completes the rest of the for loop according to the production 'Stmt → For ID = Expr To Expr OptStep CRStmtCRs EndFor'. In response to the completion request for 'For i = 1', an IDE would return this candidate γ. Building on this work, their subsequent research [4] introduced a ranking mechanism to prioritize more likely candidates when multiple completions are possible for a given prefix. This approach leverages pre-analyzed frequencies of candidate occurrences in existing open-source projects.

Figure 2a illustrates an example of programming in Small Basic, an educational programming language by Microsoft, aided by the suggested syntax completion technique. When a user queries suggestions at the cursor position on Line 3 and 4, the editor prompts a dropdown menu showcasing various syntax completion options. These suggestions are ranked based on their frequency in sample programs, which our previous method employs to gather these candidates. The suggested system was designed to be language-agnostic; it is applicable to any programming language if an LR grammar for it can be defined. The C programming language is such a case where the syntax completion system can be applied immediately. For example, Figure 2b shows a usec ase of the system for C programs. When programmers want the editor to complete the remaining part of line 7, a menu of structural candidates appears.



```
1 number = 100
2 While (number > 1)
3   TextWindow.
4
5 EndWhile
```

```
ID ( Exprs )  131828
ID = Expr      45876
```

```
1 number = 100
2 While (number > 1)
3   TextWindow.WriteLine(number)
4   number =
5 EndWhile
```

```
ID            44694
ID.ID ( Exprs )42463
NUM           34261
STR           20111
ID Idxs        8883
ID.ID          6123
( Expr )       3004
- Primary      2564
```

```
1 #include <stdio.h>
2
3 int main(void) {
4   int lower=0, upper=300, step=20;
5   float fahr=lower, celsius;
6
7   while ( fahr
8
9 }
```

```
( option_argument_expression_list ) 97172
-> general_identifier  59925
=  38167
&& inclusive_or_expression  27803
>  25223
[ expression ]  20924
.general_identifier  17389
+  17214
/  16550
<  15373
```

(a) Syntax structure completion example for Microsoft Small Basic.

(b) Syntax structure completion example for C.

Figure 2: Syntax structure completion for Microsoft Small Basic and C in our prior system [4].

The methods proposed in [30, 31, 4] offer several advantages. They ensure that the suggested candidates are syntactically correct, enable ranking customization tailored to individual software projects, and can be implemented in a language-independent manner. Unlike identifier completion techniques, which focus solely on suggesting variable or function names [29, 10, 12, 32, 8], these methods take a broader approach. Nevertheless, the suggested candidates are restricted to sequences of terminal and nonterminal symbols. Once a candidate is selected, programmers must manually edit it into actual code, which can reduce productivity. Generating a code snippet from a candidate lies outside the scope of syntax analysis based on LR parsing. For this reason, users must manually type code to complete the suggested line. Moreover, this process is time-consuming and uncomfortable for programmers.

Therefore, there is a need for a sophisticated code completion prediction method that offers textual suggestions rather than just structural ones. LLMs can be an effective technology in this regard, having already played crucial roles in many fields of computer science. ChatGPT, a widely used LLM, allows coders to seek assistance by providing it with well-crafted prompts. However, if users cannot accurately describe the function (code suggestion) they need, the generated code may not serve the intended purpose. In most cases, users are unable to fully control ChatGPT's output, leading to arbitrary or unnecessary code that does not meet their needs. To address this problem, we introduced a hybrid method combining LR-parsing-based candidate ranking with LLM-based code generation.

In this study, we explore how LLM [26] can complement these methods. Given a prefix text $\alpha\beta$, the LR parsing- based approach first generates a candidate γ. Subsequently, the LLM generates a complete code snippet that adheres to the structure of the suggested candidate for the given prefix text. For example, our system can automatically compose a prompt to the LLM as

```
This is the incomplete Microsoft Small Basic programming language code:
1: For i = 1 To 5
2: TextWindow.Write("User" + i + ", enter name: ")
3: name[i] = TextWindow.Read()
4: EndFor
5: TextWindow.Write("Hello ")
6: For i = 1 {To Expr OptStep CRStmtCRs EndFor}

Complete the {To Expr OptStep CRStmtCRs EndFor} part of the code.
Just show your answer in place of {To Expr OptStep CRStmtCRs EndFor}.
```

where the suggested structural candidate is placed inside the braces. Then the LLM successfuly returned exactly what we expected as this.

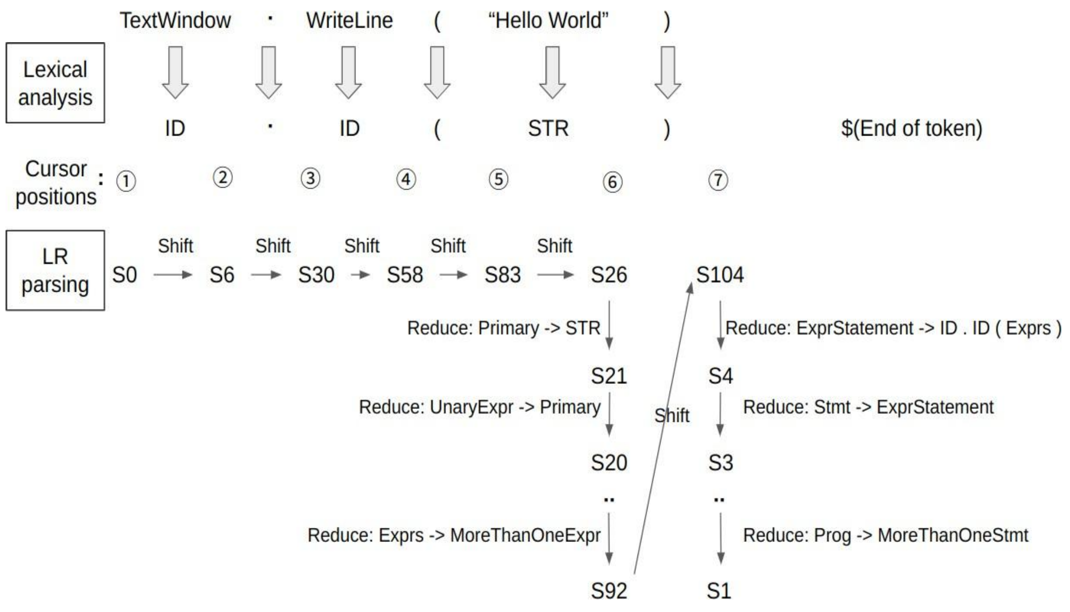
```
6:              To 5
7: TextWindow.Write(name[i] + ", ")
8: EndFor
```

Thus, the two approaches can work synergistically. The LR parsing-based analytical method excels at accurately defining the syntactic structure of code to be completed, while the LLM-based statistical technique can generate the corresponding textual content within the defined structure. To the best of our knowledge, this work represents the first effort to guide an LLM through prompts that incorporate structural candidate information derived from LR-parsing.

A fundamental research question is whether LLMs truly benefit from LR structure candidates in improving code completion. Due to the black-box nature of neural networks, it is difficult to understand how LLMs generate code completions. Since LLMs have already been trained on vast amounts of knowledge, they might achieve the same level of precision in code completion even without being provided with LR structural candidates.

To answer the question in this study, we assume an oracle perspective, where the next LR structural candidates are known in advance. By comparing the code completion output when these candidates are provided to the LLM versus when they are not, we verify that LLMs indeed benefit from LR structure candidates.

This study also investigates how to effectively leverage their potential to improve code completion precision when integrating LLMs with the guidance of LR structure candidates. Our strategy builds on findings from our previous research, which indicate that the top 1–3 ranked candidates often align with the correct option. As reported in [4], the top 1.8 candidates suggested for SmallBasic programs and the top 3.15 candidates for C11 programs, on average, matched the expected results during testing. These evaluation outcomes suggest that lower-ranking structural candidates should not be considered by the LLM. Constructing prompts based on the highest-ranking structural candidates is likely to guide the LLM effectively, ensuring that lower-priority candidates are excluded during code completion. Moreover, by developing two VSCode extensions, we demonstrated the practical



Viable Prefixes	States	Candidates	Ranks
ϵ	S0	ID . ID (Exprs)	1
ID	S6	. ID (Exprs)	1
ID .	S30	ID (Exprs)	1
ID . ID	S58	(Exprs)	1
ID . ID (	S83	STR	1
ID . ID (STR	S26		
ID . ID (Primary	S21		
ID . ID (UnaryExp	S20		
...	...		
ID . ID (Exprs	S92	)	1
ID . ID (Exprs)	S104		
ExprStatement	S4		
Stmt	S3		
...	...		
Prog	S1		

Cursor Position	States
1	S0
2	S6
3	S30
4	S58
5	S83
6	S26
6	S21
6	S20
6	...
6	S92
7	S104
7	S4
7	S3
7	...
7	S1

Figure 3: Collecting candidates using LR parsing (Hello World program in Microsoft Small Basic). The left table lists viable prefixes, parser states, predicted candidates, and ranks. The right table maps cursor positions to parser states [4].

application of our proposed system for programmers.

This is an extended version of our earlier work-in-progress report [2] by the following major contributions.

• We investigate the maximal improvement achievable by LLM based code completion in utilizing ideal LR structural candidates.

• We suggest how to choose LR structural candidates that can guide LLMs to produce better textual candidates than LLM without the guidance.

• We built two VSCode extensions or plugins for Microsoft Small Basic and C to apply our proposed system for programmers in a practical way where users can edit code with the aid of a list of suggested textual candidates.

• Our tool is language-agnostic and facilitates automatic ranked syntax completion for any programming language defined by an LR grammar, leveraging the capabilities of LLMs.

The rest of this paper are organized as follows. Section 2 describes the background knowledge of our work. Section 3 details our overall system. Section 4 presents the evaluation results and performance analysis with discussion in Section 5. Section 6 explains an implementation of a VSCode extention for code completion. Section 7 compares our work with existing research. Section 8 concludes the paper, which also describes the potential future work of this study.

2. Background: Ranked Syntax Completion with LR Parsing

In this section, we review a system for ranked syntax completion with LR parsing, as proposed in the previous work [4], which serves as the foundation for this research. The system operates in two main phases. The first phase, collection and ranking, is performed offline. During this phase, the system constructs a database from sample programs, mapping parse states to sets of ranked candidates. The second phase, query, is executed online. In this phase, the system retrieves a sorted list of candidates based on their ranks for the parse states associated with the cursor position being edited by a programmer. To clarify the underlying principles of the algorithms in each phase, Figure 3 provides an illustrative process with a sample program written in Microsoft Small Basic. For more details, readers can refer to [4].

2.1. Candidate Collection and Ranking (Offline Phase)

This section explains the key idea of candidate collection and ranking in the previous research [4]. During the candidate collection and ranking phase, a set of syntactically valid sample programs in a specific programming language were processed. This phase involves performing lexical analysis, LR parsing, and gathering candidate structures. The table on the left in Figure 3 shows the analysis outcomes from this collection phase for the given sample program. This section covers the details of this offline phase below.

2.1.1. Lexical Analysis

Lexing, also known as lexical analysis, is the process of converting a sequence of characters in source code (data programs) into a sequence of tokens, which represent the atomic units of the language. For instance, the lexical analysis phase processes the “Hello World” program in Microsoft Small Basic:

```
TextWindow.WriteLine("Hello World")
```

into a sequence of tokens:

```
ID . ID ( STR )
```

as shown in Figure 3. In this process, TextWindow and WriteLine are recognized as identifiers (ID), while the dot (.) and parentheses ((,)) are treated as their respective terminal symbols. The string literal "Hello World" is classified as the terminal STR. These terminal symbols, including ID and STR, are defined in the lexical analyzer for Microsoft Small Basic grammar. After tokenization, the sequence is finalized with the special token $, signaling the end of input for the LR parser.

2.1.2. LR Parsing

LR parsing [1] is a leading type of bottom-up parsing technique utilized in our study. It operates using a shift-reduce procedure, leveraging a stack to store grammar symbols and an input buffer to hold the string awaiting parsing. Initially, the stack is empty, and the entire input string resides in the buffer. The parser processes the input from left to right, shifting symbols from the buffer onto the stack. When the sequence of symbols β at the top of the stack matches the body of a production $A \rightarrow \beta$, a reduction is performed. This replaces β with the head of the production A on the stack. The process repeats until the start symbol is the only remaining element on the stack and the input buffer is empty.

For LR parsing, LR automaton is derived from LR grammrs [1]. LR automaton consists of states and transitions. Each state represents a set of LR items, while the transitions correspond to pushing terminal or nonterminal symbols onto the stack. A shift j action pushes the lookahead terminal onto the stack and transitions to state j. A reduce action for $A \rightarrow \beta$ pops symbols β from the stack, pushes A onto the remaining stack, and then transitions to another state.

Figure 3 shows how LR parser processes the Microsoft SmallBasic example program. We assume an LR automaton derived from the SmallBasic grammar[①]. The parsing begins from the initial state, S0, at cursor position 1. Given the token ID as the initial lookahead, the parser transitions from state S0 to S6, simultaneously pushing ID onto the stack. This transition is visualized by an arrow leading from S0 to S6, labeled "Shift", with the lookahead, ID, indicated above this label. The resulting stack is described by

[①] https://github.com/kwanghoon/sbparser the viable prefix ID for S6 in the table. The stack strings that appear in the stack of a shift-reduce parser in accepting computations are called vaiable prefixes [46].

Next, the parser transitions from S6 to S30 based on the lookahead, the terminal dot (.), pushing it onto the stack. The parser continues shift actions until the parser reaches state S26 at cursor position 6, where the viable prefix is ID · ID (STR and it encounters a close parenthesis as the lookahead. At this state, a reduce action is conducted for STR using the production Primary → STR, popping the terminal STR from the stack and pushing the nonterminal Primary onto the stack. The viable prefix becomes ID · ID (Primary. During this reduce action, the parser does not consume any additional lookahead, transitioning to state S21, still at the same cursor position 6. Following this, reduce actions are repeated, replacing the terminal STR with Primary, the nonterminal Primary with UnaryExpr, and, eventually, the nonterminal MoreThanOneExpr with Exprs. Throughout this process of reduce actions, the parser reaches at state S92 with the viable prefix ID · ID (Exprs. Note that the cursor position remains as 6. The parser now shifts on the lookahead close parenthesis, transitioning to state S104 at cursor position 7. With the sentinel token signaling the end, the parser, at state S104 with the viable prefix ID · ID (Exprs), conducts the reduce action for the important production.

ExprStatement → ID . ID (Exprs)

This production is crucial because the reduce action triggers candidate collection from the sample program. Following a series of subsequent reduce actions, the parser reaches state S1 and encounters the end of tokens, and the program has been successfully parsed.

2.1.3. Structural Candidates

Structural candidates for code completion can be intuitively explained using the notion of LR items [1]. Every LR item is a dotted production, represented as $A \rightarrow \beta \cdot \gamma$. If a user writes a text ending with the symbols β preceding the dot and requests completion suggestions afterward, γ can serve as a structural candidate to complete β. Consider state S6 with the viable prefix ID in Figure 3. The state comprises a set of 14 items:

01: [Stmt → ID · :, $]

02: [Stmt → ID · :, CR]

03: [ExprStatement → ID · = Expr, $]

04: [ExprStatement → ID · = Expr, CR]

05: [ExprStatement → ID · . ID = Expr, $]

06: [ExprStatement → ID · . ID = Expr, CR]

07: [ExprStatement → ID · . ID (Exprs), $]

08: [ExprStatement → ID · . ID (Exprs), CR]

09: [ExprStatement → ID · ( ), $]

10: [ExprStatement → ID · ( ), CR]

11: [ExprStatement → ID · Idxs = Expr, $]

12: [ExprStatement → ID · Idxs = Expr, CR]

13: [Idxs → · [Expr], =]

14: [Idxs → · [Expr] Idxs, =]

The items 07 and 08 are relevant to the cursor position (2) in the example program, which occurs right after the user has input TextWindow, i.e., ID. The expected subsequent input from the user is “.WriteLine ("Hello World")”. The ideal completion suggestion should be “. ID (Exprs)” from two items.

Utilizing parsing sample programs to identify structural candidates for code completion sets, our method apart from previous approaches [30, 31] that exclusively explored the LR automaton to deduce code completion candidates. At the same position by state S6, traditional methods would generate all conceivable candidates by examining symbols that can follow the centered dot in each LR item of the state. That is, they would suggest “:” from items 1 and 2, “= Expr” from items 3 and 4, and “. ID = Expr” from items 5 and 6, “( )” from items 9 and 10, and “Idxs = Expr” from items 11 and 12.

The offline method [4] that our research adopts is significantly different from the online methods [30, 31] that compute code completion structural candidates on-demand for specific program prefix positions. The offline method [4] collects structural candidates for all potential prefix positions of a particular sample program. We are then able to construct a database of these pre-ranked candidates, which users can reference later.

2.1.4. Collecting Structural Candidates

Structural candidates are gathered when a reduce action is performed on a production ($A \rightarrow \beta\gamma$) [4]. Specifically, candidates are collected after a sequence of shift and goto actions pushes the symbols γ onto the stack from a state containing the LR item A -> $\beta \cdot \gamma$.

Consider an initial parse state S_0 at cursor position 1 in Figure 3. The parsing begins with an empty prefix β_{S0}, and the parser simulates a request to complete the prefix. As the parsing progresses, symbols such as γ_{S0} = ID. ID (Exprs) are pushed onto the stack. This process continues until the parser reaches a state such as S_{104}, where a reduce action is performed for the production ExprStatement → ID . ID (Exprs). At this point, the candidate γ_{S0} is identified and inserted into the candidate database for the state S_0 with a ranking 1:

$$[\,S0 \rightarrow \{\, \text{ID . ID (Exprs)1} \,\}\,]$$

As parsing continues, candidates are identified for other states, such as S_6, at cursor position 2. The process is repeated for subsequent states, collecting candidates and updating the candidate database, ultimately building a structured database of possible candidates for each parse state. For example, when the parse reaches state S_{83}, the candidate STR1 is identified and added to the database as:

$$[\,S0 \rightarrow \{\, \text{ID . ID (Exprs)1} \,\}, S6 \rightarrow \{\, \text{. ID (Exprs)1} \,\},\ \cdots\ , S83 \rightarrow \{\, \text{STR1} \,\}\,]$$

This systematic gathering of candidates at each cursor position forms the basis of the candidate database, which is essential for syntax completion in LR parsing. Details of the collection algorithm can be found in [4].

2.2. Structural Candidate Query (Online Phase)

When a user requests syntax completion candidates at a cursor position during program editing, the program text up to the designated cursor position is parsed using the LR parser. Unlike traditional parsers that return an error, LR parser is designed to return the parse state where it halts. This is a parse state that is translated from the current cursor position. For instance, with cursor position 1 in Figure 3, the parse state S0 is returned. If the cursor is at position 3, it returns the parse state S30. It is worth noting that a single cursor position can correspond to multiple parse states. For the cursor positioned at 6, the returned set of states comprises { S26, S21, S20, ..., S92 }. The association between cursor positions and LR parse states is detailed in the right-hand side table of Figure 3. Details of the algorithm computing a set of parse states for a cursor position can be found in [4].

Upon obtaining the set of parse states corresponding to the cursor position, the pre-constructed database from the collection phase can be queried to fetch a set of ranked candidates for each state. The combined results from these sets will then be relayed to the editor, with structural candidates presented in accordance with their ranks.

2.3. Limitations

The suggested candidates are limited to structural ones, consisting of sequences of terminal and nonterminal symbols. After selecting a candidate, users are required to manually edit it into actual code, which can hinder productivity. To address this limitation, there is a need for a more advanced code completion prediction method that provides textual candidates rather than structural ones.

Our research aims to leverage LLMs such as ChatGPT, to generate textual candidates that align with one of the structural candidates retrieved from the database. The proposed approach is a hybrid method that combines LR-parsing- based ranked candidates with LLM-powered code generation, bridging the gap between structural and textual code suggestions.

3. Methodology: Combining LR Parsing-Based Ranked Syntax Completion with LLM

This section provides an overview of the proposed system that combines the LR parsing based ranked syntax completion with LLM as illustrated in Figure 4. The proposed system operates in two phases. The candidate collection phase, which is adopted from the previous research [4], constructs a database from sample programs, mapping parse states to sets of ranked candidates using LR parsing technique. A query (online) phase, which is proposed to integrate with LLMs in this research, retrieves a sorted list of candidates based on their ranks for a parse state corresponding to a given cursor position being edited. Following that, probable structural candidates are chosen from the sorted list to compose prompts, and then the LLM fleshes out the structural candidates to produce textual candidates, which will be displayed to the programmer for code completion.

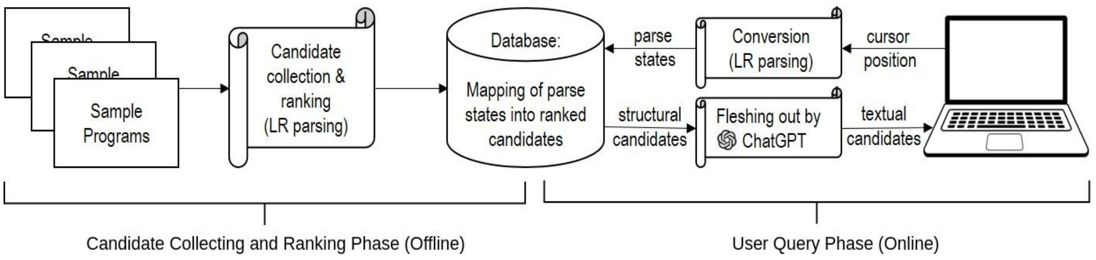

Sample
Sample
Sample Programs
Candidate collection & ranking (LR parsing)
Database: Mapping of parse states into ranked candidates
parse states
Conversion (LR parsing)
cursor position
structural candidates
Fleshing out by ChatGPT
textual candidates
Candidate Collecting and Ranking Phase (Offline)
User Query Phase (Online)

Figure 4: Overview of the system architecture.

This research focuses on the online phase, a critical aspect of the system, which involves automatically composing prompts for the LLM (e.g., ChatGPT). These prompts are generated using structural candidates provided by the LR- based method in combination with the user's partially written code information. During code editing, the system responds quickly to programmers' requests, providing relevant suggestions in an interactive manner. This approach represents a significant advancement over previous work [4], as it enables the system to suggest textual candidates rather than merely structural ones, thereby enhancing its practical feasibility and utility.

3.1. Do LR Structure Candidates to an LLM Actually Help? How?

Before delving into the online phase proposed in this study, let us discuss the following two research questions. First, does providing LR structure candidates to an LLM actually help? It might seem obvious that they would. However, since LLMs already possess extensive knowledge, they might be able to suggest code completions with similar precision, regardless of whether LR structure candidates are provided or not. Given the black-box nature of neural networks, which characterizes LLMs, it is often unclear how the outputs for code completion are generated. Therefore, our research investigates whether providing LR structure candidates is truly beneficial and, if so, to what extent it can help LLMs to improve performance for code completion.

Second, if LR structure candidates have the potential to assist LLMs in producing more precise code completions, it is crucial to identify effective strategies to unlock and utilize this potential fully. Is it possible to devise a strategy that maximizes the performance improvement achievable by providing LR structure candidates to an LLM? If not, what would be the best or reasonable strategy to achieve substantial gains while considering the limitations? Furthermore, can we design optimal strategies that are versatile and not restricted to specific programming languages, thereby broadening their applicability across diverse coding environments? Understanding these aspects would not only improve the utility of LR structure candidates but also provide insights into the fundamental interactions between LLMs and structured guidance.

The two research questions are summarized as follows.

• RQ1. Do LR structure candidates to an LLM actually help? If so, to what extent can it improve performance for code completion?

• RQ2. What is an effective method for selecting LR structural candidates to improve LLM-based code completion? How much of the potential performance gain identified in RQ1 can be realized through such a selection method?

To answer the two research questions, our methodology conducts the following experiments. First, to measure whether providing LR structure candidates improves LLM code completion, a test program set is pre-analyzed to identify which LR structure candidate is used at each cursor position. The precision of code completion is then compared between scenarios where the correct LR structure candidates are always provided to the LLM and where they are not for each cursor position. If the LLM already possesses knowledge about LR structure candidates, there will be no significant difference in code completion precision between the two cases.

However, if the LLM benefits from the provided LR structure candidates, the precision of code completion will be higher when the candidates are supplied.

Since the correct LR structure candidates are always provided to the LLM, there is no way to achieve better results than this. Therefore, the optimal approach to maximizing the LLM's code completion performance based on LR structure candidates is always to provide the correct LR structure candidates.

However, at this stage, it is impossible to accurately predict the sentences that will be written next or their corresponding LR structure candidates consistently. Therefore, a strategy is required to predict the upcoming LR structure candidates as accurately as possible. In the previous study [4], programs collected from open-source repositories were analyzed to investigate which LR structure candidates appeared more frequently in parser states corresponding to cursor positions. Notably, the study observed that, in the targeted languages—Microsoft Small Basic and C programs—one of the top 1 to 3 ranked LR structure candidates usually matched the upcoming LR structure candidate.

Based on this observation, we adopt a strategy where the top three LR structure candidates are provided to the LLM, generating three text suggestions for the programmer to choose the one that matches their intended next code. This approach enables the system to leverage the LLM's code completion capability by offering three highly accurate predictions of the next LR structure candidates, thereby maximizing the benefits of providing LR structure candidates. This is our methodology for combining LR structure candidates with an LLM to enhance the precision of code completion and evaluating the effectiveness of this approach. In the following subsections, we will delve into the specifics of how LR structure candidates are used to construct prompts for the LLM and the methods employed to assess the precision of code completion.

3.2. Automated Prompt Construction with Structural Candidates for Generating Textual Candidates Using LLM

We describe how our system automatically constructs prompts with structural candidates, enabling LLMs to generate textual candidates.

Prefix Extraction and Cursor Processing: In this step, relevant prefix and cursor position data are gathered from the user-written code in the code editor. Then, conversion of the cursor position to parse states is performed through the parser program that operates based on LR parsing [4]. The parse state information is used to generate a potential set of structural candidates from the database, which are further processed in subsequent steps to provide code completion suggestions.

Prompt Engineering: Leveraging the gathered prefix and structural candidate data, we formulated prompts to obtain ChatGPT's response. For this experiment, we selected the 'gpt-3.5-turbo-0125' model due to its superior performance in code completion tasks. Through extensive prompt engineering, we identified the most effective prompt structure. Prefix and candidate information was subsequently input into the ChatGPT prompt to request replacements for our structural candidates with actual candidates.

Figure 5 shows one prompt template featuring LR structural candidate guidance and the other without it. The prompt template featuring LR structural candidate guidance provides explicit context and direction to the system (ChatGPT), enabling it to generate more accurate and contextually relevant responses. The prompt templates are instantiated with Microsoft SmallBasic and its example program by replacing {Name of Programming Language} with Microsoft Small Basic Programming language, providing a program prefix for {Program Prefix}, and substituting a structural candidate for {Suggested Structural Candidate}, as shown in Figure 6. In this example of a prompt with structural candidate guidance, the prefix consists of incomplete code (lines 2 to 4), while the candidate in line 5 serves as a clear instruction to complete the '(Expr)', representing the expression part of the code. The prompt specifies that ChatGPT should provide the answer in place of '(Expr)' at the cursor position at the end of line 4, offering clear guidance for generating the response.

On the other hand, the other prompt template without LR structural candidate guidance can similiarly be instantiated to a prompt example. A prompt without LR syntactic structure (guidance) provides no explicit direction, leaving the system to interpret the code's intent with minimal input (prefix). While the prefix remains unchanged in this example, line 5, without structural candidate guidance, offers no concrete hints. Instead, it provides a vague instruction to replace the 'next token or line,' as illustrated in Figure 5. This lack of guidance can result in responses that deviate from the expected outcome, as the system lacks specific cues to complete the task accurately.

Similarly, Figure 7 shows two prompts constructed from instantiating the same prompt templates with C programming language and an example C program. They can be understood in the same way as before.

Note that our system using LLMs can remain to be language agnostic by having these prompt templates instantiated with parameters depending on each specific programming language.

Prompt Template with Structural Candidate Guidance
1: This is the incomplete {Name Of Programming Language} code:
2: {Program Prefix}
3: {Suggested Structural Candidate}
4: Complete the {Suggested Structural Candidate} part of the code
5: in the {Name Of Programming Language}.
6: Just show your answer in place of {Suggested Structural Candidate}.

Prompt Template without Structural Candidate Guidance
1: This is the incomplete {Name Of Programming Language} code:
2: {Program Prefix}
3: 'next token or line'
4: Complete the 'next token or line' part of the code
5: in the {Name Of Programming Language}.
6: Just show your answer in place of 'next token or line'.

Figure 5: Prompt templates featuring LR structural candidate guidance.

Fleshing Out Code with ChatGPT: After formulating the prompts using the gathered prefix and structural candidate data, ChatGPT is tasked with fleshing out the structural candidates into full code suggestions. The selected

Example of Prompt with Structural Candidate Guidance

```
This is the incomplete Microsoft Small Basic programming language code:
number = 100
While (number > 1)
  TextWindow.WriteLine
  '(Expr)'
Complete the '(Expr)' part of the code in the Microsoft Small Basic
programming language. Just show your answer in place of '(Expr)'.
```

Example of Prompt without Structural Candidate Guidance

```
This is the incomplete Microsoft Small Basic programming language code:
number = 100
While (number > 1)
        TextWindow.WriteLine
                'next token or line'
Complete the 'next token or line' part of the code in the Microsoft
Small Basic programming language. Just show your answer in
place of 'next token or line'.
```

Figure 6: Prompt examples for Microsoft SmallBasic

gpt model processes the input prompt, replacing the structural candidates with corresponding textual suggestions that complete the code snippet. These textual candidates, generated by ChatGPT, are then presented as potential code completions for the user. This step significantly enhances the system's ability to provide meaningful and contextually relevant code completions, making it a key part of the overall code completion process.

3.3. Evaluation Metrics

To measure how precise textual candidates are, we compare the responses generated by ChatGPT with the correct answers from our test set. The quality of ChatGPT's responses was evaluated using established metrics, including SacreBLEU, a widely used method for assessing LLMs, as well as SequenceMatcher similarity.

Example of Prompt with Structural Candidate Guidance

```
This is the incomplete C programming language code:
int main(void)
  {
       char s[1000];
       int i = 0;
       int loop = 1;
       'while (expression) scoped_statement'
Complete the 'while (expression) scoped_statement' part of the code
in the C programming language. Just show your answer in place of
'while (expression) scoped_statement'.
```

Example of Prompt without Structural Candidate Guidance

```
This is the incomplete C programming language code:
int main(void)
  {
       char s[1000];
       int i = 0;
       int loop = 1;
       'next token or line'
  Complete the 'next token or line' part of the code in the C programming
  language. Just show your answer in place of 'next token or line'.
```

Figure 7: Prompt examples for C.

SacreBLEU: SacreBLEU is a metric used to evaluate the quality of machine-generated text by comparing it to reference text. It calculates a score based on the overlap of n-grams between the generated and reference sequences. The formula for the BLEU (BiLingual Evaluation Understudy score) score can be expressed as:

$$\text{BLEU} = \text{BP} \cdot \sum_{n=1}^{N} wn \log pn$$

$$\text{BLEU} = \text{BP} \cdot \textstyle\sum_{n=1}^{N} wn \log pn$$

where BP is a brevity penalty, wn is the weight assigned to n-grams of length n, N is the number of n-grams, and pn is the precision of n-grams of length n. The brevity penalty is applied to penalize short translations, and pn is calculated as the ratio of matched n-grams between the generated and reference sequences to the total number of n-grams in the generated sequence. SacreBLEU is an implementation of BLEU that standardizes preprocessing and tokenization, ensuring consistent comparisons across different systems [25]. In our experiment, the SacreBLEU score measures the n-gram similarity between the reference code sequence and the generated code sequence. It counts how many n-grams in the generated code sequence match (token-by-token) n-grams in the reference code sequence.

In our experiment, we utilized 1-gram precision, which measures the proportion of individual tokens in the generated text that match those in the reference text, evaluated on a token-by-token basis. This metric was chosen because it provides a straightforward measure of token-level accuracy, which is crucial for evaluating machine- generated code. By focusing on the correct tokens being present, regardless of their order, 1-gram precision ensures a clear assessment of the generated text's alignment with the reference. In this experiment, 1-gram precision is also represented by the sentence 'SacreBLEU score'.

SequenceMatcher: SequenceMatcher is a Python class from the difflib module (Python version 3.12.5) that measures the similarity between two sequences of strings (in terms of characters) by identifying the best alignment between them. Given two sequences, find the length of the longest subsequence present in both of them. It uses a greedy algorithm to identify matching subsequences and calculates a similarity ratio based on the total length of matching elements. The ratio is given by:

ratio = $2 \cdot M / (TA + TB)$

where M is the number of matching characters, and TA and TB are the lengths of the two sequences being compared. Generally, SequenceMatcher works at the character level, aligning sequences to maximize the number of matched characters. In our experiment, we used the parameter isjunk=None, meaning no elements are ignored during the comparison [6].

4. Experiment Results

Following our previous research [4], we selected two programming languages, Microsoft Small Basic [22] and C11 [14] for our experiments. These languages are popular choices for introductory programming. Microsoft Small Basic is designed for coding education, whereas C is a widely portable programming language known for its pointer features. Table 1 provides a summary of the grammatical statistics for the two programming languages. It details the number of productions for each language's grammar, the count of parse states, and the dimensions of each LALR(1) automaton based on the quantities of shift, reduce, and goto actions [4].

For each programming language, we prepared both a learning set and a test set of pre-existing programs [4]. The training set, referred to as the dataset, consisted of 3,701 Small Basic programs totaling 789,023 lines of code, sourced from the Small Basic community. Additionally, it included 412 C11 programs comprising 308,599 lines of code, obtained from open-source software repositories as cJSON-1.7.15, lcc-4.2, cdsa (commit c336c7e), bc-1.07, gzip-1.12, screen-4.9.0, make-4.4, and tar-1.34. Using the training set, a ranked LR structural candidate database was constructed through the method previously described in Section 2.1.

The test set for Microsoft Small Basic was obtained from its tutorial programs. It consists of 27 programs, totaling 155 lines, sourced from the widely recognized Microsoft Small Basic tutorial available on its official webpage. This tutorial offers engaging and educational content tailored for beginner learners. All the programs are available in [3]. The test set for C11 included 106 programs, totaling 11,218 lines, consisting of solutions from the renowned C programming book by Kernighan and Ritchie. All the programs can be found in [3].

Table 1: Grammatical statistics for Microsoft Small Basic and C [4].

PLs	Microsoft SmallBasic	C11
Num. of prod. rules	61	335
Num. of parse states	119	529
Num. of shift/reduce	816	9209
Num. of goto	222	1907

To answer the research questions presented in Section 3.1, we pre-analyzed the test set programs to prepare the LR structure candidates together with the actual texts. These actual texts are utilized to evaluate the response from the LLM. Specifically, for every cursor position in the test set programs, we identified the LR structure candidates, the list of tokens composing each candidate, and the position and lexeme of each token in advance. For each test program with a name, a list of triples of an LR parse state, a structural candidate, and a list of another triples of a line number, a column number, and a lexeme of each token. Figure 8 illustrates an example of a pre-analyzed test set.

Using the pre-analyzed information, prompts are created for each cursor position (parser state) based on the corresponding pre-identified LR structure candidates, which represent the ideal choices, and provided to the LLM. The code completion returned by the LLM are then compared to the expected text candidate for that cursor position, and the precision is calculated using the previously explained metrics by comparing the returned code completion with the expected text candidate. This process is repeated for every cursor position in all test programs to determine the average precision when LR structure candidates are provided to the LLM. The same process is then repeated, but with prompts modified to exclude LR structure candidates, to calculate the average precision for all cursor positions across all test programs. By comparing these two average precision values, we aim to answer Research Question 1.

```
0 ID . ID ( Exprs )
   1,1: TextWindow
   1,11: .
   1,12: WriteLine
   1,21: (
   1,22: "Hello World"
   1,35: )

6 . ID ( Exprs )
   1,11: .
   1,12: WriteLine
   1,21: (
   1,22: "Hello World"
   1,35: )

30 ID ( Exprs )
   1,12: WriteLine
   1,21: (
   1,22: "Hello World"
   1,35: )

58 ( Exprs )
   1,21: (
   1,22: "Hello World"
   1,35: )

83 STR
   1,22: "Hello World"

92 )
   1,35: )
```

Figure 8: A pre-analyzed test set example for Microsoft SmallBasic HelloWorld Program in Figure 3.

The pre-identified LR structure candidates are often among the top-ranked candidates analyzed in [4]. However, they can be the second, third, or even lower-ranked candidates found in the database for the corresponding parse state. This variability is why we refer to them as ideal LR structural candidates.

Table 2: Comparative analysis of experimental results: WithIdealGuide and WithoutGuide.

PLs	SacreBLEU (%) WithIdealGuide	SacreBLEU (%) WithoutGuide	SequenceMatcher (%) WithIdealGuide	SequenceMatcher (%) WithoutGuide
Microsoft Small Basic	49.790	40.798	44.703	37.897
C11	28.368	15.472	28.658	15.074

The pre-analyzed information for the Microsoft SmallBasic and C test sets is publicly available in the repository②. It is worth noting that this analysis is also based on LR parsing and can be applied to any programming language defined with LR grammars.

In the following sections, we present the experimental results, analyze the performance of our system in code completion tasks, discuss the impact of candidate guidance, and compare results across Microsoft Small Basic and C.

4.1. Maximal Improvement in LLM-based Code Completion by Supplying Ideal LR Structural Candidates

To address the research question 1, we aim to investigate the maximum improvement that can be achieved when providing ideal LR structural candidate as prompts to ChatGPT, one of the popular LLMs.

② https://github.com/monircse061/ChatGPT-Code-Completion-Work

With Ideal Structural Candidate Guidance (WithIdealGuide): In this setup, ideal LR structural candidates are explicitly provided as prompts to guide the LLM during code completion. For each program, a set of cursor positions with ideal structural candidates is extracted from the test set and LR parser states are computed for the cursor positions. The working steps in this type of experiment are outlined as follows:

1. Set the program text up to just before the cursor position (line1, col1) as the prefix.
2. Specify the ideal structural candidate as the guide to the ChatGPT.
3. Construct a prompt that requests code completion matching the ideal structural candidate.
4. Request a code completion from ChatGPT and compare it with expected text candidate.

Without Structural Candidate Guidance (WithoutGuide): This setup analyzes LLM performance without explicitly including LR structural candidates in the prompts. The extraction process for each program in the test set includes the cursor positions ignoring the ideal structural candidates. The steps for this type of experiment are as follows:

1. Set the program text up to just before cursor position as the prefix.
2. Construct a prompt without including the structural candidate.
3. Request a code completion from ChatGPT and compare it with expected text candidate.

4.1.1. Analysis of Average Precision Improvements

We present a summary of this experimental results for both WithIdealGuide and WithoutGuide in Microsoft Small Basic and C languages which are depicted in Table 2. When ideal candidate guidance is included in the prompt, we observed an improvement of approximately 7% to 14% in ChatGPT's predictions, as measured by SacreBLEU and SequenceMatcher precision, for Microsoft Small Basic and C11 based on our test programs. Here, the precision indicates how similar our system's LLM response is to the actual text.

With ideal structural candidate guidance in Microsoft Small Basic, we iterated our experiment through all possible cursor positions for each of the 27 test programs, calculated the evaluation metrics, and averaged the precision for each program. This process was repeated for the whole test set. Finally, we calculated the mean precision across the 27 programs using SacreBLEU and SequenceMatcher similarity. On average, with ideal structural candidate guidance, our system predicts the textual code suggestion with nearly 50% accuracy for each testing program of Microsoft Small Basic when using SacreBLEU as an evaluation metric. Precision is at nearly 45% when SequenceMatcher similarity is taken into account for this language. When candidate guidance is not provided in the prompt, ChatGPT's predictions can sometimes deviates. Without candidate guidance, SacreBLEU and SequenceMatcher precision are

nearly 41% and 38%, respectively, for Microsoft Small Basic, which is approximately 7% to 9% lower than with ideal guidance, as shown in the results in Table 2. In other words, the ideal candidate guidance improved SacreBLEU and SequenceMatcher precision for Microsoft Small Basic by 7% to 9%.

A similar process was applied to C11. For 106 C11 test programs with ideal structural candidate guidance in the prompt, the average SacreBLEU score was 28.368%, indicating that our system can forecast accurate code completion suggestions in this language as well. Additionally, the SequenceMatcher similarity was 28.658%. However, without candidate guidance in the prompt, the SacreBLEU and SequenceMatcher precision were 15.472% and 15.074%, respectively, showing approximately 13% to 14% decrease compared to predictions with ideal guidance. Inversely, guiding the LLM with the ideal structural candidate resulted in an improvement of approximately 13% to 14% during experiments conducted with the C programming language using the LR parsing technique.

4.1.2. Analysis of Winning Count Distributions

In this section, we provide a detailed analysis of the comparison between WithIdealGuide and WithoutGuide by examining the distribution of winning count based on the length of the ranked LR structure candidate list in parser states.

Precision Comparison Graph of WithIdealGuide and WithoutGuide in Microsoft Small Basic Language: For each program in the Small Basic test set, we analyze the outcomes of the precision, based on SacreBLEU precision (1-gram), of two textual parsing candidates generated from the two experiments for various candidate list lengths, as follows:

• Count the number of cases where WithIdealGuide achieves higher precision than the WithoutGuide.

• Count the number of cases where WithoutGuide achieves higher precision than the WithIdealGuide.

• Count the number of cases where both WithIdealGuide and WithoutGuide yield the same precision.

These results are plotted in the Figure 9a for structural candidate list lengths ranging from 1 to 11. The graph highlights the significant improvement of textual parsing candidate guided precision for shorter candidate list lengths (approximately 1 to 3). In this graph, the blue line, representing instances where ideal guided precision surpasses unguided precision, dominates, indicating that textual candidate by the ideal guidance is highly effective when the suggestion list is concise. This aligns with the notion that shorter lists are more user-friendly and allow for faster identification of correct textual candidates.

Precision Comparison Graph of WithIdealGuide and WithoutGuide in C Language: For programs in the C11 dataset, we follow a similar analysis where WithIdealGuide achieves higher, lower, or equal precision to WithoutGuide. These results are plotted in the graph of Figure 9b for structural candidate list lengths ranging from 1 to 35.

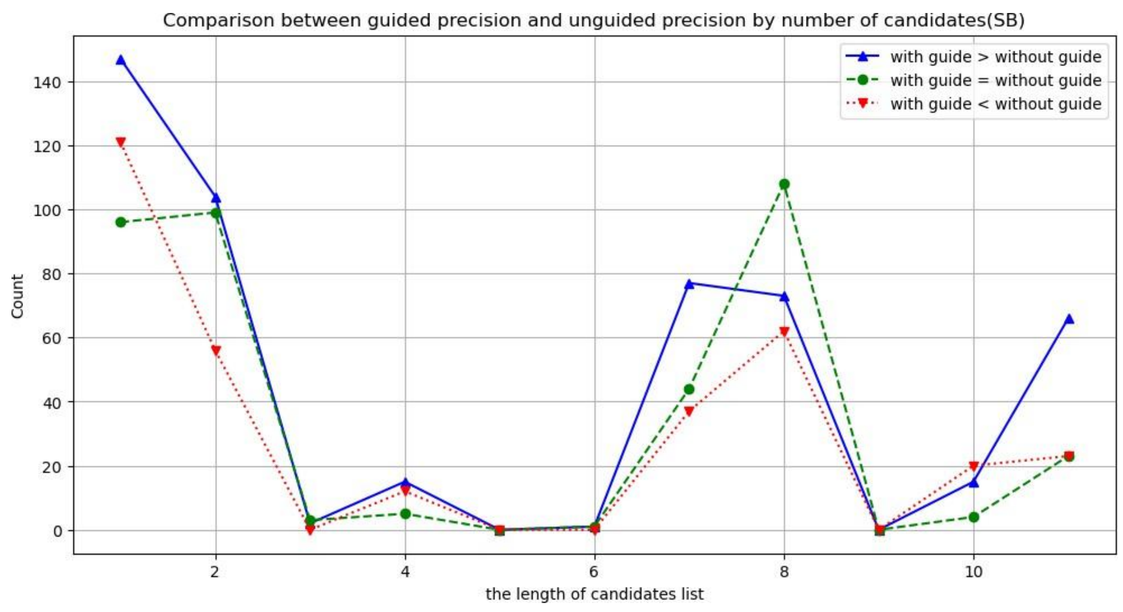
The graph illustrates the precision results for the textual candidates based on SacreBLEU precision (1-gram) for various candidate list lengths, based on result analysis of C programming language. The plot differentiates cases where guided precision is higher, equal, or lower than unguided precision across a range of candidate list lengths. The results consistently indicate that guided precision surpasses unguided precision across nearly all candidate lengths. Specifically, at shorter candidate list lengths (1–3), the number of cases with higher guided precision is substantially greater, suggesting that a smaller suggestion list leads to higher precision when guidance is provided. This trend highlights the importance of having a concise candidate list combined with guidance to achieve optimal precision. Additionally, although differences in precision become less noticeable as the candidate list length increases, guidance still provides a measurable benefit, confirming its effectiveness across various list lengths.

Based on the experimental results and graphs, we can assert that although LLMs encapsulate a vast amount of information, their black-box nature makes it impossible to determine exactly what information they contain. However, the experimental results from WithIdealGuide and WithoutGuide confirm that providing LR structure candidates improves the precision of LLM code completion, thereby addressing RQ1. It is expected that expanding the dataset to construct enhanced ranking database could further enhance precision.

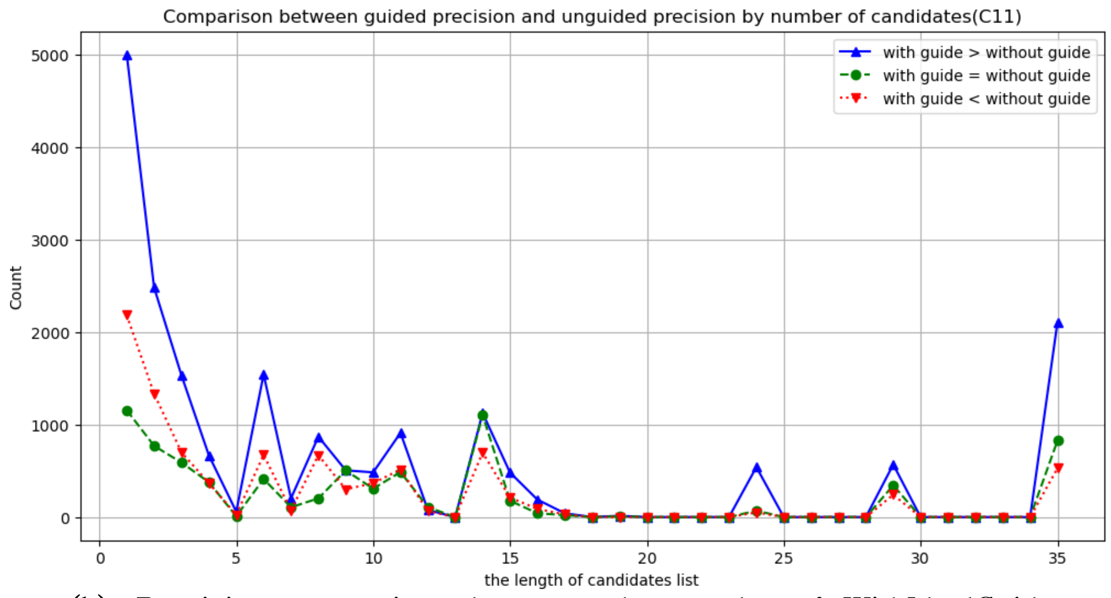
4.2. An Effective Strategy of Selecting the Top Three LR Structural Candidates to Enhance LLM-based Code Completion

Now, we have established that providing ideal LR structure candidates has the potential to enhance LLM-based code completion. In this section, we propose an effective strategy for selecting LR structure candidates that is close to selecting the ideal ones using the collected ranking database. Furthermore, we conduct experiments to assess how closely the LLM's code completion precision, when provided with candidates selected through this strategy, approaches the precision achieved with the ideal LR structure candidates analyzed in the previous section. Through this experiment, we aim to address Research Question 2.

Our strategy is to select the top 1 to 3 LR structural candidates from the ranked list for a given cursor position. This aligns with the method described in our previous work [4]. Insights from that study reveal that, in both the Microsoft Small Basic and C11 languages, the top 1–3 candidates were frequently the correct LR structural candidates, further supporting the validity of this selection criterion.



(a) Precision comparison between the results of WithIdealGuide and WithoutGuide in Microsoft Small Basic.



(b) Precision comparison between the results of WithIdealGuide and WithoutGuide in C.

Figure 9: Consistent precision gains across candidate list lengths with ideal structural candidate guidance, in both Microsoft Small Basic and C.

Table 3: Comparative analysis of experimental results: WithIdealGuide, WithinTop3Guide, WithTop1Guide, and WithoutGuide.

Programming Languages	Experiment Types	SacreBLEU (%)	SequenceMatcher (%)
Microsoft Small Basic	WithoutGuide	40.798	37.897
	WithIdealGuide	49.790	44.703
	WithinTop3Guide	45.733	43.897
	WithTop1Guide	38.524	37.097
C	WithoutGuide	15.472	15.074
	WithIdealGuide	28.368	28.658
	WithinTop3Guide	26.222	27.810
	WithTop1Guide	20.217	20.464

To address RQ2, we conducted another experiment where LR structural candidates corresponding to the LR parser state were retrieved in ranked order (e.g., Rank 1 candidate, Rank 2 candidate, Rank 3 candidate, etc.) based on their frequency in the ranking database. The top three ranked candidates were then selected, and separate prompts were constructed for each candidate to request code completions from LLM. Among the three results returned from the LLM for the top three ranked structural candidates, we selected the one that yielded the highest precision. We refer to this strategy as WithinTop3Guide.

To contrast with the WithinTop3Guide, we also introduce another strategy, WithTop1Guide, in which only the top-ranked candidate (Rank 1 candidate) is always selected for every LR parser state (cursor position) to construct prompts for the LLM across all test programs.

1. **WithinTop3Guide:** For each LR parser state, select the structural candidate from the top three that yields the highest precision. Then, calculate the average of these highest precision values across all LR parser states.

2. WithTop1Guide: Calculate the average precision when only the top-ranked structural candidate is used across all LR parser states.

The overall results with WithIdealGuide, WithinTop3Guide, WithTop1Guide, and WithoutGuide are presented in Table 3. The difference between WithIdealGuide and WithoutGuide in SacreBLEU precision is 8.992 for Small Basic and 12.896 for C. However, the difference between WithIdealGuide and WithinTop3Guide in SacreBLEU precision is significantly smaller: 4.057 for Small Basic and 2.146 for C. In other words, it was confirmed that the WithinTop3Guide strategy achieves two to six times precision closer to the precision of WithIdealGuide, which always suggests the correct LR structure candidates for code completion. The strategy for selecting effective syntactic structures, as shown in the experimental results, demonstrated that presenting the top 3 structural candidates is suitable for the Small Basic tutorial programs and C11 K&R exercise solution programs.

As previously, we also provide a detailed analysis of the distribution of winning count based on the length of the ranked LR structural candidate list in parser states.

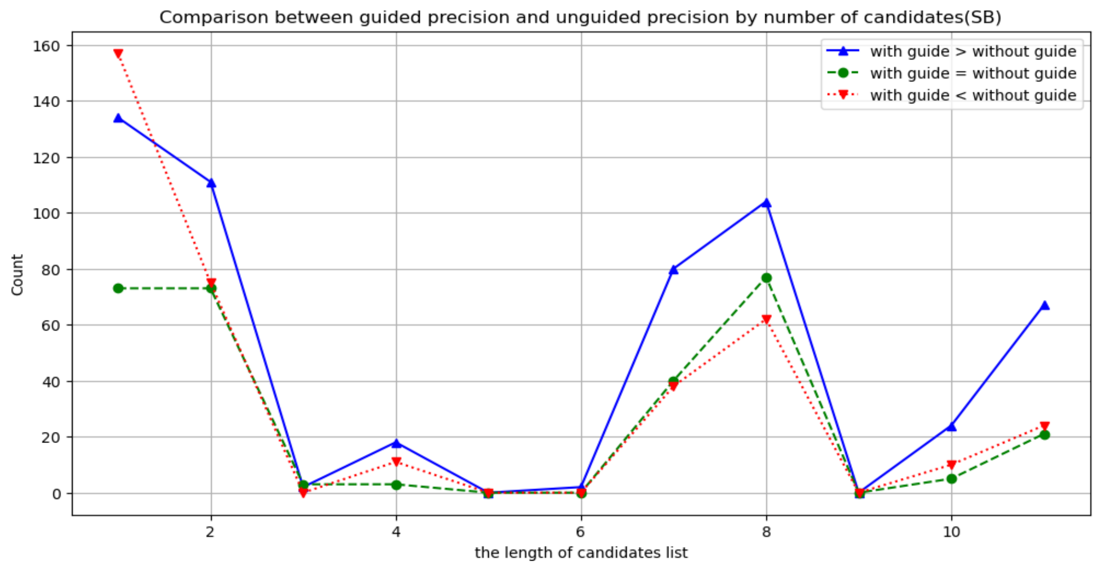
Precision Comparison Graphs for Different Candidate List Lengths: Given the experimental results, we compared the average precision among WithinTop3Guide, WithTop1Guide, and WithoutGuide for each LR parser state (cursor position) in all test programs of the two introductory programming languages.

Figure 10 and Figure 11 illustrate the precision comparison among different approaches across various candidate list lengths: WithinTop3Guide, WithTop1Guide, and WithoutGuide for code completion in the Microsoft Small Basic language and C. Each graph analyzes the effectiveness of guided versus unguided structural candidate selection across various candidate list lengths, providing insights into their impact on precision.

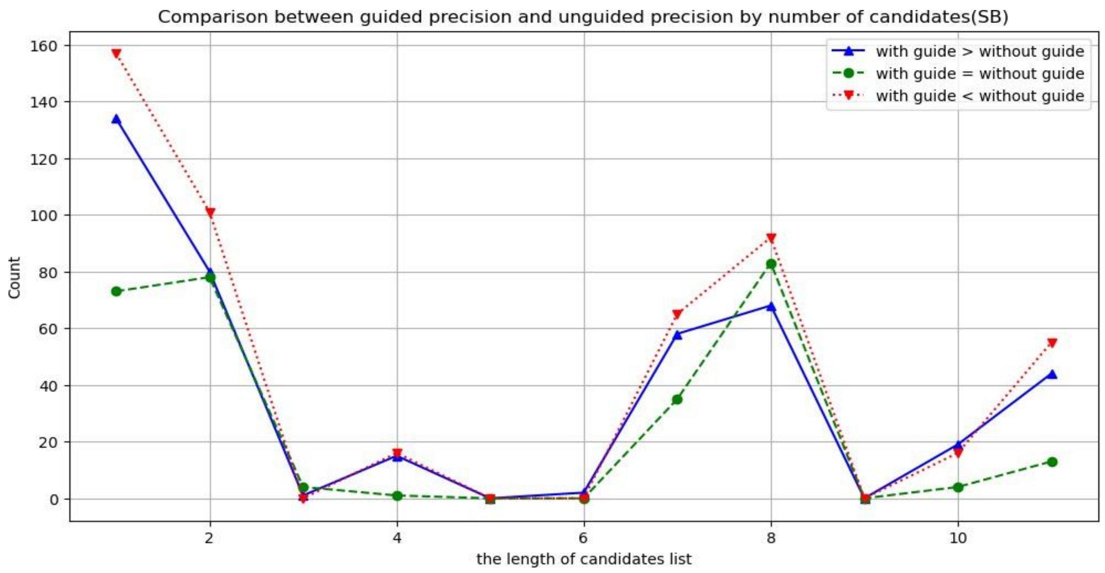
The results in Figure 10a, which compares the number of cases for WithinTop3Guide and WithoutGuide in Microsoft Small Basic, indicate that WithinTop3Guide performs, overall, better in terms of precision compared to WithoutGuide. However, Figure 10b shows that WithTop1Guide does not produce as precise code completions as WithoutGuide in Microsoft Small Basic. The reason for this could be attributed to our previous research findings [4] which revealed the top 1 LR structure candidates account for less than half of the correct ones in Small Basic. Another reason may be that Small Basic is a relatively small programming language, allowing LLMs to predict code completion more effectively.

For C, WithinTop3Guide in Figure 11a demonstrates improved precision compared to WithoutGuide. Even WithTop1Guide, as depicted in Figure 11b, outperforms WithoutGuide as well. WithoutGuide generally performs worse in terms of precision across all list lengths, highlighting the importance of guided approaches in enhancing completion accuracy. These results strongly support that incorporating guided syntactic structures significantly improves code suggestion systems, especially for the top 1-3 ranked candidates, making them effective.

Overall, using ChatGPT with LR parsing-based structural candidates was effective in providing code completion suggestions, particularly when the selected candidate is within the top three. In such cases, our system demonstrates correct suggestions, which is notable. This indicates that the system can be beneficial. Based on the evidence provided, we can affirmatively answer RQ2.

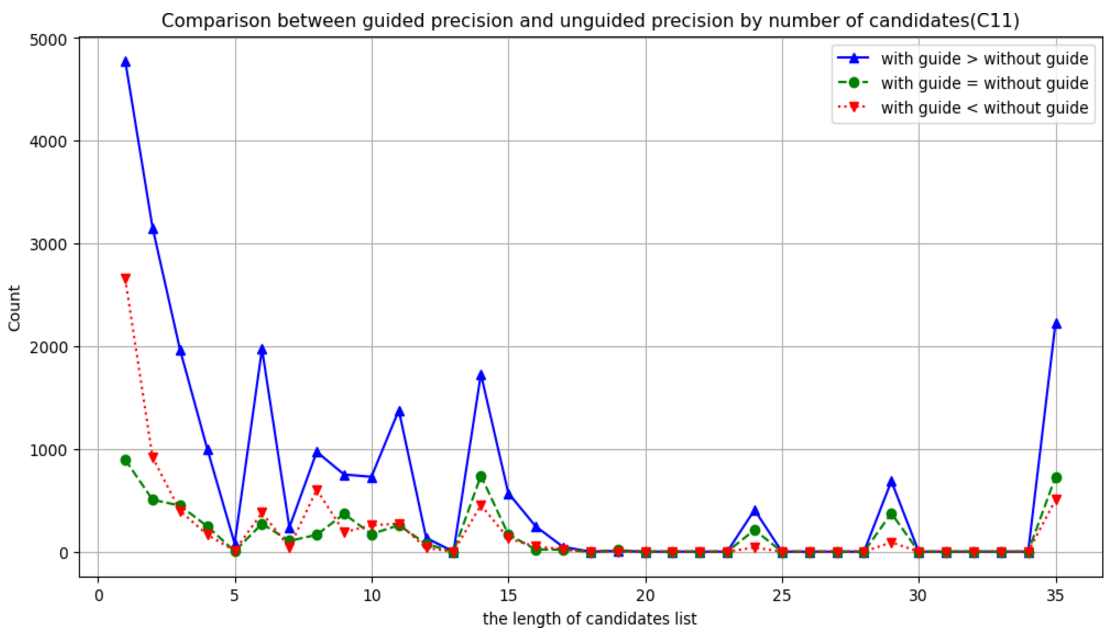


(a) Precision comparison of WithinTop3Guide and WithoutGuide for different candidate list lengths.

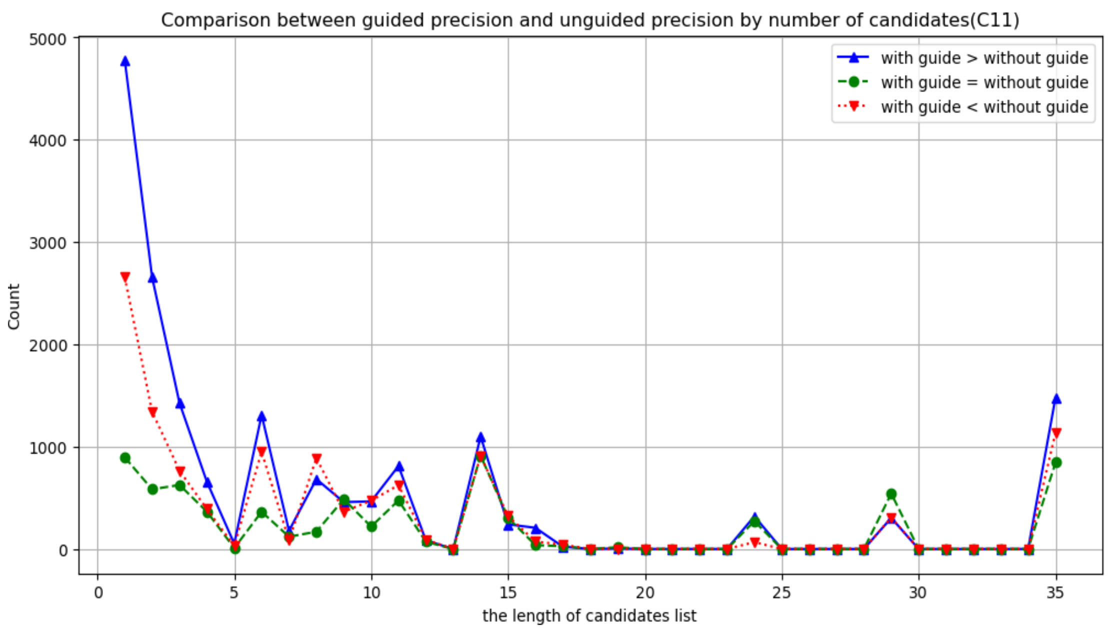


(b) Precision comparison of WithTop1Guide and WithoutGuide for different candidate list lengths.

Figure 10: Precision comparison: WithinTop3Guide outperforms WithTop1Guide and WithoutGuide in SmallBasic.



(a) Precision comparison of WithinTop3Guide and WithoutGuide for different candidate list lengths.



(b) Precision comparison of WithTop1Guide and WithoutGuide for different candidate list lengths.

Figure 11: Precision comparison: WithinTop3Guide outperforms WithTop1Guide and WithoutGuide in C.

4.3. A Comparative Analysis of Grammar Provision in LLM-based Code Completion using LR parsing

Code completion is a key feature in modern integrated development environments (IDEs), aiding programmers in improving coding efficiency. Grammar plays a crucial role in programming, ensuring syntactic correctness and improving code comprehension. Recently, large language models (LLMs) have enhanced code completion without explicit knowledge of language grammar in the given prompt. Yet, the role of explicit grammar provision in enhancing prediction accuracy remains an open question. This study investigates whether incorporating LR grammar-based structural candidates into LLM-driven code completion improves performance.

4.3.1. Grammar in Code Completion

Recent advancements in LLMs, such as ChatGPT, have demonstrated remarkable capabilities in code generation. LLM-based code completion leverages vast amounts of training data to predict and generate code, often achieving high accuracy even without explicit knowledge of the underlying syntax rules of the grammar of a programming language. However, LLMs sometimes produce inconsistent or arbitrary results due to their probabilistic nature. This raises an important research question: Does providing explicit grammar-based guidance improve LLM-based code completion? In this study, to address the research question, we investigate the effect of grammar provision on LLM-based code completion in conjunction with the LR parsing-based analytical method [30,31].

4.3.2. Grammar Definition

Recent A context free grammar, grammars for short defines the syntax of a programming language and plays a crucial role in structuring code completion tasks. As described by Aho et al. [1], a context-free grammar consists of:

1. Terminal symbols (tokens): The elementary symbols of a language.
2. Nonterminal symbols (syntactic variables): Represent a set of strings of terminals.
3. Productions: Rules defining how nonterminals can be replaced by sequences of terminals and other nonterminals.
4. Start symbol: The root nonterminal from which derivations begin.

For example, a production rule of if-else statement can be structured as:

stmt → **if** (expr) stmt **else** stmt

where stmt and expr are nonterminals, while if, else, and parentheses are terminals. In this experiment, Microsoft Small Basic's grammar consists of 60 production rules, whereas C programming language has 335 production rules. All production rules can be found in [4].

4.3.3. Prompt Engineering With and Without Grammar

Figure 12 compares two prompt templates: one with LR structural candidate guidance and grammar and another without grammar guidance. The grammar-based prompt includes production rules to enforce syntax constraints, while the grammar-agnostic prompt relies only on code context.

Figure 13 shows examples for Microsoft Small Basic and C, where grammar rules guide code completion in the structured approach. These prompts help assess whether explicit grammar constraints improve completion accuracy. Conversely, another example can be shown without grammar rules but with an LR structural candidate in Microsoft Small Basic and C.

Prompt Template with Grammar and Structural Candidate Guidance
1: {Grammar: Production Rules} 2: This is the incomplete {Name of Programming Language} code: 3: {Program Prefix} 4: {Suggested Structural Candidate} 5: Complete the {Suggested Structural Candidate} part of the code 6: in the {Name of Programming Language}. 7: Just show your answer in place of {Suggested Structural Candidate}.

Prompt Template without Grammar and Structural Candidate Guidance
1: This is the incomplete {Name of Programming Language} code: 2: {Program Prefix} 3: {Suggested Structural Candidate} 4: Complete the {Suggested Structural Candidate} part of the code 5: in the {Name of Programming Language}. 6: Just show your answer in place of {Suggested Structural Candidate}.

Figure 12: Prompt engineering with and without grammar.

4.3.4. Workflow Diagram of Grammar Provision

The workflow of our experimental setup follows a structured pipeline. The experimental workflow consists of two main phases: Candidate collection & ranking (offline phase) and User query & completion (online phase), as illustrated in Figure 14. The diagram below outlines this process:

The key steps in this workflow are:

1. Input Program: The user provides an incomplete code snippet.

Example of Prompt with Grammar-based Structural Candidate Guidance in Microsoft Small Basic Language

```
1: {1: Prog -> MoreThanOneStmt
2: ....
3: 60: Idxs -> [ Expr ] Idxs}
4: This is the incomplete Microsoft Small Basic programming
5: language code:
6: number = 100
7: While (number > 1)
8:                      TextWindow.WriteLine
9:                                                   '(Expr)'
10: Complete the '(Expr)' part of the code in the Microsoft Small Basic
11: programming language. Just show your answer in place of '(Expr)'.
```

Example of Prompt with Grammar-based Structural Candidate Guidance in C Language

```
1: {1: typedef_name -> NAME TYPE
2: ....
3: 335: list_eq1_typedef_declaration_specifier -> declaration_specifier
4: list_eq1_typedef_declaration_specifier}
5: This is the incomplete C programming language code:
6: int main(void)
7: {
8:     char s[1000];
9:     int i = 0;
10:    int loop = 1;
11:                 'while (expression) scoped_statement'
12: Complete the 'while (expression) scoped_statement' part of the code
13: in the C programming language. Just show your answer in place of
14: 'while (expression) scoped_statement'.
```

Figure 13: Prompt examples for Microsoft SmallBasic and C.

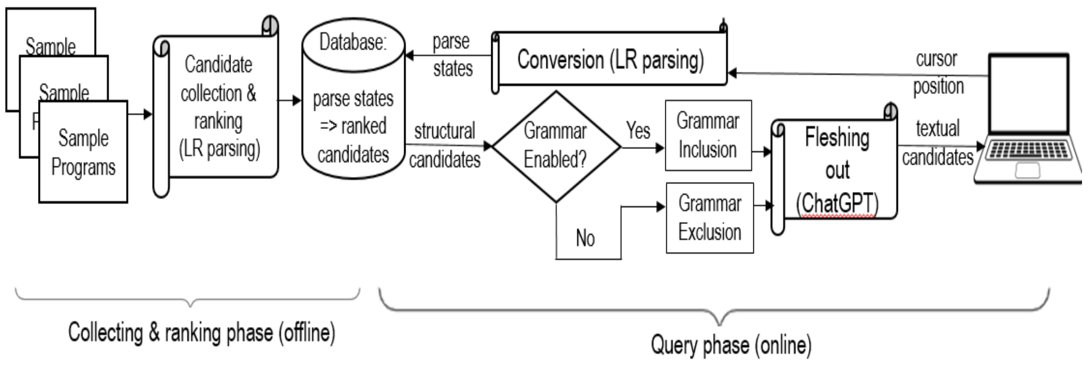


Figure 14: Workflow diagram grammar provision in our system.

2. Candidate Collection & Parsing: The system collects code samples and applies LR parsing technique to generate structural candidates.

3. Candidate Ranking: The parsed candidates are ranked based on their occurrence counts in the source programs. Such as a ranked list, 1: ‘ID = Expr’, 2: ‘ID.ID(Exprs)’, 3: ‘ID.ID = Expr’, 4: ‘Sub ID CRStmtCRs EndSub’, 5: ‘ID()’, 6: ‘ID Idxs=Expr’, 7: ‘If Expr Then CRStmtCRs MoreThanZeroElseIf’, 8: ‘For ID=Expr To Expr OptStep CRStmtCRs EndFor’.

4. Database Storage and Retrieval: Ranked candidates and parse states are stored in a database for future queries.

5. Query Processing: When a user requests code completion, the system transforms the current cursor position into a parse state and retrieves ranked structural candidates from the database.

6. LLM-Based Completion: The ranked structural candidates are used to compose completion prompts, which are sent to ChatGPT. The system then determines whether grammar inclusion or exclusion applies.

7. Grammar Processing: If grammar is enabled, grammar rules are applied to refine the generated completion; otherwise, the processing the structural candidates are used as is. The system then receives textual code completion suggestions.

8. Evaluation: The final generated code is assessed using SacreBLEU (n-gram similarity) and SequenceMatcher (character-level alignment) for accuracy.

4.3.5. Grammar Provision Experimental Results and Discussion

For two programming languages, we prepared separate datasets for training and testing. The training set consists of 3,701 Small Basic programs collected from its community and 412 C11 programs sourced from open-source software repositories. The testing set includes 27 Small Basic programs from its tutorial materials and 106 C11 programs from the exercises in the C programming language by Kernighan and Ritchie.

Prompts are generated for each cursor position using pre-identified LR structure candidates and provided to the LLM. The returned completions are compared to expected text candidates, and precision is measured. This process is repeated across all test programs, both with and without LR candidates, to evaluate their impact on completion accuracy. The table below presents the comparative results:

Table 4: Impact of grammar provision on code completion accuracy.

PLs	Experiment Types	SacreBLEU (%) Without Grammar	SacreBLEU (%) With Grammar	SequenceMatcher (%) Without Grammar	SequenceMatcher (%) With Grammar
Small Basic	WithIdealGuide	43.856	49.790	42.618	44.703
	WithinTop3Guide	44.773	45.733	43.532	43.897
	WithTop1Guide	37.905	38.524	36.775	37.097
C11	WithIdealGuide	25.173	28.368	26.537	28.658
	WithinTop3Guide	27.385	26.222	28.989	27.810
	WithTop1Guide	21.125	20.217	21.547	20.464

The results indicate a marginal improvement (nearly 1-6%) when grammar is provided, but the difference is not statistically significant. In some cases, such as C11 WithinTop3Guide and WithTop1Guide, providing grammar results in slightly lower accuracy. This suggests that simply appending grammar rules to the prompt may not be an effective approach. Possible explanations include:

1. **Grammar Redundancy:** LLMs may already internalize syntactic patterns from training data, rendering explicit grammar provision unnecessary.

2. **Prompt Length Complexity:** Adding grammar rules increases prompt length, potentially reducing focus on completion-specific tokens.

This additional experiment reveals that explicit grammar provision does not significantly enhance code completion accuracy. While minor improvements were observed in some cases, LLMs appear to inherently capture syntactic structures, making explicit grammar guidance redundant. Future research should explore alternative methods of incorporating grammar, such as structured in-context learning or constrained decoding. Another promising direction is to analyze whether grammar affects completion quality for specific types of syntactic structures, such as deeply nested expressions or function definitions.

4.4. A Comparative Analysis of ChatGPT 3.5, Llama 3, and other LLMs in LLM-Based Code Completion Using LR Parsing

Recent research has introduced LR-parsing-based approaches that generate structurally sound code suggestions by leveraging language syntax and open-source programs, but they often require manual refinement.

Meanwhile, recent advancements in large language models (LLMs) have significantly amplified predictive performance in code completion tasks. However, despite these progress, the impact of LLM selection on syntax-aware code generation remains underexplored. In this study, we conduct a comparative analysis of the performance and impact of two prominent LLMs, ChatGPT 3.5 and Llama 3, within an LR parsing-based code completion framework. Our experiments, evaluated using SacreBLEU and SequenceMatcher accuracy metrics, reveal that ChatGPT 3.5 achieves higher accuracy than Llama 3, underscoring the importance of selecting an appropriate LLM for enhanced code completion. These findings highlight the role of model selection in LLM-based code completion using LR parsing. Future research could extend this comparative analysis to a broader range of LLMs.

4.4.1. Different LLM in Code Completion

Recent growth in LLMs have transformed code completion strategies by leveraging vast training data to generate sophisticated, accurate, context-aware, and probabilistic predictions. Notably, models such as OpenAI's ChatGPT (3.5 Turbo) and Meta AI's Llama 3 have demonstrated substantial capabilities in generating syntactically correct and semantically meaningful code suggestions. However, despite these advancements, ensuring broader structural correctness, including adherence to language constraints and maintaining logical consistency, remains a challenge in LLM based code completion. More recently, in our previous study [2], we proposed LLMs, such as ChatGPT, with LR parsing-based structural candidates. This approach presents a promising solution for syntax-aware code completion, enhancing both productivity and usability. However, the comparative effectiveness of different LLMs in LR parsing-based code completion remains unexplored, leading to several key research questions:

1. Do different LLMs process LR-parsed structural candidates in code completion?
2. Which model, ChatGPT or Llama, demonstrates better performance in LR parsing-based code generation?

To address these research questions, this paper presents a comparative experimental study analyzing the performance of ChatGPT 3.5 and Llama 3 in LLM-based code completion using LR parsing. We evaluated their performance using SacreBLEU and SequenceMatcher accuracy metrics to determine the impact of model selection on syntax-aware code generation with LR parsing. Our key findings indicate that ChatGPT 3.5 outperforms Llama 3, achieving higher accuracy in syntactically correct code completions in experiments using Microsoft Small Basic and C languages. This reinforces the importance of selecting an appropriate LLM to optimize code completion within LR-structured candidates. This study makes the following key contribution:

• We assess the performance of different LLMs, such as ChatGPT 3.5 and Llama 3, in LR parsing-based code completion, highlighting the impact of model selection on accuracy.

• Our findings indicate that choosing the ChatGPT model over Llama provides advantage in achieving higher accuracy for LLM-based code completion using LR parsing.

The significance of this additional experiment lies in its contribution to understanding the role of LLM selection in syntax-aware code completion and the interplay between LR structural candidates and probabilistic language models. Our findings offer valuable insights into the application of LLMs in code completion, benefiting IDE developers and researchers aiming to optimize code completion strategies. Given these insights, future research could expand upon this study by investigating alternative methods for integrating structured parsing techniques.

4.4.2. Experiment with Different LLM in Code Completion Using LR-Parsing

To evaluate the performance of ChatGPT and Llama 3 in LR parsing-based code completion, we conducted experiments on two programming languages: Microsoft Small Basic (SB) and C11. These languages were chosen for their distinct syntax structures and their relevance in introductory programming and system-level programming, respectively. Our experimental workflow consists of two main phases: the collecting and ranking phase (offline) and the query phase (online) which is depicted in Figure 15. The offline phase serves as the foundation of this research and has been detailed in previous work [4], while the online phase is the primary focus of this study. The online phase provides an efficient code completion system by leveraging LR parsing and LLM-based candidate code generation.

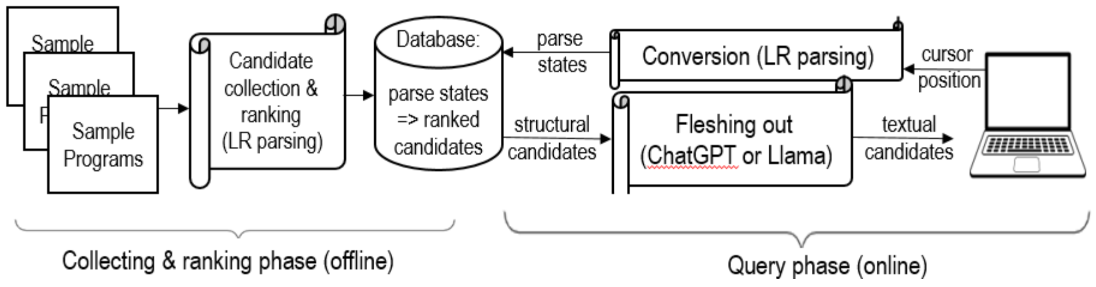

Sample
Sample
Sample Programs
Candidate collection & ranking (LR parsing)
Database: parse states => ranked candidates
parse states
structural candidates
Conversion (LR parsing)
Fleshing out (ChatGPT or Llama)
cursor position
textual candidates
Collecting & ranking phase (offline)
Query phase (online)

Figure 15: Overview of our system workflow

The experimental setup involved the following steps:

1. Candidate collection & ranking by LR parsing (offline): The training set, collection of samples undergoes LR parsing to generate structural candidates and then ranked in this phase. The training set consists of 3,701 Small Basic programs collected from its community and 412 C11 programs sourced from open-source software repositories.
2. Database storage: The parsed structural candidates are stored in a database, where they are ranked based on their occurrence frequency in the training set. The database [4] maintains a mapping between parse states and their ranked candidates, enabling efficient retrieval. For instance, in the database for State 0, several structural candidates are

stored along with their occurrence frequencies. Below, we present three ranked candidates for State 0 as an example:

```
[ID=Expr] : 422
[ID.ID= Expr] : 399
[ID.ID(Exprs)] : 246
```

Among these, one structural candidate is the correct choice for parse state 0, referred to as the ideal structural candidate (e.g., ID.ID(Exprs) in this example) for a specific cursor position in a test program. In this case, ID represents a terminal identifier, parentheses () are also terminal symbol, and Exprs is a non-terminal expression. Terminal symbols, also known as tokens, serve as the fundamental building blocks of a language, while nonterminal symbols, or syntactic variables, represent sets of strings composed of terminal symbols.

3. Parsing & candidate retrieval: Upon receiving a user query, the system converts the cursor position into a parse state using the LR parsing technique and retrieves the corresponding ranked structural candidates from the database. To evaluate the system, we used a testing set comprising 27 Small Basic programs from its tutorial materials and 106 C11 programs from the exercises in The C Programming Language by Kernighan and Ritchie.
4. Model invocation: The selected ideal structural candidates are used to construct completion prompts, which are first processed by ChatGPT

(gpt-3.5-turbo-0125) and subsequently by Llama 3 (llama-3.1-8b-instant). Since the correct (ideal) LR structural candidates are always provided to the LLM in this study, this represents an optimal approach, ensuring the highest possible accuracy in code completion.

5. Final code suggestions: Our system automatically constructs prompts with structured candidates, enabling LLMs to generate relevant textual suggestions. These generated suggestions are then finally presented to the user at the specific cursor position.
6. Evaluation metrics: The generated textual code completions were evaluated using two primary metrics, SacreBLEU (%) and SequenceMatcher (%):

4.4.3. Prompt Engineering for ChatGPT 3.5 and Llama 3

In the fourth step of our proposed system, prompts containing ideal structural candidates were constructed to generate completion responses for all possible cursor positions in the test set. Each prompt template consists of an incomplete code prefix, a selected ideal structural candidate for each parse state (cursor position), and an instruction to the LLM for performing code completion, as illustrated in Figure 16. This template is then fed into the ChatGPT 3.5 and Llama 3 models. Figure 17 presents example prompts for Microsoft Small Basic and C. Such prompts were crafted for all test programs during the online phase. Notably, our system remains language-agnostic by instantiating prompt templates with parameters tailored to each specific programming language. Figure 18 presents a comparison of ChatGPT 3.5 and Llama 3 responses to a prompt in Microsoft Small Basic. ChatGPT 3.5 achieves perfect scores with a SacreBLEU of 100% and SequenceMatcher similarity of 100%, exactly matching the expected output.

Prompt Template with Ideal Structural Candidate Guidance
This is the incomplete {Name of Programming Language} code: {Program Prefix} {Suggested Ideal Structural Candidate} Complete the {Suggested Ideal Structural Candidate} part of the code in the {Name of Programming Language}. Just show your answer in place of {Suggested Ideal Structural Candidate}.

Figure 16: Prompt engineering with ideal structural candidate

Example of Prompt with Ideal Structural Candidate Guidance in Microsoft Small Basic Language
This is the incomplete Microsoft Small Basic programming language code: number = 100 While (number > 1)   TextWindow.   'ID(Expr)' Complete the 'ID(Expr)' part of the code in the Microsoft Small Basic programming language. Just show your answer in place of 'ID(Expr)'.

Example of Prompt with Ideal Structural Candidate Guidance in C Language

This is the incomplete C programming language code:
int main(void)
{
    char s[1000];
    int i = 0;
    int loop = 1;
                'while (expression) scoped statement'
Complete the 'while (expression) scoped_statement' part of the code
in the C programming language. Just show your answer in place of
'while (expression) scoped_statement'.

Figure 17: Prompt examples for Microsoft SmallBasic and C

Comparative Evaluation of the Example in Figure 17 in Microsoft Small Basic Language Using ChatGPT 3.5 and Llama 3
ChatGPT 3.5 Response: WriteLine(number) Response Evaluation: SacreBLEU (%) score: 100   SequenceMatcher(%) similarity precision: 100 Llama 3 Response: TextWindow.WriteLine Response Evaluation: SacreBLEU (%) score: 33.333   SequenceMatcher(%) similarity precision: 43.902 Actual Textual Answer: WriteLine(number)

Figure 18: Comparative Evaluation of LLM Responses in SB

Here, it follows the candidate structure 'ID(Expr)'. In contrast, Llama 3 scores significantly lower, with a SacreBLEU of 33.33% and a SequenceMatcher similarity of 43.90%, only slightly following the structural candidate.

4.4.4. Comparative Experimental Results Analysis of ChatGPT and Llama 3 in Code Completion

Table 5 presents the comparative results of code completion performance between ChatGPT 3.5 and Llama 3 for SB and C11, using ideal structural candidate guidance using LR parsing technique for all test programs in terms of average SacreBLEU and SequenceMatcher.

The results indicate that ChatGPT consistently outperforms Llama 3 in code completion accuracy, emphasizing that the choice of LLM plays a crucial role in optimizing syntax-aware code generation.

Our key observations from the experimental results are as follows:

Table 5: Code completion experiment results with ideal structural candidate guidance using different LLMs.

Language	LLM Type	SacreBLEU(%)	SequenceMatcher (%)
SB	ChatGPT	43.856	42.618
	Llama 3	29.086	30.374
C11	ChatGPT	25.173	26.537
	Llama 3	15.290	16.913

Higher accuracy with ChatGPT: ChatGPT demonstrates significantly better performance than Llama 3, achieving an improvement of nearly 10% to 15% in both SacreBLEU and SequenceMatcher scores. Specifically, ChatGPT achieves a SacreBLEU improvement of 14.77 for Small Basic and 9.883 for C11. Similarly, the SequenceMatcher scores show an increase of 12.244 for Small Basic and 9.624 for C11. These results indicate that ChatGPT generates more precise and structurally aligned code completions compared to Llama 3 across different programming languages.

Significance of model selection: These results emphasize the importance of selecting the right LLM for syntax-aware code completion. While both models benefit from structured candidate guidance, ChatGPT's architecture appears better suited to incorporating explicit structural candidates into its predictions.

This additional experiment presents a comparative experimental report evaluating the performance of ChatGPT 3.5 and Llama 3 in LR parsing-based code completion. Our findings demonstrate that selecting ChatGPT offers a clear advantage in improving accuracy, as measured by SacreBLEU and SequenceMatcher scores. This highlights the importance of choosing the right LLM for optimizing syntax-aware code completion.

Future research should focus on expanding the comparative analysis to include a wider variety of LLMs to assess their effectiveness in different programming environments. Exploring the integration of additional fine-tuned LLMs could further refine accuracy. Additionally, assessing real-world developer usability and efficiency metrics will provide deeper insights into practical adoption. This research lays the groundwork for further advancements in selecting and refining LLMs for high-accuracy code completion.

4.4.5. Comparison with experiments using other LLMs

Previously, we experimented with ChatGPT and Llama 3.1. Now, we tested our methodology with Mistral and Llama 3.3 Versatile. The results are almost similar to the previous ChatGPT experiment. In this experiment, we focused only on Small Basic for performance investigation rather than experimenting with C language, as experiments with C showed about 22 to 18 percent lower performance than Small Basic. Experiments with Mistral and Llama 3.3 Versatile show that performance is not improving which is shown in Table 6. Among these four LLMs, ChatGPT outperforms the others. We attempted to experiment with the DeepSeek-R1 and Qwen QwQ-32B models; however, due to the nature of their responses, it was difficult to measure performance within our proposed system[③]. An example of such a response is provided below:

Parse State: 30 **Cursor Position:** 1 12

This is the incomplete Small Basic programming language code:
TextWindow.
'Identifier = Expression'
Complete the 'Identifier = Expression' part of the code in the Small Basic programming language. Just show your answer in place of 'Identifier = Expression'.

Received response:
<think>
Okay, the user provided an incomplete Small Basic code snippet with "TextWindow." followed by an incomplete line that needs to be filled in. The task is to complete the "Identifier = Expression" part. Since the user wants only the code snippet without explanations, I need to figure out a valid Small Basic statement that fits here. Looking at the context, since the code starts with "TextWindow.", maybe the next line is part of interacting with the TextWindow object. Common methods of TextWindow include WriteLine, Read, etc. But the user's instruction specifies an assignment, so perhaps they want something like storing the result of a TextWindow method into a variable. For example, if using Read(), which gets user input, then maybe something like: input = Text

Actual result: ForegroundColor = "Yellow"

Response Evaluation:
SacreBLEU score:1.463
SequenceMatcher similarity precision:0.0211

SB	ChatGPT	**43.856**	**42.618**
	Llama 3	29.086	30.374
	Mistral Saba 24B	42.099	40.520
	Llama 3.3 Versatile	40.776	38.670

5. Discussion

In this section, we discuss when our system produces high or low precision code completions by expanding on the best prediction and the

③ https://github.com/monircse061/Code-Completion-Work-with-Llama worst prediction for Microsoft Small Basic and C languages, by analyzing the causes of low precision results, and by finding that the LLM tends to predict better code completion when there is only one LR structural candidate for the cursor position.

5.1. Case Studies on Microsoft Small Basic

To demonstrate the effectiveness of guidance by an LR structural candidate, we discuss a case representing our system's best prediction example in the Microsoft Small Basic language experiment. In Figure 19, the top box shows the parse state and cursor position of a test program, followed by the next box, which provides the ranked candidate list for this specific parse state (6). The subsequent box shows the prompt given to ChatGPT. Lines 2 through 5 of this prompt contain the prefix code, extracted from the test program. Subsequently, line 6 displays a candidate structure, '= Expr', after the cursor position (5, 11). This syntactic structure is the top-ranked candidate in the candidate list and matches the actual structural candidate in this example. The next two lines (7 to 8) provide instruction to ChatGPT following the prefix code. After processing the prompt with the candidate structure guidance, the system responds with '= number / 2', which matches the code shown in the actual answer's box and aligns with the actual candidate structure. Based on our guidance, the system's response with this candidate pattern, reaveling the LR candidate guidance is highly accurate. This contributes to the precision for the SacreBLEU precision (1-gram) is 100%, as indicated in the response evaluation box, and other metrics also show satisfactory results. This example demonstrates that our candidate suggestion plays a crucial role in guiding ChatGPT's responses. Each terminal and non-terminal component contributes to achieving an accurate result from ChatGPT. This clearly indicates that the top-ranked candidate contributes to producing the correct response when it is included in the prompt to guide and control ChatGPT's prediction.

Parse State: 6   Cursor Position: 5 11

Candidate List: [1: '= Expr', 2: '.ID (Exprs)', 3: '[Expr]', 4: '.ID = Expr', 5: '( )', 6: '[Expr] Idxs', 7: ':']

Prompt

```
1:   This is the incomplete Microsoft Small Basic programming language code:
2:   number = 100
3:   While (number > 1)
4:         TextWindow.WriteLine(number)
5:         number
6:                     '= Expr'
7:   Complete the '= Expr' part of the code in the Microsoft Small Basic
8:   programming language. Just show your answer in place of '= Expr'.
```

ChatGPT's Response WithTop1Guide	**Actual Candidate**
= number / 2	= Expr

Actual Textual Answer

= number / 2

Response Evaluation

SacreBLEU score: 100.0

SequenceMatcher similarity precision: 96.0

Figure 19: Best prediction example in the Microsoft Small Basic experiment.

In the Microsoft Small Basic language, we also considered another case that represents one of the worst predictions example. In Figure 20, the top box outlines the parse state and cursor position information of a test program. Next box contains the ranked candidate list. Then, lines 2 of the prompt contain the prefix for this example, with a syntactic structure appearing in line 3. The selected candidate is the first-ranked candidate from the list, while the ideal (actual) candidate is the ninth-ranked candidate in this example. Given this prompt, the system responded with ‘number = 100 ID = number * 5’.

Parse State: 11 Cursor Position: 3 1	
Candidate List: [1: ‘ID = Expr’, 2: ‘ID.ID(Exprs)’, 3: ‘ID.ID = Expr’, 4: ‘Sub ID CRStmtCRs EndSub’, 5: ‘ID()’, 6: ‘ID Idxs=Expr’, 7: ‘If Expr Then CRStmtCRs MoreThanZeroElseIf’, 8: ‘For ID=Expr To Expr OptStep CRStmtCRs EndFor’, 9: ‘While Expr CRStmtCRs EndWhile’, 10: ‘ID:’, 11: ‘Goto ID’]	
Prompt	
1: This is the incomplete Microsoft Small Basic programming language code: 2: number = 100 3: ‘ID = Expr’ 4: Complete the ‘ID = Expr’ part of the code in the Microsoft Small Basic 5: programming language. Just show your answer in place of ‘ID = Expr’.	

ChatGPT’s Response WithTop1Guide
 	
number = 100
ID = number * 5
	Actual Candidate
	
	While Expr CRStmtCRs EndWhile

Actual Textual Answer	

While (number > 1)
  TextWindow . WriteLine(number)
  number = number / 2
EndWhile	

Response Evaluation	
SacreBLEU score: 37.5 SequenceMatcher similarity precision: 33.0	

Figure 20: Worst prediction example in the Microsoft Small Basic experiment.

However, the actual answer was different, as shown in the actual textual result’s box. In this case, the ChatGPT’s response was inappropriate with the first ranked candidate guidance, resulting in a precision of SacreBLEU score is 37.5. This example demonstrates that the first-ranked structural candidate was the wrong choice for recognition. To better understand this poor prediction, we analyzed the case in detail. In this example, the first-ranked candidate of the candidate list was selected in the prompt. Despite providing candidate information to the system, the response was mismatched as the ideal candidate is the ninth candidate, leading to low precision. This suggests that, in this case, the selected ranked candidate may have misdirected ChatGPT's response.

5.2. Case Studies on C

We also present a case where our system generated the best prediction in C. To provide a clearer understanding, an example of a C test program is depicted in Figure 21. In this example's prompt, lines 2 to 7 represent the program's prefix and cursor position (line:7, column:31), followed by line 8, which outlines the syntactic structure as the top-ranked candidate from the ranked candidate list, matching the actual LR structural candidate. After processing this prompt through ChatGPT, the system's response is: 'y', which exactly matches the actual textual answer. In this example's response evaluation, the 1-gram precision of the SacreBLEU score is 100%. This example further demonstrates that our LR syntactic structure guidance helps to produce better response from ChatGPT.

Conversely, we now present a worst prediction example involving C11 with candidate guidance. As shown in Figure 22, first box represents the parse state and cursor position, followed by a box containing the ranked candidate list information. The prompt fed to ChatGPT includes lines 2 to 9, which provide the prefix, and line 10, containing the LR candidate structure for the incomplete part of the code. Here, the LR structural candidate is 'CONSTANT', which is the top-ranked candidate from the candidate list. However, the actual structural candidate, '(type_name) cast_expression' was ranked sixth. Consequently, ChatGPT was misguided and generated '(SIZES[div - 1] > 1) ? SIZES[div - 1] : 0.1 * SIZES[div - 1]', as seen in

ChatGPT's response box. The correct answer is '(float)rem', as shown in the actual answer box. As a result, the SacreBLEU (1-gram) score is 7.6, and other metrics also indicate low precision.

Parse State: 429 Cursor Position: 7 31

Candidate List: [1: 'NAME VARIABLE', 2: 'CONSTANT', 3: 'STRING_LITERAL', 4: '(expression)', 5: '(type_name) cast_expression', 6: '&', 7: 'sizeof unary_expression', 8: 'sizeof(type_name)', 9: '*', 10: '-', 11: '--unary_expression', 12: '!', 13: '++unary_expression', 14: ' builtin_va_arg(assignment_expression, type_name)']

Prompt

```
This is the incomplete C11 programming language code:
int main(void)
  {
    int x = 2, y = 3;
    printf("x: %d, y: %d\n", x, y);
    int temp; temp = x; x = y; y = temp;
    printf("x: %d, y: %d\n", x,
                              'NAME VARIABLE'
   Complete the 'NAME VARIABLE' part of the code in the C11 programming
   language. Just show your answer in place of 'NAME VARIABLE'.
```

ChatGPT's Response WithTop1Guide	Ideal (Actual) Candidate
y	NAME VARIABLE

Actual Textual Answer

y

Response Evaluation

SacreBLEU score: 100.0
SequenceMatcher similarity precision: 1.00

Figure 21: Best prediction example in the C experiment.

5.3. Low Precision Analysis

The relatively low precision observed in our experimental results can be attributed to several issues identified in the Microsoft Small Basic and C test programs, particularly those involving numerical values and inconsistent formatting. These programs have discrepancies in numerical values, as seen in cases where our system's output differs from expected results (e.g., values like "10, 50, 100, 150" versus "10, 100, 100") which is illustrated in Figure 23. Such discrepancies result in 1-gram SacreBLEU precision scores that are low, as the scoring mechanism is sensitive to these variations. Furthermore, inconsistent use of quotation marks differences adding additional challenges, as they interfere with SacreBLEU's ability to accurately match expected outputs. These formatting and numerical inconsistencies ultimately reduce the precision of code completion.

In Figure 23, in the first example of numerical inconsistency, the system's selected structural candidate is ', MoreThanOneExpr'. The system responds with the candidate ', 10, 50, 100, 150', while the actual result is ', 10, 100, 100'. This mismatch leads to a SacreBLEU 1-gram precision score of 62.5, highlighting the system's partial success in capturing the correct values but failing to fully align with the expected numerical output. In the second example, the selected candidate is 'Number'. The system predicts the number 10, which does not match the actual answer of 70, resulting in a zero SacreBLEU score. Many testing programs for such cases exist in many Microsoft Small Basic test program, and these are a primary cause of low precision in our experiment. In the third example of numerical inconsistency, the selected candidate contains an 'Expr' representation of a number. Although the actual result was 600 instead of the expression, the system generated the number 150, leading to 80.0 in 1-gram precision for the SacreBLEU score. In the first example of inconsistency in quotation marks, the system's selected candidate is'EndFor', and the system's response matches this exactly as 'EndFor'.

Parse State: 246 Cursor Position: 9 36

Candidate List: [1: ‘CONSTANT’, 2: ‘(expression)’, 3: ‘NAME VARIABLE’, 4: ‘sizeof unary_expression’, 5: ‘sizeof (type_name)’, 6: ‘(type_name) cast_expression’, 7: ‘*’, 8: ‘!’, 9: ‘++ unary_expression’, 10: ‘&’]

Prompt

```
1:   This is the incomplete C11 programming language code:
2:   size_t rem = 0;
3:   while (size >= 1024 && div < (sizeof SIZES / sizeof *SIZES))
4: {
5:   rem = (size % 1024);
6:   div++;
7:   size /= 1024;
8:   }
9:   printf("%6.1f%s ", (float)size +
10:                               ‘CONSTANT’
11: Complete the ‘CONSTANT’ part of the code in the C11 programming language.
12: Just show your answer in place of ‘CONSTANT’.
```

ChatGPT’s Response WithTop1Guide	Ideal (Actual) Candidate
(SIZES[div-1]>1)?SIZES[div-1] :0.1*SIZES[div-1]	(type_name) cast_expression

Actual Textual Answer

(float)rem

Response Evaluation

SacreBLEU score: 7.6
SequenceMatcher similarity precision: 6.0

Figure 22: Worst prediction example in the C experiment.

However, the actual result is EndFor (without quotation marks). This subtle difference leads to a SacreBLEU 1-gram precision score of 0.0, despite the system generating the correct content. In the second example, the system's selected candidate is ‘)’, which is also returned exactly by the system as ‘)’. The actual result, however, is) (without quotation marks). Once again, this mismatch in formatting results in a SacreBLEU 1-gram precision score of 0.0, even though the numerical content is correct. These examples illustrate that minor formatting discrepancies, such as the inclusion or exclusion of quotation marks, can significantly impact evaluation metrics. This emphasizes the importance of designing systems that account for such inconsistencies to better reflect the true performance of the model.

First Example of Numerical Inconsistency	
Our system’s selected structural candidate	: MoreThanOneExpr
System’s response with structural candidate	: , 10, 50, 100, 150
Actual textual answer	: , 10, 100, 100
SacreBLEU’s 1-gram precision	: 62.5
Second Example of Numerical Inconsistency	
Our system’s selected structural candidate	: Number
System’s response with structural candidate	: 10
Actual textual answer	: 70
SacreBLEU’s 1-gram precision	: 0.0
Third Example of Numerical Inconsistency	
Our system’s selected structural candidate	: ID.ID = Expr
System’s response with structural candidate	: GraphicsWindow.Height = 150
Actual textual answer	: GraphicsWindow.Height = 600
SacreBLEU’s 1-gram precision	: 80.0
First Example of Inconsistency in Quotation Marks	
Our system’s selected structural candidate	: EndFor
System’s response with structural candidate	: ‘EndFor’
Actual textual answer	: EndFor
SacreBLEU’s 1-gram precision	: 0.0
Second Example of Inconsistency in Quotation Marks	
Our system’s selected structural candidate	:)
System’s response with structural candidate	: ‘)’
Actual textual answer	:)
SacreBLEU’s 1-gram precision	: 0.0

Figure 23: Our system’s low precision results analysis.

5.4. Ranking in Candidates List and the Importance of the Top-Ranked Candidate

In our experiment, for each parse state in the test set, the candidate collection and ranking algorithm [4] generates a list of ranked LR structural candidates along with their occurrences in the sample programs. Although the top-ranked candidate is generally the most likely candidate for the prompt, there are some instances where the second, third, or even lower-ranked candidate is selected. Here, ranking is based on the frequency of a candidate in the sample programs. Sometimes the lower-ranked structural candidate is the correct syntactic structural candidate.

Notably, it was observed that when the candidate list contained only one candidate, ChatGPT responded correctly in the majority (roughly 90%) of the cases. This suggests that, with minimal ambiguity in the candidate options, the system is more likely to provide an accurate prediction. The absence of additional competing candidate suggestions allowed ChatGPT to focus solely on the most likely solution, leading to higher precision. This finding highlights the crucial benefits of having a focused candidate set, especially in scenarios where the context is clear and unambiguous. For example, in Figure 24, the prediction precision with a single candidate is 100%. This scenario is common in many test programs in Microsoft Small Basic and can also be observed in the C programming language. In such cases, a single candidate tends to provide a more accurate response than a list with multiple candidates, reducing ambiguity.

```
Parse State: 8    Cursor Position: 7 5
```

Candidate List: [1: 'ID = Expr To Expr OptStep CRStmtCRs EndFor']	
Prompt	
This is the incomplete Microsoft Small Basic programming language code: For i = 1 To 5 TextWindow.Write("User" + i + ", enter name: ") name[i] = i EndFor TextWindow.Write("Hello") For 'ID = Expr To Expr OptStep CRStmtCRs EndFor' Complete the 'ID = Expr To Expr OptStep CRStmtCRs EndFor' part of the code in the Microsoft Small Basic programming language. Just show your answer in place of 'ID = Expr To Expr OptStep CRStmtCRs EndFor'.	

System's Response with Guidance
i = 1 To 5
TextWindow.Write(name[i] + ", ")
	System's Response without Guidance
	i = 1 Endfor

Actual Textual Answer	
i = 1 To 5 \n TextWindow . Write (name [i] + ", ") \n EndFor	
Result Evaluation	
SacreBLEU precision with guidance : 100.0 SacreBLEU precision without guidance : 33.34	

Figure 24: Single top-ranked candidate contribution to prediction results in Microsoft Small Basic.

5.5. Performance Comparison with Commercial Tools (JetBrains IntelliJ and GitHub Copilot)

In this section, we compare our approach with leading commercial code completion tools, namely JetBrains IntelliJ's Full Line Code Completion (FLCC) and GitHub Copilot's Neural Code Completion. We focus on their design methodologies, capabilities, and reported performance metrics.

JetBrains' FLCC employs Transformer-based model to generate syntactically valid, multi-token code suggestions. Their model operates entirely on the local machine and is integrated into IDEs such as IntelliJ and PyCharm, without requiring cloud services or GPU support. According to Semenkin et al. [47], FLCC achieves an 18% acceptance rate for C code, but this metric pertains only to single-line completions. In contrast, our system supports partial, full, and multi-line completions (typically spanning 2–3 lines) and achieves approximately 29% SacreBLEU accuracy on C11 code during evaluation. This broader scope suggests that our approach can offer more contextually complete suggestions compared to FLCC's single-line focus. Our method is fundamentally different in design. While FLCC relies on Transformer-based neural networks for multi-token prediction, our system utilizes LR parsing techniques to drive completion. This structural foundation allows for more grammar-aware and deterministic completions, particularly beneficial in statically typed languages such as C.

GitHub Copilot and similar tools like TabNine perform neural code synthesis by predicting multiple tokens or even entire code blocks based on the current coding context. These tools are designed to improve developer productivity by generating relevant and often extensive code suggestions. As reported by Ziegler et al. [48]. Copilot achieves a 27% acceptance rate across various programming languages. Similar to FLCC, this metric emphasizes user engagement and usability rather than exact syntactic or semantic correctness. Our work diverges from Copilot in its objective. Rather than aiming for broad, full-function code generation (e.g., generating an entire DFS or factorial implementation), our focus is on fine-grained, controlled code completion within a structured syntactic framework. This provides more predictable and verifiable results, which can be especially valuable in contexts requiring correctness and maintainability.

It is important to acknowledge the different evaluation methodologies adopted by commercial tools. JetBrains and GitHub Copilot primarily assess their systems based on acceptance rate, defined as the percentage of presented completions that are selected or used by developers, rather than strict correctness. This reflects their emphasis on usability and seamless integration within real-world development workflows. While our evaluation currently focuses on syntactic and semantic correctness, specifically top-3 accuracy, we recognize the value of complementary usability metrics such as acceptance rate. In future work, we plan to incorporate these broader measures to enable more comprehensive and practical comparisons. We also recognize that our tool, while academically motivated, is not yet at a commercial level in terms of user experience or productivity. Nonetheless, it introduces a fundamentally different approach that emphasizes structural soundness and multi-line, context-aware code completions.

The importance of the first suggested candidate is well recognized in practical development settings, where developers often prefer immediate relevance over navigating multiple options. To address this concern, we analyzed the top-1 and top-3 correctness of our completions. Under the WithTop1Guide condition, the top-1 structural candidate resulted in textually correct suggestions in 10.8% of cases for SmallBasic and 3.0% for C11. When considering the top three suggestions (WithTop3Guide), the correct textual result appeared in 14.0% of cases for SmallBasic and 4.4% for C11. In the WithIdealGuide condition, which provides a more optimal structural context, these rates increased to 17.9% and 9.7%, respectively. These results highlight the critical role of structural candidate quality in guiding the language model toward correct completions. Although the current accuracy is limited by the relatively small and domain-specific nature of our dataset, we expect substantial improvements as the structural database is expanded and refined in future work.

5.6. Impact of Grammar on LLM-Based Code Completion

Although this study adopts the same grammar as used in prior work, it did not analyze how the choice of grammar affects the performance of LLM-based code completion. It is possible to define the same programming language using more than one grammar, and depending on the language's structural characteristics, the grammar can exhibit varying degrees of complexity. The method proposed in this paper depends on the grammar. Consequently, if the grammar changes, the code completion results produced by the LLM might also change.

To discuss this, let us define the tightness of a grammar as its ability to describe the target language precisely, leaving no ambiguity to be resolved later (e.g., during semantic analysis).

For example, SmallBasic can be said to have a relatively tighter grammar than C11. SmallBasic's grammar includes many keywords by default, and the distinction between the lexer and the parser is clear, which eliminates the need for additional semantic analysis. In contrast, consider C11: in an expression like a * b, if a is a userdefined type, then * b is interpreted as pointer direct_declarator, whereas if a is a variable, * b is parsed as multiplicative_operator cast_expression. This ambiguity cannot be resolved by the grammar alone but requires cooperation between the lexer and the parser via semantic analysis. Therefore, C11's grammar is considered less tight than that of SmallBasic. The grammar of Haskell, a functional programming language, contains relatively few keywords. For instance, a function call is simply written as e1 e2 ... ek without parentheses. When the grammar rules of Haskell include more terminals or keywords like the parentheses, the grammar can offer more distinguishable structural contexts. Furthermore, parsing Haskell programs necessitates interaction between the lexer and the parser to resolve declaration groups based on indentation rules, suggesting that Haskell's grammar is less tight.

A language with a tight grammar can have its syntactic structure unambiguously determined from the program text alone, without any semantic analysis information (such as a symbol table). For languages with less tight grammars, the semantic analysis performed during parsing is transient and temporary, existing only in memory. As a result, the LLM is trained solely on the program text, without direct access to this semantic analysis information. If the LLM can nonetheless reliably infer the structure from program texts written in languages with less tight grammars, it implies that it must have undergone a training process that approximated a more complex form of semantic analysis.

Future research is needed to determine whether there is a significant difference in the accuracy of LLM-based code completion when comparing tight grammars to less tight ones.

5.7. On Top-1 Correctness as a Metric

So far, we have focused on how much LLM-based code completion improves when guided by LR-derived structural candidates, compared to when such guidance is absent. However, from a programmer's perspective, a critical concern is how often the first suggested candidate is actually 100%-correct. If a user has to scroll through a list to find the right suggestion, it might be faster to simply type it manually. Therefore, top-1 accuracy becomes particularly relevant from a usability standpoint—especially for programmers who prefer typing over navigating long suggestion lists. To evaluate this, we analyzed the results of our existing experiments:

• Under the WithTop1Guide condition, the top-1 structural candidate led to a textually correct suggestion in 10.8% of cases for SmallBasic, and 3.0% for C11.

• Under the WithTop3Guide condition, the correct textual suggestion appeared within the top 3 in 14.0% of cases for SmallBasic, and 4.4% for C11.

• With the WithIdealGuide setting, these rates increased significantly to 17.9% for SmallBasic and 9.7% for C11.

As discussed earlier in Section 5.3, several factors contributed to these relatively low numbers, such as numerical mismatches, inconsistent use of quotation marks, and other minor textual deviations.

Another contributing factor is the limited size and diversity of our current codebase, which was constructed from a relatively small set of open-source projects. We expect that scaling to a larger and more diverse corpus will improve the precision of both structural and textual suggestions.

It is also worth noting that previous studies on commercial tools—such as JetBrains IntelliJ [47] and GitHub Copilot [48]—have not used exact correctness of suggestions as their primary evaluation metric. Instead, they assess usability through metrics such as acceptance rates and completion ratios within interactive environments.

That said, we acknowledge the value of measuring the exact correctness of top-1 suggestions, and we plan to include this metric in future work.

5.8. Other Limitations of our system

In our experiments with Small Basic and C, we measured the response time from issuing a query to receiving a response at each cursor position, as documented in the experimental results file highlighted in the previous footnote. However, we did not measure the response time for intermediate steps, such as fetching structural candidates before generating the final response. Additionally, while response time was recorded during the experiments, it was not measured within the implementation of the VS Code plugin or extension. Compared to commercial tools, the acceptance rate is a key metric for evaluating the performance of code completion plugins or extensions. In this study, however, we focused on precision, specifically the accuracy of completions on a moderately sized test set.

6. Implementation: A VSCode Extension Based on Our Approach

So far, this study has focused on demonstrating that the proposed method can assist LLMs in achieving more precise code completion. Now, we aim to show that this method can be implemented in an actual code editor for practical use.

Visual Studio Code is an IDE launched by Microsoft. It ranked first in the popularity survey with a usage rate of 74.48% and is widely used by programmers around the world. VSCode offers a variety of tools as extensions to help programmers develop projects smoothly. Overall, VSCode extensions for programming languages typically require users to type at least one character before providing a code suggestion function that displays a list of variable and method (or function) names containing the entered characters. This indicates that word-based code completion, which relies on the characters input by the user, becomes available only after those characters are typed. Moreover, since VSCode extensions assume prior knowledge of the programming language's grammar, they do not offer an optimal coding experience for all users. When writing code in VSCode, most developers rely on IntelliSense, a feature that provides functionalities such as code completion, parameter information, quick info, and debugging support for various programming languages. While VSCode extensions offer various features, the extension for Microsoft Small Basic is limited to syntax highlighting and basic snippets, lacking advanced capabilities like code completion. Similarly, the C extension requires typing many characters before offering suggestions, highlighting the need for more robust features in both. Furthermore, Copilot, an AI-powered coding assistant, provides automatic code suggestions but has significant limitations. It requires users to write comments or prompts describing the desired functionality, and if these comments are unclear or incomplete, the generated code may fail to accurately implement the intended features. We believe that our method can compensate these limitations.

This study developed two new VSCode extensions for Microsoft Small Basic and C, incorporating a code completion function to enhance their capabilities and improve productivity. These extensions provide code suggestions without requiring users to input any initial characters. Additionally, they assist users unfamiliar with the programming language's grammar by enabling easy code writing through the code completion feature. The VSCode extensions for Microsoft Small Basic, a popular language for beginners, and C, one of the most widely used languages, offer syntax-structure-based code completion instead of traditional word-based completion. By incorporating generative AI for code suggestions, the extensions aim to provide context-aware, syntax-compliant code completions. The project materials and the code for the two VSCode extensions are publicly available in the GitHub repositories④.

Our implementation of the extensions, based on Microsoft Small Basic and C11 (both utilizing LR grammars), is built using the YAPB parser builder tool [16]. To develop a language-specific syntax completion tool, developers

④ https://github.com/monircse061/Small-Basic-Extension, https://github.com/monircse061/C-Extension only need to provide a lexer and parser specification compatible with YAPB's interface, as demonstrated in [18, 30, 31]. Once the parser is defined, the tool incorporates a candidate collector and a cursor position converter, both leveraging LR parsing. We created syntax completion systems for these two languages, with supplementary materials—including LR grammars, LALR automata, learning and test programs, and evaluation results available in [3]. Our system uses the Small Basic and C11 grammars from [5, 13]. Although modifying the grammars, as suggested in [31], could enhance completion quality, we opted not to modify the grammars for this evaluation.

The workflow of the two extensions is illustrated in Figure 25. The VSCode code completion extension operates as follows:

1. Initiating the Code Completion Feature: At any cursor position, the user can activate the code completion feature by pressing the shortcut key (Ctrl + c), which triggers the code completion event.

2. Sending Cursor and Code Information: The user's written code and current cursor position are then sent to the Microsoft Small Basic or C parser server.

3. Deriving Parse State: The parser server processes the current cursor position and derives a parse state using a key algorithm based on LR parsing.

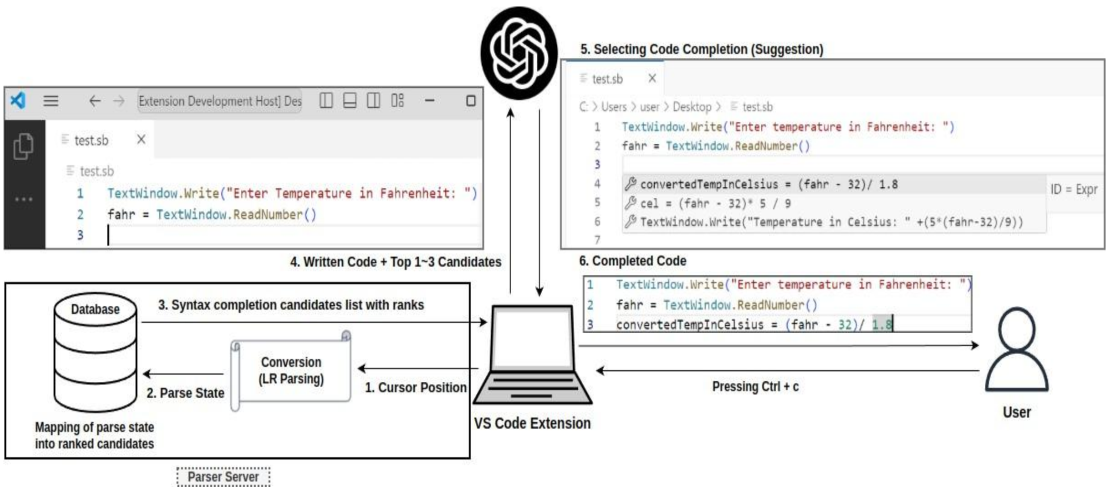
4. Retrieving Syntax Structure Candidates: Based on the derived parse state, the server refers to a database where parse states are mapped to corresponding syntax structure candidates. It returns a list of possible syntax structures, ranked according to the frequency of each specific candidate in the source code. We selected the top 1–3 candidates, as in our previous work [4], where it was reported that, on average, 2.15 down key strokes were required to locate the correct answer among the candidates.

5. Integrating Generative AI: For each ranked structural candidate, the user's code, along with the candidate information, is sent as a prompt to a generative AI model (ChatGPT) to receive a response as a suggestion.

6. Providing Suggestions to the User: Our VSCode extension displays a pop-up menu with structural candidates and code completion suggestions. The user can select the desired suggestion from the list.

7. Inserting Completed Code: The selected code suggestion is inserted into the VSCode editor at the user's current cursor position.

Figure 26 depicts two screenshots of the VSCode editor where users are editing code in Microsoft Small Basic with the extension activated. At a specific cursor position, users can enable the code completion feature to receive suggestions generated by the extension. In this example, at line 3, the programmer intends to complete this line using the extension. After pressing the designated hotkey, a list of three suggestions appears. Upon selecting one of the suggestions, the chosen candidate is inserted into the editor. Note that based on our strategy WithinTop3Guide, whose effectiveness was demonstrated in Figure 10 and Figure 11, the VSCode plugin has been configured to display three textual candidates in the menu by default.



```
TextWindow.Write("Enter Temperature in Fahrenheit: ")
fahr = TextWindow.ReadNumber()
```

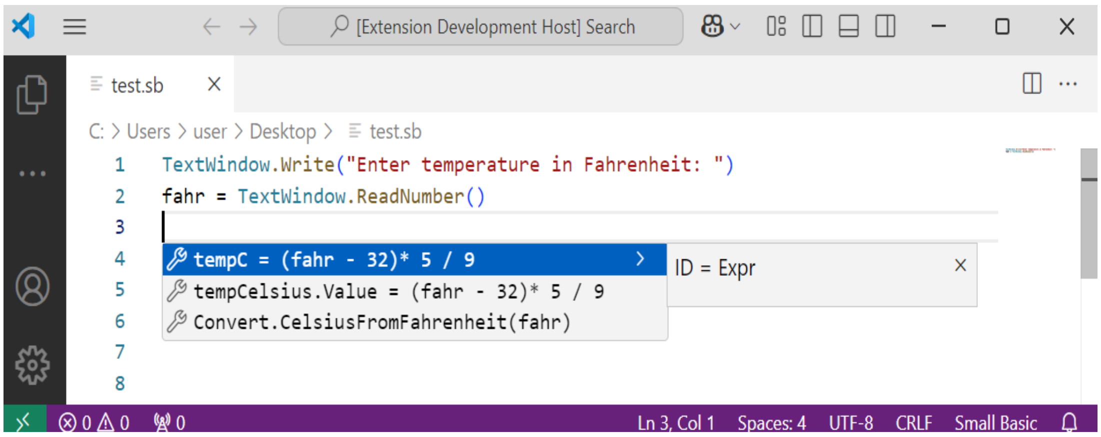
```
TextWindow.Write("Enter temperature in Fahrenheit: ")
fahr = TextWindow.ReadNumber()

convertedTempInCelsius = (fahr - 32)/ 1.8
cel = (fahr - 32)* 5 / 9
TextWindow.Write("Temperature in Celsius: " +(5*(fahr-32)/9))
```

```
TextWindow.Write("Enter temperature in Fahrenheit: ")
fahr = TextWindow.ReadNumber()
convertedTempInCelsius = (fahr - 32)/ 1.8
```

Figure 25: VSCode extension workflow diagram.

We also provide candidate information upon user request, displayed by clicking on the '>' sign. Users can click the '>' sign for the first candidate and then scroll down to view subsequent candidates. In the accompanying figure, an additional menu shows a candidate list containing an ID and an expression (Expr), offering the user a preview of the potential suggestions. The ID represents an identifier, which may correspond to a variable or function name, while the Expr denotes an expression that can further decompose into one or more sub-expressions. The primary purpose of displaying the candidate structure is to provide contextual hints, particularly for intermediate or advanced users. These hints offer insight into the syntactic structure of the suggestions, allowing users to better understand and evaluate them. This feature ultimately enhances the code-writing experience by fostering a deeper comprehension of the provided suggestions.



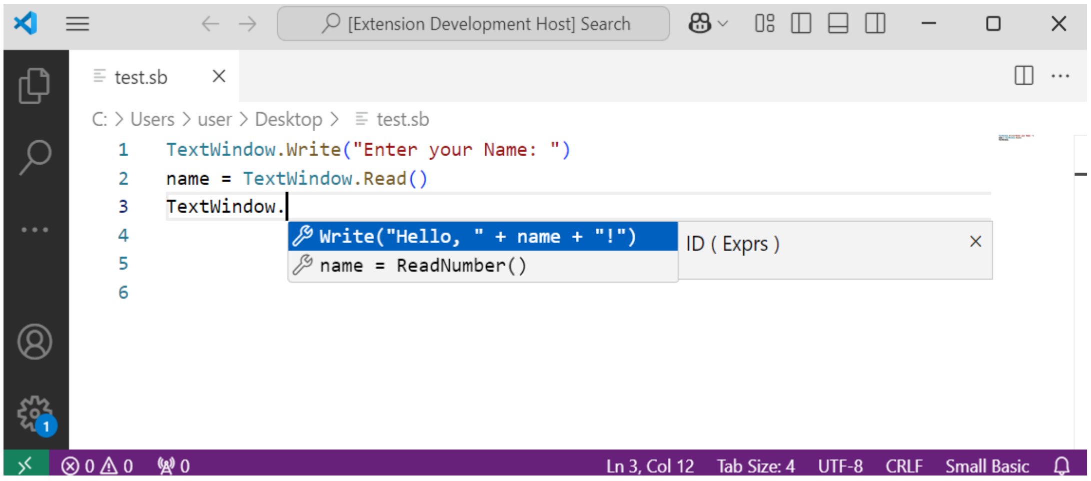
```
TextWindow.Write("Enter temperature in Fahrenheit: ")
fahr = TextWindow.ReadNumber()

```

```
tempC = (fahr - 32)* 5 / 9
tempCelsius.Value = (fahr - 32)* 5 / 9
Convert.CelsiusFromFahrenheit(fahr)
```

[Extension Development Host] Search
test.sb
C: > Users > user > Desktop > test.sb
ID = Expr
Ln 3, Col 1 Spaces: 4 UTF-8 CRLF Small Basic

(a) Microsoft Small Basic programming VSCode extension example-1.



```
TextWindow.Write("Enter your Name: ")
name = TextWindow.Read()
TextWindow.
```

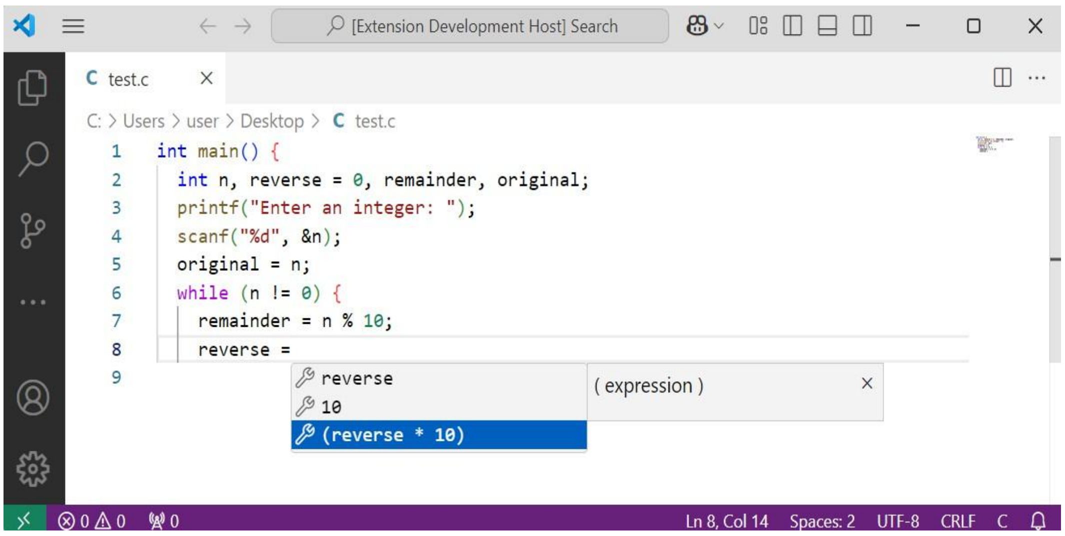
```
Write("Hello, " + name + "!")
name = ReadNumber()
```

[Extension Development Host] Search
test.sb
C: > Users > user > Desktop > test.sb
ID (Exprs)
Ln 3, Col 12 Tab Size: 4 UTF-8 CRLF Small Basic

(b) Microsoft Small Basic programming VSCode extension example-2

Figure 26: Examples of Microsoft Small Basic programming in the VSCode extension.

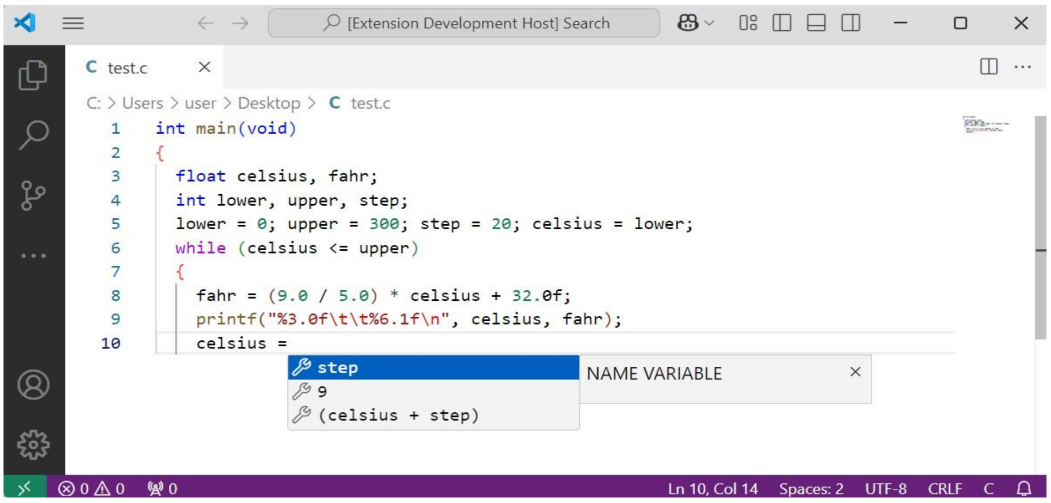
When editing code in the C programming language, users can utilize a similar feature provided by the C extension. Figure 27 presents two example programs in C that a user is currently editing. At line 8 and 10, the user



```
int main() {
  int n, reverse = 0, remainder, original;
  printf("Enter an integer: ");
  scanf("%d", &n);
  original = n;
  while (n != 0) {
    remainder = n % 10;
    reverse =
```

[Extension Development Host] Search
test.c
C: > Users > user > Desktop > test.c
reverse
10
(reverse * 10)
(expression)
Ln 8, Col 14 Spaces: 2 UTF-8 CRLF C

(a) C programming VSCode extension example-1.



```
int main(void)
{
  float celsius, fahr;
  int lower, upper, step;
  lower = 0; upper = 300; step = 20; celsius = lower;
  while (celsius <= upper)
  {
    fahr = (9.0 / 5.0) * celsius + 32.0f;
    printf("%3.0f\t\t%6.1f\n", celsius, fahr);
    celsius =
```

[Extension Development Host] Search
test.c
C: > Users > user > Desktop > test.c
step
9
(celsius + step)
NAME VARIABLE
Ln 10, Col 14 Spaces: 2 UTF-8 CRLF C

(b) C programming VSCode extension example-2.

Figure 27: Examples of C programming in the VSCode extension.

activates the extension feature by pressing the designated hotkey. This action displays a list of suggestions that are likely to follow the prefix on that line. Upon selecting an item from the list, the chosen suggestion is dynamically inserted into the editor.

7. Related Work

Various studies on code completion have been conducted to date. These include approaches using large codebases, machine learning, parsers, attribute grammars, and more. In this section, we discuss these studies from different viewpoints, including environments, rankings, and guiding LLMs.

7.1. Environments and Editors Providing Code Completion

One of the most widely used code editors is Visual Studio Code (VSCode). VSCode supports IntelliSense [11], which offers identifier completion, quick information on library functions and methods, and other features for various languages, including JavaScript. Additionally, programmers can enhance functionality by installing extensions, including our own tool, which is provided as a VSCode extension.

Code completion can be triggered by pressing Ctrl+Space or a language-specific character, such as a dot (.) in JavaScript. The quick info feature displays documentation, including type information, to assist developers. Other editors, such as Eclipse, Emacs, Vi, and Atom, also support similar features. In these editors, users can extend functionality by writing or installing scripts.

Recently, Microsoft introduced the Language Server Protocol (LSP) [21] to standardize code completion and other language features across multiple editors. LSP facilitates communication between editors such as VSCode and language servers for different programming languages. Once a feature is implemented as a language server using LSP, it becomes accessible in any editor that supports the protocol.

7.2. State-of-the-Art Techniques in Code Completion

IntelliSense [11] is widely used in VSCode, providing essential features such as identifier completion, syntax error checking, and type information display for functions and methods. More recently, IntelliCode Compose [34], developed by Svyatkovskiy et al. at Microsoft, extends IntelliSense capabilities. Available as a VSCode extension, IntelliCode Compose leverages GPT-C, a variant of OpenAI's GPT-2 [26], trained on a vast dataset of program source code. It generates syntactically correct language constructs such as statements containing local variables, method names, and keywords for languages including C#, C++, and JavaScript.

Their approach relies on the probability distribution of token sequences of fixed length, computing their frequencies in a large codebase. The most probable sequence is suggested to the user, based on a model trained on extensive source code. A key distinction between their system and ours is that while IntelliSense and IntelliCode Compose generate sequences of tokens (terminal symbols), our approach generates sequences of both terminal and non-terminal symbols, with non-terminals subsequently expanded into concrete lexemes.

Another widely used tool is GitHub Copilot, which provides code completion and generation for languages such as Python and JavaScript. Unlike IntelliSense, Copilot extends beyond single-line suggestions, generating entire code blocks or function bodies from natural language comments. It is powered by OpenAI Codex, based on GPT-3 and further trained on millions of GitHub repositories.

Similarly, CodeT5+ [37] is a recent LLM designed for understanding and generating code. It supports code completion tasks, including generating full function/method bodies from function names, signatures, or natural language descriptions.

Our study investigates how syntactic structure in completion candidates influences code generation in the model gpt-3.5-turbo-0125. In this respect, our work complements prior LLM-based approaches, including the tools mentioned above, by exploring how structured syntax information can enhance code completion accuracy.

7.3. Parser-based Code Completion

Several studies have explored parser-based code completion, covering both identifier and syntax structure suggestions that comprise terminal and non-terminal symbols.

One such study [35] employed the ANTLR parser generator [24], developed by Parr, to implement code completion. This approach enables completion functionality to be derived directly from an ANTLR syntax description. Some aspects of their methodology align with ours, such as syntax-based generation, leveraging internal parser information, and triggering code completion upon detecting a syntax error. However, a key distinction lies in the nature of the completion candidates: while their approach suggests individual tokens, ours generates sequences of symbols, with tokens expanded into lexemes.

Other studies have explored syntax completion using parsers. Rekers et al. [28] introduced a substring parser based on GLR parsing [36]. Their parser, designed for a language L defined by any context-free grammar, processes an input string s and constructs a parse tree for a sentential form vsw in L, where v and w represent completion candidates for s. Their approach closely aligns with ours; however, their completions are strictly sentential forms, whereas ours allow prefixes of sentential forms. This flexibility is particularly useful when handling partial programs where the cursor is deeply nested. Additionally, their study was limited to the Pascal language, lacked candidate ranking, and did not expand tokens into lexemes.

Another study [31] explored the use of an LALR parser to determine syntactic completion candidates from a valid program prefix. This research employed the LALR(1) parser generator YAPB [17], which provides access to internal parser details. However, like the previous studies, it did not expand tokens into lexemes.

7.4. Using LLMs and Machine Learning for Code Completion

Beyond the studies mentioned in Section 7.2, numerous studies have explored code completion using LLMs and machine learning.

Nguyen et al. [23] combined program analysis with a language model to complete partially written statements or suggest a statement that logically follows a completed one, a technique they call statement completion. They argue that the generated code should be natural, a concept introduced by Hindle et al. [9], who observed that software code exhibits natural patterns similar to human language. Their study on Java projects demonstrated that code is even more regular than spoken English. To generate both natural and valid completion candidates, they first use an LLM to produce natural template candidates, then apply program analysis to concretize them, and finally rank the resulting candidates. While their approach integrates LLMs with programming language techniques, similar to ours, our method specifically uses an LLM to concretize non-terminal symbols in candidates generated via LR parsing. Additionally, our completion unit is not restricted to statements and may span multiple statements, depending on the grammar of the language.

Gabel et al. [7] were among the first to observe the regularity of software code. While there are infinitely many syntactically valid statements, the set of practically useful statements is much smaller, possibly even finite. To explore this conjecture, they analyzed 6,000 projects totaling 430 million lines of C, C++, and Java code.

Liu et al. [19] proposed a non-autoregressive model that predicts multiple candidate lines of code concurrently, starting from the cursor position. Their model receives the 10 preceding lines as input when suggesting completions, and during training, each line of code is paired with the 10 preceding lines.

Additionally, they incorporate token-level information, including keywords, identifiers, and operators, to enhance predictions.

While LLM-based code completion, including the studies mentioned above, is effective for generating code snippets, none of these approaches strictly ensure grammatical correctness. Our method explicitly leverages the syntactic structure of the code being edited, guiding the LLM to generate grammatically valid completion candidates by filling in plausible lexemes for tokens in structural candidates consisting of terminal and non-terminal symbols.

7.5. Ranking Candidates

Completion candidates are typically presented in a popup window, making their order crucial for efficient program development. Beyond IntelliCode Compose, mentioned in Section 7.2, several studies have investigated candidate ranking, primarily for identifier completion. These studies often count occurrences of identifiers or sequences of terminal and non-terminal symbols in source code repositories, such as GitHub, or within the code currently being developed.

Decades ago, Kersten et al. [15] explored ranking elements in IDEs, not necessarily limited to completion candidates. They introduced Mylar, an Eclipse extension that integrates task context into the development environment. Mylar filters and ranks elements, including completion candidates, based on programmer activity. Following this idea, Robbes et al.

[29] focused on ranking identifier completion candidates using program editing history. Their system stored every editing action performed by the programmer. They highlighted a key issue: in Eclipse, Java identifier completions often produce an overwhelming number of candidates, listed alphabetically. As a result, developers frequently have to scroll extensively, making selection slower than typing the full identifier.

Hu et al. [10] introduced an alternative ranking approach for identifier completion. Instead of relying solely on prefix matching, they proposed subsequence matching, allowing users to find names even when characters appear non- consecutively. For example, typing "swut" could yield suggestions such as "SwingUtilities" and "ShowFullPath", where the former is likely more relevant. Their study leveraged acronym usage patterns to optimize ranking and employed a support vector machine (SVM) trained on a large codebase to rank identifiers. While their method uses SVM-based ranking, ours ranks candidates by counting symbol frequencies per parse state and subsequently employs LLMs to expand symbols into code fragments.

Jiang et al. [12] investigated method invocation and field access completion, introducing what they termed project- specific candidates. Their approach combined heuristics and neural networks, revealing that over 50% of member accesses referred to classes declared within the same project. They examined cases where a user is writing an incomplete Java assignment statement of the form "$c\ n = e$." , where c is a class name, n is an identifier, and e is an expression of a type declared in the project. Their method ranked member name completions based on frequency within the project's source code.

Raychev et al. [27] focused on method call synthesis, filling holes in programs by treating completion as a natural language processing (NLP) problem. They estimated the likelihood of token sequences forming valid method call expressions, leveraging an extensive dataset from GitHub and other sources. Their algorithm used N-gram analysis, ranking candidates based on the frequency of N-token sequences to enhance completion accuracy.

7.6. Guiding LLMs

With the advancement of LLMs, several studies have explored techniques for guiding their outputs. In general, an LLM's response is highly influenced by its input, commonly referred to as a prompt. A recent study [33] extensively surveyed techniques for generating prompts to control LLM behavior. While LLM guidance has been applied across various domains, to the best of our knowledge, no studies have specifically focused on program text completion.

Although not explicitly aimed at guiding LLMs, a recent study [38] proposed a method for generating code snippets. Given a natural language description, their approach first retrieves relevant API or library function documentation, then incorporates this documentation into prompts for LLMs to generate the desired code. This method is particularly effective when dealing with recently developed APIs or libraries that the LLM has not been extensively trained on. Similarly, our proposed approach is beneficial when LLMs lack precise training on language grammars, ensuring that code completion aligns with syntactic correctness.

7.7. Integration of Grammar Provision in LLM-based Code Completion

Sasano and Choi [30,31] introduced a syntax completion method using LR parsing, identifying structural candidates for completion. Another study [2] has combined LLMs with parsing techniques to refine completion accuracy. Recent research has investigated the integration of grammar rules into neural models for code completion.

For instance, Melcer et al. [39] developed an incremental parser that extends the Earley parsing algorithm to handle left and right quotients of context-free grammars. This approach allows for early rejection of syntactically incorrect code and efficient detection of complete programs in fill-in-the-middle tasks, significantly reducing syntax errors in generated code. Similarly, Geng et al. [40] introduced grammar-constrained decoding (GCD) as a unified framework for structured NLP tasks. Their method ensures that outputs adhere to specified grammatical structures without requiring task-specific fine-tuning, demonstrating substantial improvements in tasks like information extraction and entity disambiguation.

Ugare et al. [41] presented SynCode, a framework that leverages context-free grammars to guide LLMs in generating syntactically valid outputs. By constructing a deterministic finite automaton-based mask during decoding, SynCode effectively reduces syntax errors in generated code across languages such as JSON, Python, and Go. Additionally, Kim et al. [42] proposed Compound Probabilistic Context-Free Grammars for grammar induction. Their approach models sentence with rule probabilities modulated by continuous latent variables, capturing dependencies beyond traditional context-free assumptions and improving unsupervised parsing in English and Chinese datasets.

These works collectively highlight the significance of incorporating grammatical structures into code generation models. However, they do not explicitly assess whether providing grammar rules enhances LLM-based code completion

7.8. Integration of Different LLMs into Code Completion

With The integration of LLMs into code completion tools has garnered significant attention in recent research. OpenAI's ChatGPT has been extensively utilized for code generation and completion tasks. Studies have demonstrated that prompt engineering can substantially enhance ChatGPT's code generation performance, highlighting the model's adaptability to various coding scenarios [43]. Additionally in their study, empirical evaluations have assessed ChatGPT's effectiveness in code generation, program repair, and code summarization, providing insights into its practical applications and limitations in software engineering.

In parallel, Meta AI introduced Code Llama, a code-specialized version of the Llama 3 model, designed to generate and discuss code. Trained on a diverse dataset, it supports multiple programming languages, including Python, C++, Java, and more, aiming to enhance developer workflows by providing efficient code generation and completion capabilities [44]. Comparative analyses have shown that it outperforms other models in specific tasks, such as OpenAPI code completion, indicating its potential superiority in certain coding applications [45].

Despite these advancements, direct comparative studies between ChatGPT and Llama-based models in the context of LR parsing-based code completion remain limited. This study aims to address this gap by empirically evaluating the performance of ChatGPT 3.5 and Llama 3 within this specific LR-parsed framework, providing insights into their relative effectiveness in generating accurate and syntactically correct code completions.

8. Conclusion and Future Work

In this study, we introduced a hybrid method for automatically composing prompts to the LLM using structural candidates provided by the LR-based technique and evaluated the method using two programming languages. Compared to the previous work [4], the improved system can now suggest textual candidates alongside structural candidates, rather than only structural structures. By incorporating structural candidates into the prompts, the system can effectively guide the LLM and control its predictions by excluding lower-ranked structural candidates for code completion.

Though there have been some notable improvements in our system, there are still many areas for future work. It is anticipated that expanding the dataset could further enhance precision. We can conduct a usability analysis to determine whether it genuinely benefits real programmers by building a more extensive database of ranked LR structural candidates collected from a wider variety of programming languages and source programs. We aim to develop a generalized VSCode extension that supports a broader range of popular programming languages, enhancing code completion and candidate guidance across diverse development environments. Teaching grammar to ChatGPT has proven ineffective in our current experiment, indicating the need for further exploration of more effective instructional methods. Future work could focus on providing grammar information to ChatGPT, emphasizing strategies that enhance its understanding and application of grammatical rules.

Acknowledgments

We would like to special thanks to Seokwoo Yang, Shin Hyo-won, and Hyojeong Kang for contributing to build the VSCode extension for Microsoft Small Basic language. This work was supported by Innovative Human Resource Development for Local Intellectualization program through the

Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (IITP-2023-RS-2023-00256629). This work was partially supported by the Korea Internet & Security Agency (KISA) - Information Security College Support Project. Also, this work was partially supported by JSPS KAKENHI under Grant Number 23K11053.

Declaration of generative AI and AI-assisted technologies in the writing process

During the preparation of this work the authors used ChatGPT in order to refine the English writing style. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication.

References

[1] Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D., 2006. Compilers — principles, techniques, and tools, 2nd edition. Addison Wesley.

[2] Atique, M.M.A.B., Choi, K., Sasano, I., Moon, H.A., 2024. Improving LLM-based code completion using LR parsing-based candidates, in: 10th International Symposium on Symbolic Computation in Software Science — Work in Progress Workshop, pp. 1–6.

[3] Choi, K., 2023. Implementations and Evaluation Results with SmallBasic and C11. https://drive.google.com/drive/folders/1UqSN3qnhn9eSW6Igx-AUsRAfQN_ZpAMH?usp=sharing.

[4] Choi, K., Hwang, S., Moon, H., Sasano, I., 2024. Ranked syntax completion with LR parsing, in: Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing, Association for Computing Machinery, New York, NY, USA. p. 1242–1251. URL: https://doi.org/10.1145/3605098.3635944, doi:10.1145/3605098.3635944.

[5] Choi, K., Kim, G., Chang, B.M., 2018. A development of open-source software for educational coding environments using small basic. KIISE Transactions on Computing Practices 24, 649–661. doi:https://doi.org/10.5626/KTCP.2018.24.12.649.

[6] Foundation, P.S., 2023. Difflib — Helpers for computing deltas. Available at https://docs.python.org/3/library/difflib.html.

[7] Gabel, M., Su, Z., 2010. A study of the uniqueness of source code, in: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA. p. 147–156. URL: https://doi.org/10.1145/1882291.1882315, doi:10.1145/1882291.1882315.

[8] Goto, T., Sasano, I., 2012. An approach to completing variable names for implicitly typed functional languages, in: Proceedings of the ACM

SIGPLAN 2012 workshop on Partial Evaluation and Program Manipulation, ACM Press, Philadelphia, Pennsylvania, USA. pp. 131–140. URL: http://doi.acm.org/10.1145/2103746.2103771, doi:10.1145/2103746.2103771.

[9] Hindle, A., Barr, E.T., Su, Z., Gabel, M., Devanbu, P., 2012. On the naturalness of software, in: 2012 34th International Conference on Software Engineering (ICSE), pp. 837–847. doi:10.1109/ICSE.2012.6227135.

[10] Hu, S., Xiao, C., Ishikawa, Y., 2019. Scope-aware code completion with discriminative modeling. Journal of Information Processing 27, 469–478. doi:10.2197/ipsjjip.27.469.

[11] IntelliSense, . Intellisense. https://code.visualstudio.com/docs/editor/intellisense.

[12] Jiang, L., Liu, H., Jiang, H., Zhang, L., Mei, H., 2022. Heuristic and neural network based prediction of project-specific api member access. IEEE Transactions on Software Engineering 48, 1249–1267. doi:10.1109/TSE.2020.3017794.

[13] Jourdan, J.H., Pottier, F., 2017. A simple, possibly correct lr parser for c11. ACM Transactions on Programming Languages and Systems (TOPLAS) 39, 1–36.

[14] JTC1/SC22/WG14, 2018. The C11 programming language. http://www.open-std.org/jtc1/sc22/wg14/.

[15] Kersten, M., Murphy, G.C., 2006. Using task context to improve programmer productivity, in: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA. p. 1–11. URL: https://doi.org/10.1145/1181775.1181777, doi:10.1145/1181775.1181777.

[16] Lim, J., Kim, G., Shin, S., Choi, K., Kim, I., 2019. A method for efficient development of parsers via modular lr automaton generation (in korean), in: Proceedings of Korea Computer Congress (KCC 2019), Korea. pp. 1578–1580.

[17] Lim, J., Kim, G., Shin, S., Choi, K., Kim, I., 2020a. Parser generators sharing LR automaton generators and accepting general purpose programming language-based specifications. Journal of KIISE 47, 52–60. doi:10.5626/JOK.2020.47.1.52. (in Korean).

[18] Lim, J., Kim, G., Shin, S., Choi, K., Kim, I., 2020b. Parser generators sharing lr automaton generators and accepting general purpose programming languagebased specifications. Journal of KIISE 47, 52–60.

[19] Liu, F., Fu, Z., Li, G., Jin, Z., Liu, H., Hao, Y., Zhang, L., 2024. Non-autoregressive line-level code completion. ACM Trans. Softw. Eng. Methodol. URL: https://doi.org/10.1145/3649594, doi:10.1145/3649594. just Accepted.

[20] Marasoiu, M., Church, L., Blackwell, A., 2015. An empirical investigation of code completion usage by professional software developers, in: Proceedings of the 26th Annual Workshop of the Psychology of Programming Interest Group, p. 12.

[21] Microsoft, . Language server protocol. https://microsoft.github.io/language-server-protocol/.

[22] Microsoft, 2023. Microsoft Small Basic. http://smallbasic.com.

[23] Nguyen, S., Nguyen, T.N., Li, Y., Wang, S., 2020. Combining program analysis and statistical language model for code statement completion, in: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, IEEE Press. p. 710–721. URL: https://doi.org/10.1109/ASE.2019.00072, doi:10.1109/ASE.2019.00072.

[24] Parr, T., Fisher, K., 2011. LL(*): The foundation of the ANTLR parser generator, in: Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 425–436. doi:10.1145/1993498.1993548.

[25] Post, M., 2018. A call for clarity in reporting bleu scores. arXiv preprint arXiv:1804.08771 .

[26] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., 2018. Language models are unsupervised multitask learners. https:

//paperswithcode.com/paper/language-models-are-unsupervised-multitask.

[27] Raychev, V., Vechev, M., Yahav, E., 2014. Code completion with statistical language models, in: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, Association for Computing Machinery, New York, NY, USA. p. 419–428. URL: https://doi.org/10.1145/2594291.2594321, doi:10.1145/2594291.2594321.

[28] Rekers, J., Koorn, W., 1991. Substring parsing for arbitrary context-free grammars. SIGPLAN Not. 26, 59–66. URL: https://doi.org/10.1145/122501.122505, doi:10.1145/122501.122505.

[29] Robbes, R., Lanza, M., 2008. How program history can improve code completion, in: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, pp. 317–326. doi:10.1109/ASE.2008.42.

[30] Sasano, I., Choi, K., 2021. A text-based syntax completion method using LR parsing, in: Proceedings of the 2021 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, Association for Computing Machinery, New York, NY, USA. p. 32–43. URL: https://doi.org/10.1145/3441296.3441395, doi:10.1145/3441296.3441395.

[31] Sasano, I., Choi, K., 2023. A text-based syntax completion method using LR parsing and its evaluation. Science of Computer Programming, 102957URL: https://www.sciencedirect.com/science/article/pii/S0167642323000394, doi:https://doi.org/10.1016/ j.scico.2023.102957.

[32] Sasano, I., Goto, T., 2013. An approach to completing variable names for implicitly typed functional languages. Higher-Order and Symbolic Computation 25, 127–163. doi:10.1007/s10990-013-9095-x.

[33] Schulhoff, S., Ilie, M., Balepur, N., Kahadze, K., Liu, A., Si, C., Li, Y., Gupta, A., Han, H., Schulhoff, S., Dulepet, P.S., Vidyadhara, S., Ki, D., Agrawal, S., Pham, C., Kroiz, G., Li, F., Tao, H., Srivastava, A., Costa, H.D., Gupta, S., Rogers, M.L., Goncearenco, I., Sarli, G., Galynker, I., Peskoff, D., Carpuat, M., White, J., Anadkat, S., Hoyle, A., Resnik, P., 2024. The prompt report: A systematic survey of prompting techniques. URL: https://arxiv.org/abs/2406.06608, arXiv:2406.06608.

[34] Svyatkovskiy, A., Deng, S.K., Fu, S., Sundaresan, N., 2020. Intellicode compose: Code generation using transformer, in: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY,

USA. p. 1433–1443. URL: https://doi.org/10.1145/3368089.3417058, doi:10.1145/3368089.3417058.

[35] Tomassetti, F., 2016. Building autocompletion for an editor based on antlr. https://tomassetti.me/autocompletion-editor-antlr/.

[36] Tomita, M., 1985. Efficient parsing for natural language: A fast algorithm for practical systems. Kluwer Academic Publishers. doi:10.1007/ 978-1-4757-1885-0.

[37] Wang, Y., Le, H., Gotmare, A.D., Bui, N.D.Q., Li, J., Hoi, S.C.H., 2023. Codet5+: Open code large language models for code understanding and generation. URL: https://arxiv.org/abs/2305.07922, arXiv:2305.07922.

[38] Zhou, S., Alon, U., Xu, F.F., Wang, Z., Jiang, Z., Neubig, G., 2023. Docprompting: Generating code by retrieving the docs. URL: https://arxiv.org/abs/2207.05987, arXiv:2207.05987.

[39] Melcer, Daniel, Nathan Fulton, Sanjay Krishna Gouda, and Haifeng Qian. "Constrained Decoding for Code Language Models via Efficient Left and Right Quotienting of Context-Sensitive Grammars." arXiv preprint arXiv:2402.17988 (2024).

[40] Geng, Saibo, Martin Josifoski, Maxime Peyrard, and Robert West. "Grammar-constrained decoding for structured NLP tasks without finetuning." arXiv preprint arXiv:2305.13971 (2023).

[41] Ugare, Shubham, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. "Syncode: Llm generation with grammar augmentation, 2024." URL https://arxiv. org/abs/2403.01632.

[42] Kim, Yoon, Chris Dyer, and Alexander M. Rush. "Compound probabilistic context-free grammars for grammar induction." arXiv preprint arXiv:1906.10225 (2019).

[43] Liu, Jiawei, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation." Advances in Neural Information Processing Systems 36: 21558-21572, 2023.

[44] Meta AI. "Introducing Code Llama, a State-of-the-Art Large Language Model for Coding." Meta AI Blog, August 24, 2023. https://ai.facebook.com/blog/code-llama-large-language-model-coding/.

[45] Caumartin, Genevieve, Qiaolin Qin, Sharon Chatragadda, Janmitsinh Panjrolia, Heng Li, and Diego Elias Costa. "Exploring the Potential of Llama Models in Automated Code Refinement: A Replication Study." arXiv preprint arXiv:2412.02789, 2024.

[46] Sippu, S., Soisalon-Soininen, E., 1990. Parsing theory volume 2: LR(K) and LL(K) parsing. Springer-Verlag, Berlin, Heidelberg.

[47] Anton Semenkin et al. Full Line Code Completion: Bringing AI to Desktop. International Conference on Software Engineering (ICSE 2025). JetBrains, 2025.

[48] Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. Productivity Assessment of Neural Code Completion. In Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (MAPS 2022), pages 21-29. https://doi.org/10.1145/3520312.3534864

LR 파싱을 활용한 LLM 기반 코드 자동 완성 개선에 관한 연구

ATIQUE MD MONIR AHAMMOD BIN

전남대학교 대학원 인공지능융합학과

(지도교수 : 최광훈 교수)

(국문초록)

현대 통합 개발 환경(IDE)에서 코드 자동 완성은 프로그래밍 효율성을 높이는 핵심 기능 중 하나이다. 기존의 자동 완성 시스템은 접두어 필터링과 정적 순위 지정을 기반으로 하지만, 사용자가 긴 목록을 일일이 탐색해야 하며, 이는 대부분 알파벳 순으로 정렬되어 있어 사용성을 저하시킨다. 최근에는 언어 구문 정보를 활용하는 LR 파싱 기반 자동 완성 기법이 제안되었으며, 오픈소스 프로그램을 활용해 후보 코드의 순위를 계산하는 방법이 연구되고 있다. 하지만 이러한 기법은 구조적 후보만을 제안하고 완전한 코드 생성을 위해서는 사용자의 수동 보완이 필요하다는 한계가 있다. 이러한 문제를 해결하기 위해, 본 연구에서는 LR 파싱과 대형 언어 모델(LLM)을 결합한 하이브리드 코드 완성 기법을 제안한다. 제안된 방법은 LR 파서를 통해 생성된 구조적 후보를 LLM 이 실제 코드로 변환하여 사용자에게 제시하며, 이때 오픈소스 프로그램에서 수집된 순위 기반 후보 데이터베이스를 참조한다. 이로써 LR 파싱의 구문 정확도와 LLM 의 생성 능력을 통합하여 코드 자동 완성의 정확성과 실용성을 동시에 향상시키고자 한다. 본 연구는 LLM 이 LR 기반 구조 후보로부터 실질적인 이점을 얻을 수 있는지를 실험적으로 분석한다. 이를 위해, 구조 후보를 포함한 경우와 포함하지 않은 경우를 비교하여 자동 완성 정확도의 차이를 평가하였다. 또한, 기존 연구를 바탕으로, 높은 순위를 가진 구조 후보를 활용하는 방식이 LLM 기반 코드 완성의 정밀도를 향상시키는 데 얼마나 효과적인지를 추가로 분석하였다. 아울러, 본 연구에서는 제안 기법을 Microsoft Small Basic 과 C

언어용 VSCode 확장 프로그램을 통해 구현하여 실제 적용 가능성을 검증하였다. 이 시스템은 언어 중립적 설계를 기반으로 하며, LR 문법이 정의된 모든 프로그래밍 언어에 적용 가능하다. 실험 결과, LR 파싱 기반 후보를 LLM 기반 완성과 통합하면 정확성과 사용성이 모두 향상됨을 확인할 수 있었다. 또한, 본 연구는 LR 문법 기반 구조 후보를 LLM 기반 코드 완성에 통합함으로써 성능이 향상되는지를 실험적으로 분석하였다. Microsoft Small Basic 과 C 언어를 대상으로 SacreBLEU 및 SequenceMatcher 평가 지표를 사용한 결과, 정확도의 유의미한 향상은 나타나지 않았으며, 이는 LLM 이 이미 내부적으로 문법을 학습했거나 기존 평가 지표로는 성능 차이를 충분히 반영하지 못할 수 있음을 시사한다. 그럼에도 불구하고, LLM 의 발전에도 불구하고, 구문 인식 기반 코드 생성에서 LLM 선택이 성능에 미치는 영향에 대한 논의는 아직 부족하다. 이에 본 연구는 LR 파싱 기반 코드 자동 완성 프레임워크 내에서 대표적인 두 LLM 인 ChatGPT 3.5 와 Llama 3 의 성능을 비교 분석하였다. 실험 결과, ChatGPT 3.5 가 Llama 3 보다 더 높은 정확도를 기록하였으며, 이는 코드 완성 성능 향상을 위해 적절한 LLM 선택이 중요함을 시사한다. 이러한 결과는 LR 파싱 기반 LLM 코드 완성에서 모델 선택이 성능에 미치는 영향을 강조하며, 향후 연구에서는 보다 다양한 LLM 을 포함한 비교 분석이 필요함을 제안한다.

Appendix A. List of publications based on this thesis

The following publications are based on or related to the research presented in this thesis:

1. Md Monir Ahammod Bin Atique, Hyeon-Ah Moon, Isao Sasano, and Kwanghoon Choi. "Improving LLM-based Code Completion Using LR Parsing “Journal of Computer Languages, Elsevier, 2025. [Minor Revisions]

2. Md Monir Ahammod Bin Atique, Kwanghoon Choi, Isao Sasano, and Hyeon-Ah Moon. "Improving LLM-based Code Completion Using LR Parsing-Based Candidates." In CEUR Workshop Proceedings, vol. 3754,

SCSS Symposium, 2024. Available at: https://ceur-ws.org/Vol-3754/paper01.pdf

3. Md Monir Ahammod Bin Atique 와 최광훈. "LR 파싱을 활용한 LLM 기반 코드 완성에서 문법 제공 비교 분석". 2025 한국스마트미디어학회&한국전자거래학회 춘계학술대회, 중앙대학교.

4. Md Monir Ahammod Bin Atique 와 최광훈. "LR 파싱을 이용한 이용한 LLM 기반 코드 완성에서 ChatGPT 3.5 와 Llama 3 의 비교 분석". 2025 년 한국정보처리학회 연례 심포지엄 (ASK 2025), 경북대학교.

Appendix B. Guide to setting Up and running the VS code extension

This appendix provides instructions on how to set up and use the **Microsoft Small Basic** and **C** Programming language extensions. Both systems are built with similar architecture and involve running a parser and a Visual Studio Code extension for real-time code suggestions.

Section 1: Prerequisites and Installation

Before proceeding, make sure your system has the required tools installed:

1. **Stack**, **GHC**, and **GHCi**:

   - Install using GHCup: https://www.haskell.org/ghcup/

   - Refer to slides 65–75 from Professor Kwanghoon Choi's lecture: https://docs.google.com/presentation/d/1fhXvoLHFgYE4AOfdl4MD_Puk7Vbj-8eKqquTi7b0t9I/edit#slide=id.g2bed8681a8e_0_205

2. **Visual Studio Code**:

- Download from https://code.visualstudio.com
- Install TypeScript extensions if prompted.

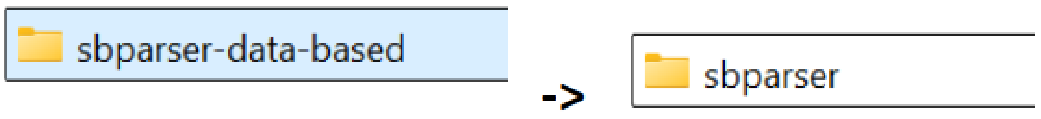
Section 2: Download Required Repositories

You need to download or clone three folders for each language. Place all three in the same directory.

2.1 For Small Basic

1. sbparser: https://github.com/kwanghoon/sbparser/tree/data-based

Unzip and rename the folder:

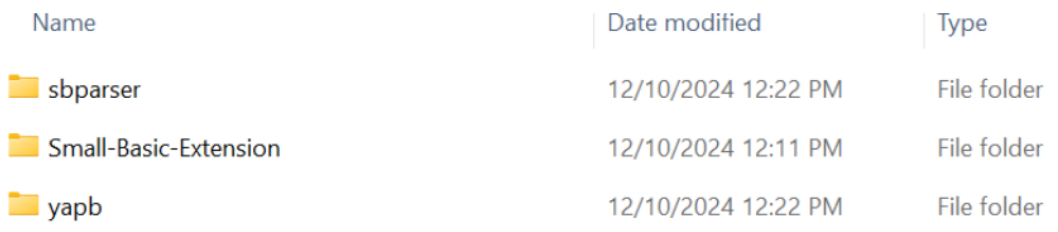


2. yapb: git clone https://github.com/kwanghoon/yapb
3. Small Basic Extension: git clone https://github.com/monircse061/Small-Basic-Extension

2.2 For C Programming

1. c11parser: https://github.com/kwanghoon/c11parser/tree/data-based
2. yapb: git clone https://github.com/kwanghoon/yapb
3. C Extension: git clone https://github.com/monircse061/C-Extension

Put these three folders in the same directory.



Name	Date modified	Type
sbparser	12/10/2024 12:22 PM	File folder
Small-Basic-Extension	12/10/2024 12:11 PM	File folder
yapb	12/10/2024 12:22 PM	File folder

The implementation, built on Microsoft Small Basic and C11 using LR grammars, uses the YAPB parser builder to simplify syntax completion tool development. YAPB enables developers to build language-specific tools by simply providing a compatible lexer and parser specification.

Key points:

- Target languages: Microsoft Small Basic & C11
- Grammar type: LR grammar
- Tool used: YAPB
- Why YAPB? It streamlines the development process—only a lexer and parser spec are needed.

Section 3: Running the Parser

Open the terminal and navigate into the parser folder and run the following two commands.

For Small Basic

```
cd sbparser
stack build
stack run sbparser-exe.exe emacs
```

For C Programming

```
cd c11parser
stack build
stack run c11parser-exe.exe emacs
```

Section 4: Launching VS Code Extension

1. Open the appropriate extension folder (Small-Basic-Extension or C-Extension) in Visual Studio Code.

2. Insert your OpenAI API key into these files:

   - src/extension.ts
   - src/sbSnippetGenerator.ts or src/cSnippetGenerator.ts
   - out/extension.js
   - out/sbSnippetGenerator.js or out/cSnippetGenerator.js

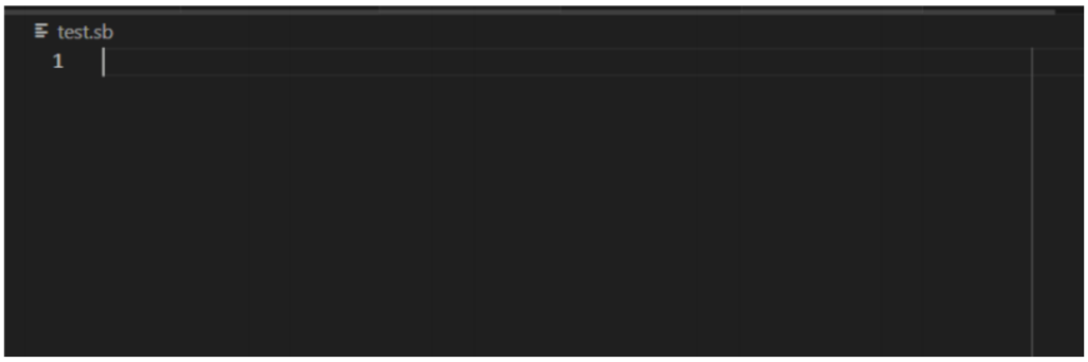
3. Open src/extension.ts and press F5, or go to (click) Run → Start

Debugging. Ignore any messages that appear. A new vscode window will appear

Section 5: Writing and Testing Programs

Small Basic Example

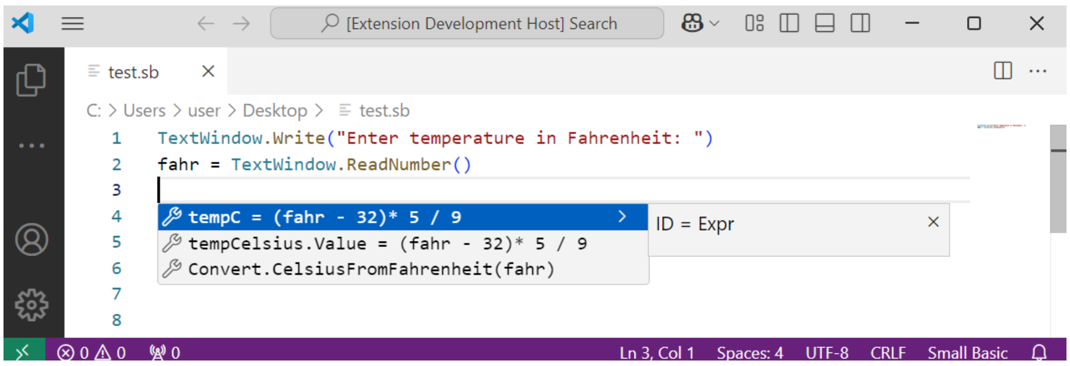
1. Create a file named test.sb in the new VS Code window.



2. Copy code from src/test_programs.sb or write code into the newly opened editor .

3. Press Ctrl + C to get suggestions. Wait a few seconds for the response to appear.

4. Use keyboard arrows to select and press Enter. Instead of selecting suggestions by mouse click, select by keyboard keypad down arrow button. In the debug console, you can view the output to see more information during the running process.


[Extension Development Host] Search
test.sb
C: > Users > user > Desktop > test.sb

```
TextWindow.Write("Enter temperature in Fahrenheit: ")
fahr = TextWindow.ReadNumber()

tempC = (fahr - 32)* 5 / 9
tempCelsius.Value = (fahr - 32)* 5 / 9
Convert.CelsiusFromFahrenheit(fahr)
```

ID = Expr

Ln 3, Col 1 Spaces: 4 UTF-8 CRLF Small Basic

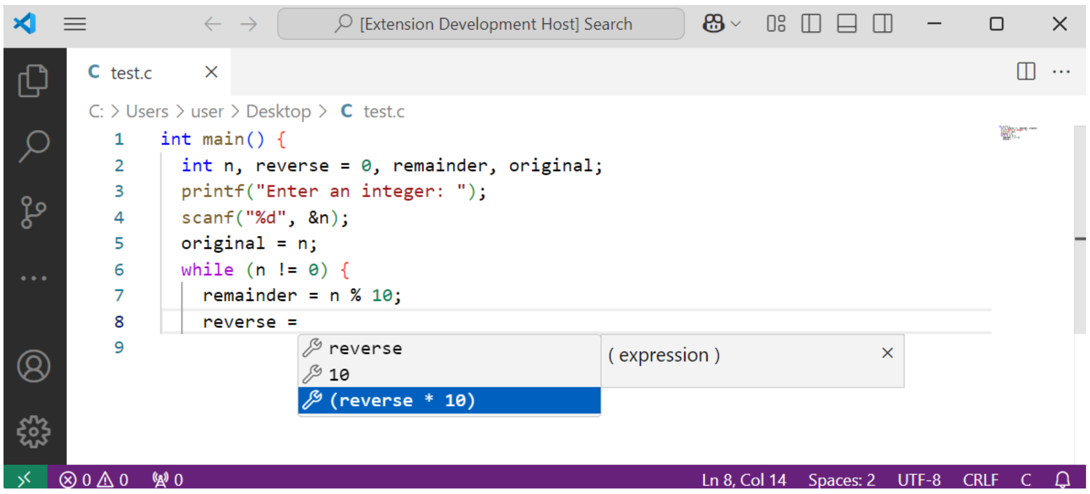
C Programming Example

1. Create a file named test.c.

2. Write or copy a C program.

3. Trigger suggestions using Ctrl + C.

4. Select using arrow keys and press Enter.


[Extension Development Host] Search
test.c
C: > Users > user > Desktop > test.c

```
int main() {
  int n, reverse = 0, remainder, original;
  printf("Enter an integer: ");
  scanf("%d", &n);
  original = n;
  while (n != 0) {
    remainder = n % 10;
    reverse =
```

```
reverse
10
(reverse * 10)
```

(expression)

Ln 8, Col 14 Spaces: 2 UTF-8 CRLF C

Section 6: Troubleshooting

If suggestions do not appear:

- Check the terminal to ensure the parser (sbparser or c11parser) is running.

- If disconnected, rerun the following:

# For Small Basic

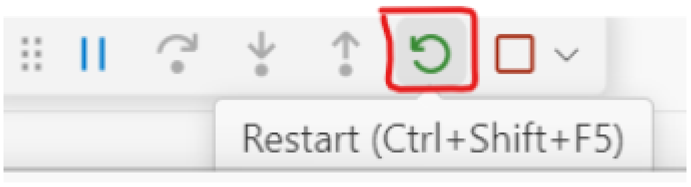
```
stack run sbparser-exe.exe emacs
```

# For C Programming

```
stack run c11parser-exe.exe emacs
```

Then restart debugging in VS Code or click Run -> Start Debugging again to restart the debugging process.

Or click here


Restart (Ctrl+Shift+F5)