Improving LLM-based Code Completion Using LR Parsing-Based Candidates

Md Monir Ahammod Bin Atique[1,†], Kwanghoon Choi[1,*,†], Isao Sasano[2] and Hyeon-Ah Moon[3]

[1]*Chonnam National University, Gwangju 61186, South Korea*
[2]*Shibaura Institute of Technology, Tokyo, Japan*
[3]*Sogang University, Seoul, South Korea*

Abstract

Programmers often use syntax completion and code suggestion features. Our methodology enhances code completion by combining structural candidate information from LR parsing with LLMs. These structural candidates are utilized to compose prompts so that ChatGPT can predict actual code under the specified structure. Tested on Small Basic and C benchmarks, this approach offers textual suggestions rather than just structural ones, showing nearly 50% prediction accuracy for Small Basic programs. While effective for Small Basic, we report that challenges remain with C11 programs.

Keywords

Syntax Completion, Large Language Model, LR parsing, Integrated Development Environments

1. Introduction

Many integrated development environments (IDEs), such as Visual Studio Code, provide syntax completion features that ease the editing process for various programming languages. Developers of IDEs should prioritize incorporating syntax completion for each supported language. To make the process more efficient and cost-effective, it is beneficial to approach this implementation methodically, guided by a detailed specification.

An analytic approach is based on syntax analysis using the well-developed LR parsing theory [1]. Sasano & Choi [2] defined code completion candidates γ for a prefix $\alpha\beta$ as suffix sentential forms derived from a start symbol S if there is a production $A \to \beta\gamma$ in the LR grammar so that $\beta\gamma$ can be reduced to a nonterminal A. Figure 1 describes this idea.

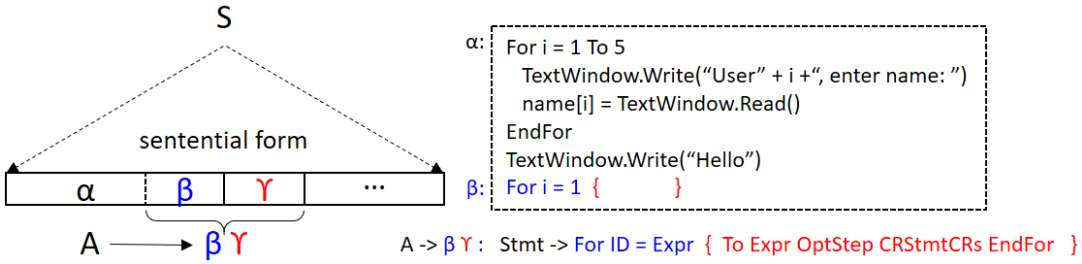

Figure 1: The idea of structural candidates for code completion using LR parsing [2]

For example, on a request for a code completion on (the part of) a prefix 'For i = 1', β is 'For ID = Expr', which is a sequence of terminal and noterminal symbols describing the beginning of the for loop.

SCSS 2024: 10th International Symposium on Symbolic Computation in Software Science, August 28–30, 2024, Tokyo, Japan
*Corresponding author.
†These authors contributed equally.
monir024@jnu.ac.kr (M. M. A. B. Atique); kwanghoon.choi@jnu.ac.kr (K. Choi); sasano@sic.shibaura-it.ac.jp (I. Sasano); hamoon@sogang.ac.kr (H. Moon)
https://monircse061.github.io/page/ (M. M. A. B. Atique); https:https://kwanghoon.github.io/ (K. Choi)
0009-0000-7103-9744 (M. M. A. B. Atique); 0000-0003-3519-3650 (K. Choi); 0000-0002-9373-6206 (I. Sasano); 0000-0001-7359-3298 (H. Moon)

Sasano & Choi's method [2] [3] can automatically uncover a candidate γ, which is 'To Expr OptStep CRStmtCRs EndFor' to complete the rest of the for loop by a production 'Stmt $\rightarrow$ For ID = Expr To Expr OptStep CRStmtCRs EndFor'. Consequently, IDEs will respond with this candidate γ to the request for a code completion on 'For i = 1'. Their continuing research [4] proposed a ranking method useful to choose more likely candidate than the others when there are more than one candidate possible for a prefix. It pre-investigates the frequencies of the occurrences of the candidates in the existing open-source projects.

These methods [2][3] [4] are advantageous. The suggested candidates are guaranteed to be correct syntactically, ranking can be customized for an individual software project, and this method can be implemented in a programming language agnostic way.

However, the suggested candidates by the methods are limited to the form of terminal and noterminal symbols. After choosing a candidate, programmers should manually edit it into a code text, which diminishes productivity. Determining such a code text for a candidate is beyond the LR parsing-based syntax analysis.

In this work, we study how Large Language Model [5] can complement these methods. Given a prefix text for $\alpha\beta$, the LR parsing based method firstly suggests a candidate γ, and the LLM produces a code completion satisfying 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 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 complement each other. The LR parsing based analytic approach can precisely specify the syntactic code structure to complete, while the LLM-based statistical approach can predict the code text under the specified structure. According to [4], the top 1.8 suggested candidates in the SmallBasic programs and the top 3.15 suggested candidates in the C11 programs on average were found to be what are expected for testing. This evaluation results imply that candidates in the form of the rest structural candidates should not be considered by the LLM. Composing prompts using the top suggested structual candidates will be effective to instruct the LLM to exclude the bottom ones for code completion.

To the best of our knowledge, this is the first attempt to guide an LLM using prompts that utilize candidate structural information obtained from LR-parsing. We report ongoing work in this direction.

Our contributions are as follows.

Firstly, we propose a code completion prediction method that combines LR-parsing-based ranking of candidate skeletons with Large Language Model (LLM)-based fleshing out of those skeletons.

Secondly, we have setup an environment to evaluate the proposed method and report initial results using SmallBasic and C11 benchmarks.

Section 2 introduces our system and presents initial evaluation results . Section 3 compares our work with existing research. Section 4 concludes the paper with future work.

2. An Overview of Our System and Its Evaluation

Figure 2 shows an overview of our system. The system operates in two phases. The collection phase constructs a database from sample programs, mapping parse states to sets of ranked candidates. The query phase retrieves a sorted list of candidates based on their ranks for a parse state corresponding to a given cursor position being edited. The top suggested structural candidates are chosen from the sorted list to compose a prompt, and then the LLM fleshes out the structural candidates to produce textual candidates, which will be displayed to the programmer for code completion.

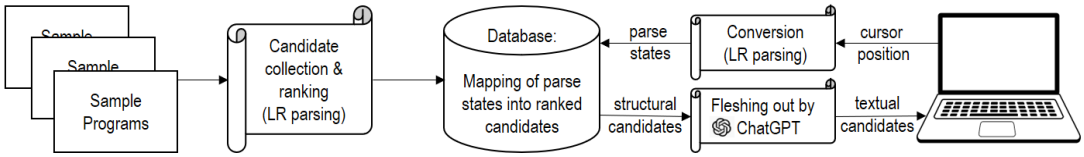

Figure 2: Overview of our system

In this work, we focus on one aspect of this system: automatically composing prompts to the LLM using structural candidates offered by the LR-based method is feasible and is an advancement to the previous work [4] in that this system can now suggest textual candidates rather than structural candiates.

Under this goal, we design an experiment to evaluate the effectiveness of using ChatGPT for code completion suggestions and to offer structural candidates (composed of terminals and nonterminals) to guide these suggestions. Our proposed system can be assessed by addressing the following two research questions (RQ):

- RQ1: Does the proposed system offer textual (actual) candidate suggestions with the aid of LLM such as ChatGPT that are beneficial in introductory programming?
- RQ2: Is it reasonable to implement the system as a language-parametric tool?

In order to address the above research questions, our methodology includes the following steps. **Selection of Programming Languages:** We selected two programming languages, Microsoft SmallBasic (MSB) and C11, for our experiments as in the previous work [4]. These languages are popular choices for introductory programming. **Data Collection:** The testing set for SmallBasic was obtained from its community. It consists of 27 programs totaling 155 lines taken from the well-known MSB tutorial. Talking about C, the test set of C11 comprises 106 programs (11,218 lines in total) that are solutions from the well-known book on the C programming language by Kernighan and Ritchie. **Prefix Extraction:** Prefixes were collected from the source code files, along with cursor position information. This information was obtained from candidates' database by the lexical analysis in the LR parsing-based method [4]. **Prompt Engineering:** Using the collected prefix and structural candidate data, we crafted prompts for ChatGPT. In this experiment, we selected the 'gpt-3.5-turbo-0125' model for its better performance with code completion. These information were then fed into the ChatGPT prompt to ask for substitutes for our structural candidates into actual candidates. We compared the answers provided by ChatGPT with the correct answers from our database. **Evaluation:** The responses from ChatGPT were evaluated using well-known techniques such as SacreBLEU, which is a popular method for assessing large language models and SequenceMatcher similarity. The SacreBLEU score measures the n-gram (sequences of n items, typically words or characters) 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 the reference code sequence. The SequenceMatcher is a class available in python module named "difflib". It compares 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. Here, we used the parameter isjunk=None so that no elements are ignored. The data set as well as the developed software are all available in the public repository [1].

[1] https://github.com/monircse061/ChatGPT-Code-Completion-Work

We present a summary of the experimental results for both MSB and C11 which is dipicted in Table 1. For MSB, we experimented with 27 programs where, for each program, we iterated our system for each structural candidate, calculated the evaluating metrics values, and then averaged the precision for the whole program. This process was done for every program. Finally, we calculated the mean precision for the 27 programs in terms of SacreBLEU and sequence matcher similarity. On average, our system predicts the textual code suggestion with over 45% accuracy for each testing program when using SacreBLEU as an evaluation metric. Precision is almost similar at nearly 45% when sequence matcher similarity is taken into account. The similar process was used with C11. For 106 C11 programs, the average SacreBLEU score is 21.463%, indicating that our system forecasts the correct code completion suggestions. Sequence matcher similarity is nearly the same for C11.

Table 1
Experimental results (precision) on specific languages

PLs	Microsoft SmallBasic (%)	C11 (%)
SacreBLEU Precision	45.247	21.463
Sequence Matcher Precision	44.354	20.384

To show the effectiveness of guidance by a structural candidate, we discuss a case representing the best prediction of our system as this. In the MSB experiment case depicted in Figure 3, line 2600 marks the parse state and cursor position, followed by the next few lines (2602 to 2612), which provide the prompt for the ChatGPT. Lines spanning from 2603 to 2608 represent the prefix code. Subsequently, a candidate structure appears in line 2609: 'To Expression OptStep CRStmtCRs EndFor'. It interprets that the actual candidate should be 'To 5 \n TextWindow . Write (name [i] + ", ") \n EndFor', which is shown in line 2618. Line 2614 outlines the time taken from the query to the ChatGPT response, which is 0.6903 seconds. In this candidate structure, the response generated by ChatGPT is highly accurate. The precision at the unigram level (1-gram) is 100% which is seen at line 2621, and other metric also show satisfactory result (line 2622). 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.

```
2600:  Parse State: 85Cursor Position: 6 11
2601:
2602:  This is the incomplete Small Basic programming language code:
2603:  For i = 1 To 5
2604:       TextWindow.Write("User" + i + ", enter name: ")
2605:       name[i] = i
2606:  EndFor
2607:  TextWindow.Write("Hello ")
2608:  For i = 1
2609:                  'To Expression OptStep CRStmtCRs EndFor'
2610:  Complete the 'To Expression OptStep CRStmtCRs EndFor' part of the code
2611:  in the Small Basic programming language. Just show your answer in place
2612:  of 'To Expression OptStep CRStmtCRs EndFor'.
2613:
2614:  Time taken: 0.6902937889099121 seconds
2615:  Received response: To 5
2616:      TextWindow.Write(name[i] + ", ")
2617:  EndFor
2618:  Actual result:  To 5 \n TextWindow . Write ( name [ i ] + ", " ) \n EndFor
2619:  SACREBLEU Score: {'precisions': [100.0, 86.66666666666667, 78.57142857142857,
2620:  76.92307692307692], 'sys_len': 16, 'ref_len': 20}
2621:  First element of precision:100.0
2622:  Sequence Matcher Similarity Precision:0.8462619469026548
```

Figure 3: High Prediction Result (MSB)

Based on the evidence provided, we can answer Research Question 1 in the affirmative.
Using ChatGPT with LR parsing-based structural candidates is effective in providing code completion

suggestions for introductory programming languages, particularly for MSB. Our system shows correct suggestions with minimal prefixes (hints), which is notable. This indicates that the system can be beneficial in educational contexts where MSB is used. However, improvements are needed to increase precision, especially for more complex languages like C11. The precision for C programs in C11 is low due to short candidate structures like '[', ';' and complex, hard-to-infer structures. Additionally, predicting the next token or line of code with minimal prefix is challenging, especially in long files.

On answering the second research question, based on the successful application of our system to the two programming languages, we can claim that our code completion system is language-agnostic. This system can be incorporated into any programming language.

3. Related Work

There are various studies conducted up to now which use large code base and/or machine learning to code completion. One is by Svyatkovskiy et al. from Microsoft, who introduced a system named *IntelliCode Compose* [6]. This system leverages GPT-C, a variant of OpenAI's GPT-2 [5], trained on a vast dataset of program source code. It is designed to generate sequences of tokens that form syntactically correct language constructs, such as statements containing local variables, method names, and keywords, for languages including C#. Another study by [7] explored identifier completion with ranking candidates. They sought solutions to improve the efficiency of the completion process. Rather than relying on prefix matching, used in many completion systems, they introduced subsequence matching, where user-input sequences of characters are compared to names containing them, even if they are non-consecutive. Recently a study by [8] delved into method invocation and field access completion. Nguyen et al. [9] combined program analysis and langauge model for completing a partially-input statement or suggesting a statement that immediately follows the current statement if it is a complete one. Gabel et al. [10] first observed the regularity of software code mentioned above. There are infinitely many syntactically valid statements, but there are much smaller, or may even be finite, number of pracitally useful statements. Liu et al. [11] presented a non-autoregressive model for concurrently computating candidates, each of which is a line of code starting at the cursor position. 10 lines of code immediately before the current empty line is given to the completion system when programmers write code, and also 10 lines of code immediately before every line is given as training data together with the current line. They also use some information of tokens such as keywords, identifiers, operators, etc.

4. Conclusion and Future Work

In this research, we introduced a method for automatically composing prompts to the LLM using structural candidates offered by the LR-based method and assessed the method using two programming languages. Compared to the the previous work [4], this system can now suggest textual candidates rather than structural candiates. By using structural candidates in the prompts, the system can effectively instruct the LLM to exclude the bottom structural candidates for code completion.

There are many topics for future work. A few important topics are to build an IDE for usability evaluation, to measure the effectiveness of structural candidates in the prompts to the LLM, and to compare the prediction performance of our system with that of the others particularly based on the Large Language Models.

Acknowledgments

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.

References

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

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

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

[4] K. Choi, S. Hwang, H. Moon, I. Sasano, Ranked syntax completion with lr parsing, in: Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing, SAC '24, Association for Computing Machinery, New York, NY, USA, 2024, p. 1242–1251. URL: https://doi.org/10.1145/3605098.3635944. doi:10.1145/3605098.3635944.

[5] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, Language models are unsupervised multitask learners, https://paperswithcode.com/paper/language-models-are-unsupervised-multitask, 2018.

[6] A. Svyatkovskiy, S. K. Deng, S. Fu, N. Sundaresan, 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, ESEC/FSE 2020, Association for Computing Machinery, New York, NY, USA, 2020, p. 1433–1443. URL: https://doi.org/10.1145/3368089.3417058. doi:10.1145/3368089.3417058.

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

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

[9] S. Nguyen, T. N. Nguyen, Y. Li, S. Wang, Combining program analysis and statistical language model for code statement completion, in: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, ASE '19, IEEE Press, 2020, p. 710–721. URL: https://doi.org/10.1109/ASE.2019.00072. doi:10.1109/ASE.2019.00072.

[10] M. Gabel, Z. Su, A study of the uniqueness of source code, in: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10, Association for Computing Machinery, New York, NY, USA, 2010, p. 147–156. URL: https://doi.org/10.1145/1882291.1882315. doi:10.1145/1882291.1882315.

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