Ranked Syntax Completion With LR Parsing

Kwanghoon Choi
Chonnam National University
Gwangju, Republic of Korea
kwanghoon.choi@jnu.ac.kr

Sooyeon Hwang
Chonnam National University
Gwangju, Republic of Korea
202918@jnu.ac.kr

Hyeon-Ah Moon
Sogang University
Seoul, Republic of Korea
hamoon@sogang.ac.kr

Isao Sasano
Shibaura Institute of Technology
Tokyo, Japan
sasano@sic.shibaura-it.ac.jp

ABSTRACT

This paper introduces a novel text-based syntax completion method that generates a sorted list of syntactic structure candidates for program writing. To date, no existing methods for syntax structure completion have offered candidates with accompanying rank information. We developed two key algorithms using LR parsing: one for collecting and ranking candidates, and another for querying them. With these algorithms, we gathered ranked candidates from SmallBasic programs in its community and from C11 programs in open-source software. We then assessed their effectiveness in code completion using Microsoft SmallBasic tutorial programs and the exercises from Kernighan and Ritchie's C programming language book. Our findings revealed that the top ranked candidate is frequently the correct choice. Furthermore, in over 96% of the cases, the correct completion is within the top 10 ranked candidates. This indicates the value of the collected rank information, assisting users in candidate selection during introductory programming tasks. Moreover, our method exhibits language-parametric characteristics; it can be applied to any programming language with syntax defined by an LR grammar.

CCS CONCEPTS

• **Software and its engineering** → **Parsers**; **Syntax**; Integrated and visual development environments.

KEYWORDS

Syntax completion, LR parsing, Ranks, Integrated development environments, Tools

ACM Reference Format:
Kwanghoon Choi, Sooyeon Hwang, Hyeon-Ah Moon, and Isao Sasano. 2024. Ranked Syntax Completion With LR Parsing. In *The 39th ACM/SIGAPP Symposium on Applied Computing (SAC '24), April 8–12, 2024, Avila, Spain.* ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3605098.3635944

SAC '24, April 8–12, 2024, Avila, Spain

ACM ISBN 979-8-4007-0243-3/24/04…$15.00
https://doi.org/10.1145/3605098.3635944

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.

In this study, we introduce a new text-based syntax completion technique that produces a ranked list of syntactic structure suggestions. Figure 1 illustrates an example of programming in SmallBasic, an educational programming language by Microsoft, aided by our suggested syntax completion technique. When a user queries suggestions at the cursor position on Line 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 method employs to gather these candidates.

Our approach aligns with the existing methods that present a list of syntax structure completion suggestions [17–19]. This stands in contrast to identifier completion techniques that recommend variable or function names [4–6, 15, 16, 20], as exemplified in Figure 2 for Microsoft SmallBasic. For instance, in the first example of Figure 1, the top suggestion at the cursor position (Line 4) is an assignment statement formatted as `ID = Expr`. Here, the terminal `ID` signifies an identifier, the terminal = represents an assignment symbol, and the non-terminal `Expr` indicates an expression. Further, the third example in Figure 1 delves deeper into the syntax structures of `Expr` after users input an identifier, `number`, followed by an assignment symbol. In contrast, Microsoft SmallBasic does not provide any suggestions at the same cursor position until the user starts typing the initial character `n`. Only then do suggestions appear, revealing one variable name, `number`, and one class name, `Network`, both beginning with that character.

Our method equips users with ranking information, making it more likely for them to select higher-ranked candidates. This contrasts with the existing methods [17–19], where the sequence of the suggested candidates is inconsequential. As illustrated in Figure 1, each candidate in the menus is labeled with numbers, indicating the frequency of each syntax structure candidate's appearance in the sample programs under investigation. Later analyses will demonstrate the utility of this feature for code completion.

There exists another distinction between our method and the existing methods [17–19] regarding the collection of candidates and their ranking information for code completion. Both approaches utilize the *LR parsing technique* to identify the so-called *suffix sentential forms* in computing candidates. Central to the LR parsing technique is the LR automaton, which, given a state and a lookahead, offers at most one parsing action. While the existing methods rely exclusively on the LR automaton to derive candidates, our approach varies. Imagine being provided with a parsing state at a cursor position for code completion. As the lookaheads remain unidentified, these methods must explore all possible combinations of a lookahead and a parsing action within the automaton's state. The computation of candidates using this strategy commences when users request code completion.

In contrast, our method leverages both the LR automaton and sample programs to gather candidates from the programs. Assuming that the examples are syntactically valid and have been successfully parsed, the lookaheads for all parsing states in the sample programs are known. These lookaheads then guide the LR automaton in determining which parsing actions to select next. Our method simply follows the parsing actions as directed by the lookaheads, selects the candidates, and tallies them to determine their ranking.

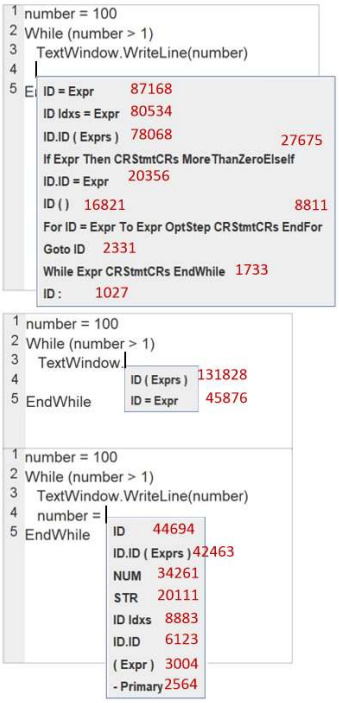


Figure 1: Syntax structure completion examples for SmallBasic in our system

To our knowledge, no existing methods for syntax structure completion provide candidates accompanied by ranking information. Only a few methods for identifier completion [5, 6, 15] have previously utilized rankings for that particular objective.

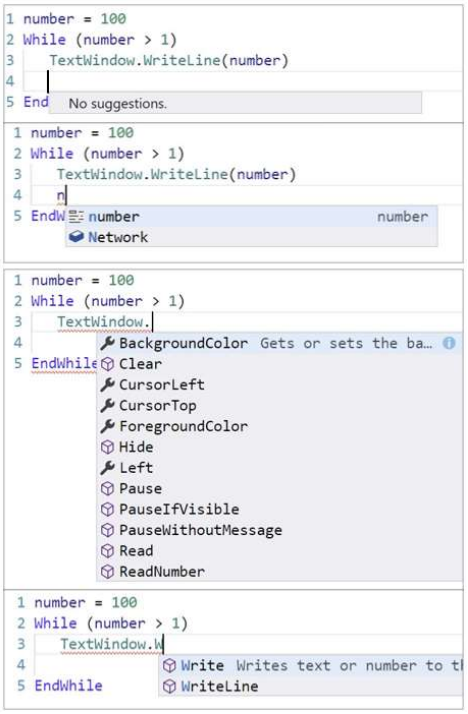


Figure 2: Identifier completion examples in Microsoft SmallBasic

A notable technical contribution of this work encompasses two algorithms using LR parsing. The first algorithm constructs a database from sample programs. This database maps LR states to syntax structure completion candidates with rankings. The second algorithm identifies matching LR states for a specified cursor position.

Moreover, our proposed method is language-parametric, implying it can be employed for any programming language, provided its syntax is definable by an LR grammar. Given that the syntax of C is represented by an LR grammar, our system can be utilized for syntax structure completion for C, as illustrated in Figure 4.

Our system comprises an editor and a syntax candidate database server, as depicted in Figure 3. The database is pre-constructed using the first algorithm mentioned earlier. While a user is coding and seeks code completion, the editor employs the second algorithm to identify a set of LR states for the cursor's position. In response, the server provides a ranked list of syntax completion candidates corresponding to those LR states.

We implemented our method and evaluated it using SmallBasic and C11 programs. Our analysis encompassed 3,701 SmallBasic programs totaling 789,023 lines from the Microsoft community and 412 C11 programs accounting for 308,599 lines from open-source software. Additionally, we prepared two sets of introductory programming materials: Microsoft SmallBasic tutorial programs and exercises from the renowned C programming language book by Kernighan and Ritchie. We then assessed how our analyzed, ranked syntax candidates database aids in suggesting syntax completions when crafting these introductory programs. Our findings revealed

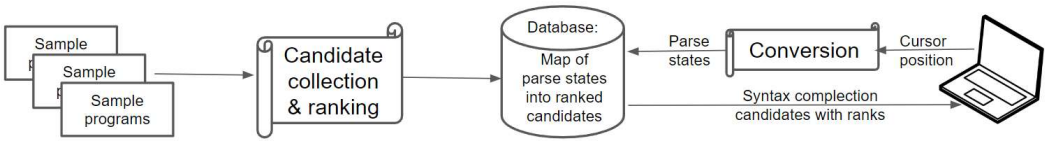


Figure 3: Overview of our system

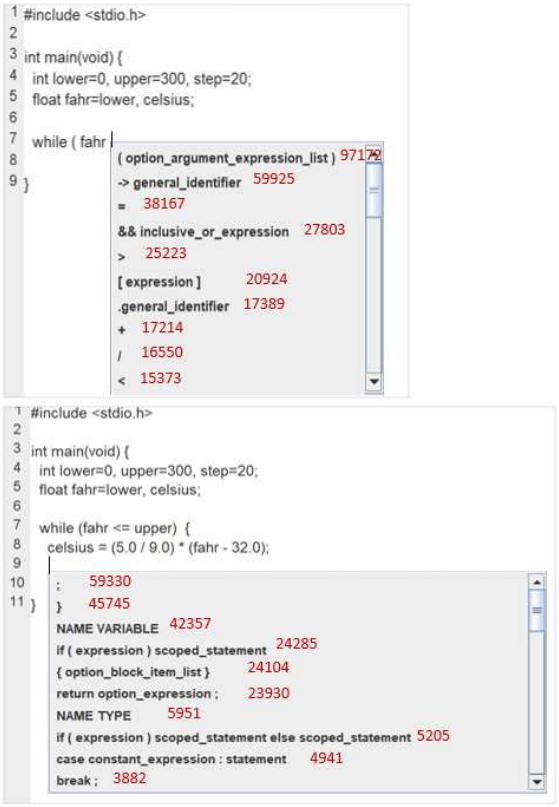


Figure 4: Syntax structure completion examples for C11 in our system

that the highest-ranked candidate is frequently the correct choice. Moreover, the correct completion is among the top 10 ranked candidates in over 96% of instances. These evaluation outcomes suggest that the compiled ranking data significantly enhances code completion.

The key contributions of this study are as follows: Firstly, we developed algorithms using an LR parser to collect and suggest ranked syntax completion candidates. Secondly, our evaluations with SmallBasic and C highlighted the importance of the ranking data in introductory programming. Thirdly, our tool is language-parametric, allowing automatic ranked syntax completion for any language defined by an LR grammar.

The structure of this paper is as follows: Section 2 introduces our system. In Section 3, we review a formal definition for candidates and outline two LR parsing-based algorithms. Section 4 presents an evaluation using SmallBasic and C11 programs. Section 5 offers a discussion on our study's results and relevant research. Concluding remarks are provided in Section 6.

2 OVERVIEW OF OUR SYSTEM

Figure 3 provides 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. While we will illustrate the underlying principles of the algorithms used in each phase with an example here, a detailed explanation of the algorithms will be provided in Section 3.

2.1 Collection and ranking phase

Given a set of syntactically valid sample programs in a particular programming language, the collection phase is broken down into three steps: lexing, parsing, and collecting candidates. Figure 5 displays a sample program, written in SmallBasic, an educational programming language by Microsoft. The table on the left presents an analysis result from the collection phase for this sample program.

2.1.1 Lexing. In the lexical analysis phase, the Hello World program

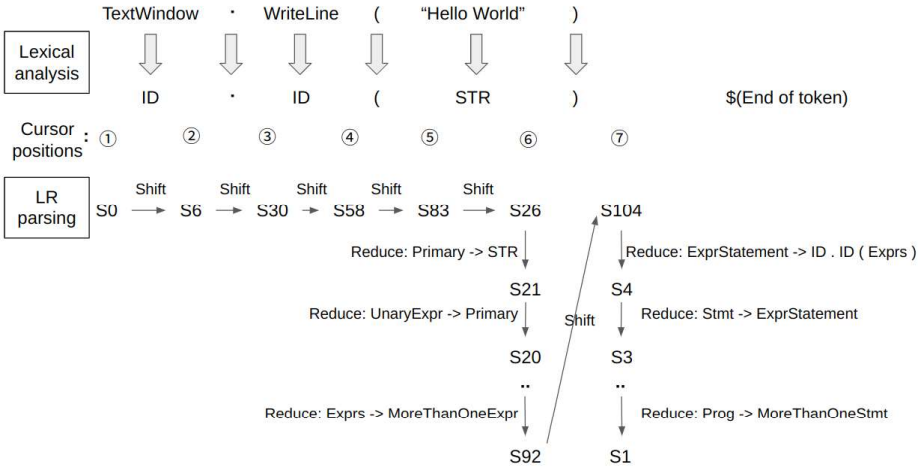
```
TextWindow.WriteLine("Hello World")
```

is tokenized into a stream,

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

as illustrated in Figure 5. The term `TextWindow` is processed into an identifier represented by the terminal `ID`, while a text dot is interpreted as a terminal dot. Similarly, `WriteLine` is analyzed like `TextWindow`. Open and close parentheses undergo the same procedure as the dot, and the string literal `"Hello World"` is translated into the terminal `STR`. It is worth noting that `ID` and `STR` are terminal names used in a lexical analyzer for SmallBasic grammar. After the analysis, the derived list of tokens (or terminals) is complemented by a special token `$`, indicating the end of tokens, which then feeds into an LR parser.

2.1.2 Parsing. Our system utilizes LR parsing, a predominant form of bottom-up parsing [1]. Here's a concise overview of the process: LR parsing operates through shift-reduce mechanisms. It uses a stack to store grammar symbols and an input buffer for the pending parsing string. Initially, the stack starts empty, and the entire input string awaits parsing in the buffer. The parser progresses left-to-right over the input, shifting symbols onto the stack. Once the string segment, denoted as β, atop the stack is ready for reduction, it is transformed to the head of the corresponding production rule,



States	Candidates	Ranks
S0	ID . ID (Exprs)	1
S6	. ID (Exprs)	1
S30	ID (Exprs)	1
S58	(Exprs)	1
S83	STR	1
S26, S21, S20, ...		
S92	)	1
S104, S4, S3, ..., S1		

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

Figure 5: Gathering and querying candidates using LR parsing (Hello World program in SmallBasic)

$A \rightarrow \beta$, by substituting β with A on the stack. The cycle repeats until only the start symbol remains on the stack, and the input buffer is depleted. This method essentially constructs the inverse of rightmost derivations, the primary objective of bottom-up parsing.

LR parsing utilizes LR automata derived from LR grammars. An LR automaton is comprised of states and edges. Each state represents a set of LR items, while edges signify the act of pushing a terminal or nonterminal onto a symbol stack. The automaton transitions between states via shift and reduce actions. A shift j action pushes the lookahead terminal onto the stack and transitions to state j. Conversely, a reduce action for $A \rightarrow \beta$ pops symbols corresponding to the production's right-hand side from the stack, pushes the left-hand side symbol, and then transitions to another state.

Let's delve into how the LR parser processes the Hello World program depicted in Figure 5. The parsing commences from the initial state, S0, at cursor position 1. Given the token `ID` as the initial lookahead, the LR automaton—derived from the SmallBasic grammar—guides the parser to shift 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.

Next, the automaton transitions from S6 to S30 based on the lookahead, the terminal dot, and adds it to the stack. This shift action continues until the parser reaches state S26 at cursor position 6, where it encounters a close parenthesis as the lookahead. At this point, it executes a reduce action for `STR` using the production `Primary` $\rightarrow$ `STR`, replacing the terminal `STR` with the nonterminal `Primary` on the stack. Without consuming any additional lookahead, the parser transitions to state S21, still at cursor position 6. Following this, it repeatedly applies reduce actions: the terminal `STR` is replaced with `Primary`, the nonterminal `Primary` with `UnaryExpr`, and so on, until the nonterminal `MoreThanOneExpr` is replaced with `Exprs`. Throughout this process, the parser arrives at state S92 without executing any further shift actions. The automaton now shifts based on the lookahead close parenthesis, transitioning to state S104 at cursor position 7. With no more tokens in sight except for the sentinel token signaling the end, the parser, at state S104, initiates the reduce action for the crucial production

`ExprStatement` $\rightarrow$ `ID . ID ( Exprs )`.

This production is pivotal as it triggers candidate collection from the sample program. Following a series of subsequent reduce actions, the automaton reaches state S1. Upon encountering the end of tokens, it signifies the successful parsing of the program. For a detailed information into the SmallBasic grammar and its corresponding LR automaton, refer to the companion website [2].

2.1.3 What are candidates? Code completion candidates can be intuitively defined using the concept of LR items [1]. An LR item is a production rule with a dot positioned at a certain location on the right-hand side, represented as $A \rightarrow \beta \cdot \gamma$. If a user writes a text that ends with the symbols β preceding the dot and requests

completion suggestions at the immediate subsequent position, γ serves as a candidate to complete β.

Consider state S6, which 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, = ]
```

Of all items, items 07 and 08 are pertinent to the cursor position (2) in the sample program. This position occurs right after the user has input `TextWindow`, i.e., `ID`. At this point, the anticipated subsequent input from the user is "`. WriteLine ( "Hello World" )`". The ideal completion suggestion should be "`. ID ( Exprs )`" from the two items.

Utilizing sample programs to identify candidates for code completion sets our method apart from previous approaches [18, 19]. Unlike earlier methods that exclusively relied on the LR automaton to deduce code completion candidates, we integrate insights from sample programs. At the same position by state S6, traditional methods would generate all conceivable candidates by examining symbols that can succeed centered dots in each item of the state. For instance, they might suggest "`:`" from items 1 and 2, "`= Expr`" from items 3 and 4, and "`. ID = Expr`" from items 5 and 6, among others.

A significant distinction lies in the nature of the methods employed. Previous studies [18, 19] are online, computing code completion candidates on-demand for specific program prefix positions. In contrast, our approach is offline, collecting candidates for all potential prefix positions of a particular sample program. We then construct a database of these pre-ranked candidates, which users can reference later.

2.1.4 Gathering and ranking candidates. For every candidate γ in an LR item represented as $A \rightarrow \beta \cdot \gamma$, the candidate is collected during LR parsing. Specifically it is gathered when the parser performs a reduce action on the production $A \rightarrow \beta\gamma$, following the shift and goto actions that push the symbols γ onto the stack from an originating parse state containing the LR item.

Consider the Hello World example program. Initially, LR parsing begins with the initial state S0 at cursor position 1. At this point, the prefix program text β_{S0} is empty, and we simulate a request to complete this prefix text. Continuing with LR parsing, we push the symbols γ_{S0}, which are "`ID . ID ( Exprs )`," onto the stack. This process continues until the automation reaches the state S104, where the parser reduces using the production "`ExprStatement → ID . ID ( Exprs )`." Subsequently, we identify this candidate γ_{S0} for the parse state S0. We then insert it into the initially empty map with the rank 1 as

$$[\ \mathrm{S0} \mapsto \{\ \texttt{ID . ID (Exprs)}^1\ \}\ ]$$

It is important to note that there are multiple reduce actions prior to reaching the parse state S104, such as "`Primary → STR`" at S26. However, this particular reduce action does not pop all the symbols that were previously pushed onto the stack down to the parse state S0. Therefore, it is not the type of reduce action we are looking for when gathering a candidate for S0.

Having completed the process for the parse state S0, we proceed to the next state, S6, by simulating a shift action with the lookahead `ID`. We then follow a similar procedure for the parse state S6 at the cursor position 2, as we did for S0. A candidate γ_{S6}, "`. ID ( Exprs )`," is identified when the automaton reaches S104 and performs a reduce action with the previously mentioned production. Consequently, the candidate map updates to:

$$[\ \mathrm{S0} \mapsto \{\ \texttt{ID . ID (Exprs)}^1\ \}, \mathrm{S6} \mapsto \{\ \texttt{. ID (Exprs)}^1\ \}\ ]$$

The process for the cursor positions 3 and 4 are the same as that for the cursor positions 1 and 2. To further illustrate, consider the process at cursor position 5, where the parse state is S83. The automaton executes a shift action using `STR`, transitioning to state 26. It then carries out a reduce action based on the production "`Primary → STR`." This action pops all the symbols, represented by γ_{S83}, `STR`, down to the state S83. As a result, `STR` is identified as a candidate for this state. The candidate map is subsequently updated as follows:

$$[\ \mathrm{S0} \mapsto \{\ \texttt{ID . ID (Exprs)}^1\ \}, \mathrm{S6} \mapsto \{\ \texttt{. ID (Exprs)}^1\ \}, \cdots \\ \mathrm{S83} \mapsto \{\ \texttt{STR}^1\ \}\ ]$$

This process outlines how the candidates in the left-hand side table of Figure 5 were assembled.

2.2 Query phase

During program editing, a user may request syntax completion candidates. The pre-constructed database from the collection phase can assist the user in finding these candidates. To facilitate this, the current cursor position where the user made a query must be translated into the relevant parse states, as illustrated in Figure 3.

During the query phase, the program text up to the designated cursor position is parsed using the LR parser. Unlike traditional parsers that return an error, our LR parser is designed to return the parse state where it halts. For instance, with cursor position 1 in Figure 5, 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 5.

Upon obtaining the set of parse states corresponding to the cursor position, the database 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 candidates presented in accordance with their ranks.

3 TWO ALGORITHMS

3.1 Specifications of candidates

This section examines the candidate specifications sourced from the previous research [18, 19]. They introduced the concept of *suffix sentential form*, which intuitively represents the remaining portion of the program text entered up to the cursor position.

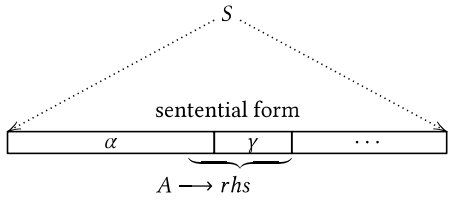


Figure 6: Definition of a simple candidate γ

Definition 1 (Suffix sentential forms [18, 19]). *Let α be a sequence of (terminal or nonterminal) symbols. The sequence β is referred to as a* suffix sentential form *with respect to α when $\alpha\beta$ is a sentential form.*

Users expect candidates to complete the syntax, meaning these candidates *must close* certain syntactic components in the program text up to the cursor position. Researchers formalized the definition of a *simple candidate* based on this observation.

Definition 2 (Simple candidates [18, 19]). *Let α be a prefix of a sentential form. A sequence of (terminal or nonterminal) symbols γ is a simple candidate with respect to α when the following conditions hold.*

(1) *γ is a prefix of a suffix sentential form with respect to α.*
(2) *A suffix of $\alpha\gamma$ constitutes the right-hand side of a production $A \longrightarrow rhs$ in the grammar, where $|rhs| \geq |\gamma|$[1] where $|s|$ denotes the length of a sequence of terminal or nonterminal symbols s.*

Intuitively, the first condition implies that when α is a prefix of a sentential form, γ does not introduce any syntax errors. The second condition indicates that γ is a suffix of the right-hand side (rhs) of a production, such that γ aids in *closing* some syntactic components in α, as expressed by $|rhs| \geq |\gamma|$. These conditions are illustrated in Fig. 6.

The concept of simple candidates serves as a crucial foundation but requires further exploration of practical syntax completion candidates, as discussed in the previous research [19]. A strategy presented in [19] suggests expanding the idea of simple candidates. This strategy involves considering a group of simple candidates, termed as *extended simple candidates*, which originate from different parse states but share the same cursor position, as demonstrated in the given example.

Let us illustrate the concept of extended simple candidates using an example. In Figure 5, after users input the program text prefix:

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

it is assumed that users request code completion at cursor position 6. They might anticipate a closing parenthesis, `)`, as a candidate. However, according to the definition, no simple candidates exist for the parse state S26, which corresponds to cursor position 6, as depicted in the figure's left-hand side table. Instead, it is the parse state 92 that offers this candidate as a simple candidate. This parse state is reached by the LR automaton after navigating through states S26, S21, S20 and so forth. All these parse states, from S26 to S92, form a sequence of parse states where no shift actions occur.

[1] In [18, 19], the original definition of simple candidates was presented with a weaker condition, represented as $|rhs| > |\gamma|$. Later this definition was expanded to $|rhs| \geq |\gamma|$ and termed as *extended simple candidates* in [19].

In simpler terms, no additional tokens are shifted from the token stream to the stack. These states are deemed *equivalent* concerning cursor position 6. The extended simple candidates concept recognizes this distinction by considering all parse states at the same cursor position. The concept of a set of *cursor position equivalent* parse states can be characterized formally by *right-end derivations* and *reductions* [19].

Our algorithms for gathering and querying candidates are designed based on the concept of extended simple candidates.

3.2 Two algorithms for collecting and querying extended simple candidates

Algorithm 1 Collecting and ranking extended simple candidates

```
function sample(state, logs, map)
    candidate ← sampleOne(state, logs)
    When map(state) is not defined, let map′ be map with the
      correspondence state ↦ {candidate^1} added.
    Otherwise, let {candidate^n} ∪ S = map(state).
    Then let map′ be map with the value of map(state) updated
      to {candidate^{n+1}} ∪ S.
    if logs are SHIFT state′ terminal : logs′ then
        return sample(state′, logs′, map′)
    if logs are REDUCE A → RHS : GOTO state′ : logs′ then
        return sample(state′, logs′, map′)
    if logs are ACCEPT : logs′ then
        return map′
function sampleOne(state, logs)
    if logs are REDUCE A → RHS : GOTO state′ : logs′ then
        return []
    return sampleSymbols(state, logs, [])
function sampleSymbols(state, logs, symbols)
    if logs are SHIFT state′ terminal : logs′ then
        return
            sampleSymbols(state′, logs′, symbols · terminal)
    if logs are REDUCE A → RHS : GOTO state′ : logs′ then
        if |RHS| ≥ |symbols| then
            return symbols
        else
            perform a reduce action with A ← RHS over symbols
            let symbols_1 be the resulting stack
            let state_1 be the resulting state
            return sampleSymbols(state_1, logs′, symbols_1 · A)
    if logs are ACCEPT : logs′ then
        return symbols
```

Our system utilizes two algorithms for candidate collection and conversion, as depicted in Figure 3. Algorithm 1 is responsible for preparing ranked candidates in advance, while Algorithm 2 is triggered whenever users seek code completion.

Algorithm 1 serves as a candidate collection algorithm. It takes a parse state, a list of LR parse actions denoted as *logs*, and a mapping of parse states to sets of ranked candidates as its input. The algorithm then returns an updated mapping as its output.

Algorithm 2 Computing a set of parse states equivalent to a parse state *state* under the corresponding stack *stack*

```
function CurrentStates(state, stack)
    result ← ∅
    let PRD be the set of all productions A ← rhs used in
      reduce actions found at the parse state state.
    for A ← rhs ∈ PRD do
        do a reduce action with A ← rhs over stack
        let stack_1 be the resulting stack
        let state_1 be the resulting state
        result ← result ∪ CurrentStates(state_1, stack_1)
    return {state} ∪ result
```

It is assumed that all example programs from which candidates are collected have been successfully parsed by LR parsing. Consequently, each of these example programs is parsed by an LR parser, producing a list of LR parse actions that concludes with the *ACCEPT* action. To start the process, the *SAMPLE* function is invoked with the initial parse state S0, the aforementioned list of parse actions, and an empty map. Each parse action in the logs adopts the format: *SHIFT state lookahead*, *REDUCE* $A \rightarrow RHS$ followed by *GOTO state*, and *ACCEPT*. Note that `(:)` is a cons operator and `[]` is the nil for list operations. Within this algorithm, the main function, *SAMPLE*, calls *SAMPLEONE* to acquire a single candidate for each parse state, subsequently advancing to the next parse state.

The auxiliary function *SAMPLESYMBOLS* retrieves terminals from shift actions and nonterminals from reduce actions based on the parse action *logs*. These are then appended to the variable *symbols* using the operator $(\cdot)$ for adding a symbol to the end of a list of symbols. This procedure persists until either the condition $|rhs| \geq \gamma$ specified for candidates is met, or equivalently, when $|RHS| \geq |symbols|$ in the algorithm, pinpointing closing candidates.

Algorithm 2 is designed to transform cursor positions into sets of parse states. Initially, our system employs LR parsing on the user's program text, starting from the beginning and extending to the cursor position, with the objective of identifying a parse state at which the LR parser gets stuck. Subsequently, Algorithm 2 leverages both the discovered parse state and the stack to deduce a set of parse states. These derived states correspond to the cursor position. Note that Algorithm 2 relies on an automaton table. This table maps pairs consisting of parse states and lookaheads to parse actions, enabling the algorithm to determine a set of all productions associated with the reduce actions for a specified parse state.

3.3 Implementation

We have confidence in the correctness of two algorithms that our system computes ranked syntax completion candidates, as outlined in the specifications for candidates. While we view the formal argument as a potential subject for future exploration, our current focus has been on implementing these algorithms.

Our implementation underwent evaluation using SmallBasic and C11, both equipped with LR grammars. It is based on a parser builder tool named *YAPB* [9]. For developers aiming to create a language-specific syntax completion tool, the only requirements are writing a lexer specification and a parser specification tailored to their programming language. These specifications were written to align with YAPB's interface, as have been demonstrated in [9, 18, 19]. Upon completion of writing the parser specifications, the tool is furnished with a candidate collector and and a cursor position converter tailored to the chosen programming language, using LR parsing.

We have implemented syntax completion systems for SmallBasic and C11 using YAPB. Supplementary materials, encompassing the LR grammars, LALR automatons, learning sets of programs, test sets of programs, syntax completion databases, and evaluation results, can be found in [2]. Our implementation has adopted the SmallBasic and C11 grammars in [3, 7]. While it is recognized that adapting the grammars can enhance the quality of code completion candidates, as discussed in [19], we chose not to modify the grammars for our evaluation.

Every candidate that users select might contain one or more terminals and nonterminals. The presentation of a candidate after its selection is also crucial, even though our research primarily focuses on suggesting ranked candidates. For nonterminals, the system could represent them with triple dots as seen in [24,25], leave some space, or offer users the opportunity to navigate a tree structure with the nonterminals as the root based on the grammar. Users can then expand some nonterminals by corresponding productions. As for terminals, users might be prompted to input text for them until the system either successfully validates it against the terminal's regular expression or the users decide to discontinue their input.

Our system mandates that each program text prefix remain *syntactically valid*, a constraint that might be deemed restrictive. Allowing for syntax errors before the cursor, which aligns with more realistic scenarios, could be feasible if we consider an error recovery mechanism in place of strict LR parsers. Exploring this avenue would be an interesting future work.

4 EVALUATION

In this study, we evaluate the proposed system to address two research questions:

- RQ1: Does the system offer candidate suggestions in an order beneficial for introductory programming?
- RQ2: Is it feasible to implement the system as a language-parametric tool?

To address the research questions, our methodology involves selecting two programming languages, collecting syntax completion candidates from existing programs in these languages, choosing sets of widely recognized introductory programming examples, and assessing the efficacy of this approach in suggesting candidates and their order.

We selected Microsoft SmallBasic [10] and C [8], both of which are defined by LR grammars. Microsoft SmallBasic is designed for coding education. An LALR(1) parser for SmallBasic, utilizing YAPB, is available at [2]. C is a widely portable programming language known for its pointer features. We developed a parser in line with the C11 standard [8] using YAPB, which is available at [2].

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.

Table 1: SmallBasic and C grammar statistics

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

For each programming language, we prepared both a learning set and a test set of pre-existing programs. From the learning set, we established a database by gathering syntax completion candidates, accompanied by their occurrence counts. We assume that syntax completion candidates derived from the test set represent the intended code users aim to write at specified cursor positions.

For SmallBasic, we derived our learning set from its community, comprising 3,701 programs that total 789,023 lines. The test set, on the other hand, consists of 27 programs spanning 155 lines, sourced from the renowned Microsoft SmallBasic tutorial. This community serves as the most extensive repository of SmallBasic programs. An example of a SmallBasic program, as seen in Figure 1, is drawn from the tutorial. Regarding C, our learning set originates from several notable open-source programs: 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. Combined, the sources encompass 308,599 lines. The test set is an aggregation of 106 programs (totaling 11,218 lines), which are solutions from the widely recognized Kernighan and Ritchie's book on the C programming language. All the programs are in [2].

Table 2: Statistics for the number of typing the down key to choose syntax structures as desired

SmallBasic											
# of ↓	0	1	2	3	4	5	6	7	8	9	10
# of occs.	354	141	49	29	8	6	8	4	1	0	2

C										
# of ↓	0	1	2	3	4	5	6	7	8	9
# of occs.	1009	465	221	184	144	92	72	49	54	33
# of ↓	10	11	12	13	14	15	16	17	18	19
# of occs.	32	15	10	1	12	2	0	1	14	0
# of ↓	20	21	22	23	24	25	26	27	28	29
# of occs.	0	1	1	1	0	0	0	0	0	1

We present a summary of the results from gathering syntax completion candidates for both SmallBasic and C. For SmallBasic, the database contains 324 candidates, covering all parse states with the exception of 38 states that have no candidates. On average, each parse state has 4.0 candidates with a standard deviation 3.2. The highest number of candidates for any parse state is 11. In contrast, the C database features 850 candidates that span all parse states, barring 336 states without candidates. Each parse state averages 4.4 candidates, a figure close to that of SmallBasic, but with a higher standard deviation 5.4. The maximum number of candidates for any parse states is observed to be 35. Further details can be found in the respective repositories [2].

There are parse states where no candidates are found, and this can be attributed to two reasons. First, the example programs in the learning set may not encompass these parse states. Second, the definition of simple candidates inherently permits situations where no candidates are present.

Based on the evidence provided, we can confidently answer Research Question 1 in the affirmative. Two main points support this conclusion: For SmallBasic, we confirmed that the syntax structures needed for all 602 cursor positions from the tutorial programs align with the syntax structure candidate database we compiled using community-contributed SmallBasic programs. In the case of C, nearly all syntaxes for the 2,416 cursor positions match those in the syntax structure candidate database derived from C programs, which facilitates their use in solution C programs. Notably only two candidates were absent in the database.

Secondly, our findings showed that the top suggestion in the syntax candidate list was consistently the most selected choice. This means users frequently did not need to scroll or use the down key to navigate the list. This trend was evident in 354 instances from the tutorial SmallBasic programs and in 1,009 instances from the C solution programs.

Table 2 categorizes the frequency of pressing the down key required to locate the desired syntax candidates within the suggested list for the tutorial SmallBasic programs. On average, finding the desired syntax candidates for the SmallBasic tutorial programs necessitated pressing the down key only 0.8 times. For the K&R C solution programs, this average was slightly higher at 2.15 times.

We also assessed the number of pages flips users needed to find the desired syntax completion candidates for the test programs. Our text editor displays a pop up menu showing 10 syntax candidates per page. For the tutorial SmallBasic programs, users found 99.7% of the required syntaxes within the first page listing the top 10 candidates. For the C solution programs, this figure stood at 96.2%. In only 2 cursor positions within the SmallBasic programs were the desired candidates not found on the first page. Similarly, in 91 out of 2,416 cursor positions for the C solution programs, the required candidates were not on the first page.

All these experimental findings confirmed that for typical introductory programming examples in SmallBasic and C, the desired syntax candidates can be readily located when organized by candidate rankings.

Research Question 2 can be addressed by noting that our implementation never be dependent on specific programming languages. Instead, it exclusively uses traces of parse actions and automaton tables from LR parsing, as evidenced by our evaluations with two distinct programming languages.

A potential threat to validity exists in our approach: For Smallbasic, we sourced nearly all community programs. In contrast, the C11 programs we gathered are somewhat arbitrary. The selection of example programs could influence the code completion results.

5 DISCUSSIONS AND RELATED WORK

The idea of ranking candidates stems from the challenges of using code completion without ranking. We highlight studies on these

challenges and then discuss research on ranking, particularly for identifier completion, and other relevant studies.

5.1 Ranking candidates

Several studies have been conducted on ranking candidates, primarily focusing on identifier completion. These studies quantify the occurrence of elements, such as identifiers, member names, token sequences, and combinations of terminal and non-terminal symbols, in source code repositories like GitHub or the code presently under development.

A study [15] investigated ranking candidates for identifier completion using program editing history. This research stored every editing activity undertaken by a programmer. They highlighted a common challenge: when writing Java programs in Eclipse, for instance, the number of candidates for a given identifier prefix can be overwhelming. These candidates are displayed in an alphabetical order in a popup window. Consequently, a programmer may need to scroll extensively to find the expected candidate, which can be more time-consuming than simply typing out the full identifier. The situation in syntax completion mirrors this scenario described in [15], with an excessive number of candidates in certain instances.

Another study by [5] also explored ranking strategies in identifier completion. Similar to [15], they sought solutions to improve the efficiency of the completion process. Rather than relying on prefix matching, they introduced subsequence matching. In this approach, user-input sequences of characters are compared to names containing those sequences, even if they are non-consecutive. For instance, typing "swu" would yield candidates such as "SwingUtilities" and "SetWrapGuidePainted". They termed this as an "acronym-like" input method. The researchers noted that when humans use acronyms, they follow certain patterns. For instance, typing "swut" is more likely to refer to "SwingUtilities" than to "ShowFullPath". They claimed that by leveraging these acronym usage tendencies, the ranking in identifier completion could be optimized. To achieve this, they employed a machine learning model, specifically the support vector machine (SVM), to rank identifiers. They trained the SVM using a large code base. While their use of a substantial code base aligns with our methodology, our techniques differ fundamentally. Our approach focuses on collecting LR parsing states for syntax completion, whereas theirs centers on training the SVM for identifier completion.

Recently a study by [6] delved into method invocation and field access completion. They introduced what they termed as *project-specific* candidates, which they derived through a combination of heuristics and neural network models. Using open-source Java programs for their empirical evaluation, they discovered that over half of member accesses pertained to members of classes declared within the same project. They delved into a specific scenario where a user is writing an incomplete Java assignment statement. The incomplete assignment should have the form "$c\ n\ =\ e.$" where c represents a class name, n an identifier, and e an expression with a type corresponding to a class declared within the project. Following the dot, they proposed member names as completion candidates, ranking them based on the frequency of each member's occurrence within the current project source code.

In the study by [13], the focus was on synthesizing method calls to fill *holes* within programs. They viewed this synthesis as akin to a natural language processing problem, wherein they calculated the likelihood of potential token sequences that would form these method call expressions. Using a comprehensive codebase sourced from GitHub and other platforms, they ranked candidates according to their estimated probabilities. Their algorithm incorporated N-gram analysis, considering the frequency of sequences of N tokens to aid in candidate determination.

Recently, Svyatkovskiy et al. from Microsoft introduced a system named *IntelliCode Compose* [21]. This system leverages GPT-C, a variant of OpenAI's GPT-2 [12], trained on a vast dataset of program source code. It is designed to generate sequences of tokens that form syntactically correct language constructs — like statements containing local variables, method names, and keywords — for languages, including C#. Their method is based on the probability distribution of consecutive token sequences of some fixed length, wih relatively counting frequencies of token sequences in the code base. The top sequence, in terms of the probability, is presented as their *suggestion*, derived from a model trained on extensive program source code. It is worth noting the contrast between their system and ours: while they generate sequences of tokens (terminal symbols), ours generate sequences of both terminal and non-terminal symbols. This distinction means that in our system, programmers would have the potential to further expand non-terminal symbols.

5.2 Parser-based code completion

Several studies have explored the use of parsers for code completion, including identifiers and syntax structures (comprising both terminal and non-terminal symbols).

A study [22] employed the parser generator ANTLR [11], developed by Parr, for code completion. This approach allows code completion functionality to be derived from an ANTLR syntax description. Some aspects of their generation methodology align with ours, such as: syntax-based generation, utilizing internal parser information, and initiating code completion on detecting a syntax error. However, a key distinction lies in the nature of the completion candidates: their approach suggests individual tokens, whereas ours proposes sequences of both terminal and non-terminal symbols.

Some other studies have delved into syntax completion using a parser. Rekers et al. [14] introduced a substring parser based on GLR parsing [23]. This substring parser, designed for a language L defined by any context-free grammar, takes a string s and constructs a parse tree for a sentential form vsw in L, with v and w serving as completion candidates for s. Their idea of syntax completion closely aligns with ours. However, a key distinction is that their completed strings are strictly sentential forms, while in our approach, they can be prefixes of sentential forms. This allows for more flexibility in our method, especially when dealing with partial programs having nested syntax. Moreover, their research was limited to the Pascal language and did not have the ranking of candidates.

The study [19] explored the use of an LALR parser to determine syntactic candidates from a prefix of a valid program. The research employed an LALR(1) parser generator called YAPB [9]. This tool creates an LR parser with the capability to access its internal details. However, the research did not consider ranking the candidates.

6 CONCLUSIONS

In this research, we introduced a text-based syntax completion method that produces a ranked list of syntax structure candidates for coding. We assessed this method using two programming languages to verify its efficacy in suggesting candidates and prioritizing them for introductory programming.

In the user's view, there is considerable scope for further refinement and exploration. While the proposed candidates in this study focus on syntax structures, such as `ID . ID ( Exprs )` with terminal and nonterminal symbols from the grammar, a more intuitive presentation would embellish these with actual identifiers and concrete expressions. For instance, replacing the first identifier with a class name like `TextWindow`, the second with a function name like `WriteLine`, and the expressions with placeholders would enhance clarity. Displaying nonterminal names directly in the user interface may not be beneficial. Instead, using a representation like `"..."` as shown in [17–19] might be more user-friendly. Only very high-level nonterminal names such as "`expr`", "`stmt`", or "`decl`" might be readily comprehensible to users.

From a user's viewpoint, an integrated completion system would be more intuitive and streamlined. While identifier completion operates separately from syntax structure completion, they complement each other, making their integration feasible. Incorporating positional information could further enhance this combined approach. For instance, in the given syntax structure of SmallBasic, the initial instance of `ID` is intended for class names, not functions or variable names. Conversely, the subsequent `ID` should represent a function name, rather than a class or variable.

At times, users may favor more intricate suggestions. For instance, in a functional programming context, on writing "`let val add = fn x =>`", the suggested candidates could extend beyond merely completing the inner function (`fn`), to include suggestions for concluding the encompassing `let` block, as in `expr in expr end`. Our system suggests extended simple candidates. We posit that simpler suggestions are often be more beneficial than intricate ones. Furthermore, one could construct intricate suggestions by combining several simple ones. In this context, offering simpler candidates might suffice.

ACKNOWLEDGEMENT

This work was partially supported by JSPS KAKENHI under Grant Number 20K11752 and 23K11053, by the Korea Internet & Security Agency (KISA) - Information Security College Support Project, and by NRF of Korea funded by the MoE No. 2019R1I1A3A01058608.

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers — principles, techniques, and tools, 2nd edition.* Addison Wesley.

[2] Anonymized authors. 2023. Implementations and Evaluation Results with SmallBasic and C11. https://drive.google.com/drive/folders/1UqSN3qnhn9eSW6IgxAUsRAfQN_ZpAMH?usp=sharing.

[3] Kwanghoon Choi, Gayoung Kim, and Byeong-Mo Chang. 2018. A Development of Open-Source Software for Educational Coding Environments Using Small Basic. *KIISE Transactions on Computing Practices* 24, 12 (2018), 649–661. https://doi.org/10.5626/KTCP.2018.24.12.649

[4] Takumi Goto and Isao Sasano. 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 (PEPM '12)*. ACM Press, Philadelphia, Pennsylvania, USA, 131–140. https://doi.org/10.1145/2103746.2103771

[5] Sheng Hu, Chuan Xiao, and Yoshiharu Ishikawa. 2019. Scope-aware Code Completion with Discriminative Modeling. *Journal of Information Processing* 27 (2019), 469–478. https://doi.org/10.2197/ipsjjip.27.469

[6] Lin Jiang, Hui Liu, He Jiang, Lu Zhang, and Hong Mei. 2022. Heuristic and Neural Network Based Prediction of Project-Specific API Member Access. *IEEE Transactions on Software Engineering* 48, 4 (2022), 1249–1267. https://doi.org/10.1109/TSE.2020.3017794

[7] Jacques-Henri Jourdan and François Pottier. 2017. A Simple, Possibly Correct LR Parser for C11. *ACM Trans. Program. Lang. Syst.* 39, 4, Article 14 (sep 2017), 36 pages. https://doi.org/10.1145/3064848

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

[9] Jintaeck Lim, Gayoung Kim, Seunghyun Shin, Kwanghoon Choi, and Iksoon Kim. 2020. Parser generators sharing LR automaton generators and accepting general purpose programming language-based specifications. *Journal of KIISE* 47, 1 (2020), 52–60. https://doi.org/10.5626/JOK.2020.47.1.52 (in Korean).

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

[11] Terence Parr and Kathleen Fisher. 2011. LL(*): The foundation of the ANTLR parser generator. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. 425–436. https://doi.org/10.1145/1993498.1993548

[12] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2018. Language Models are Unsupervised Multitask Learners. https://paperswithcode.com/paper/language-models-are-unsupervised-multitask.

[13] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 419–428. https://doi.org/10.1145/2594291.2594321

[14] Jan Rekers and Wilco Koorn. 1991. Substring Parsing for Arbitrary Context-Free Grammars. *SIGPLAN Not.* 26, 5 (may 1991), 59–66. https://doi.org/10.1145/122501.122505

[15] Romain Robbes and Michele Lanza. 2008. How program history can improve code completion. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE '08)*. 317–326. https://doi.org/10.1109/ASE.2008.42

[16] Isao Sasano. 2014. Toward modular implementation of practical identifier completion on incomplete program text. In *Proceedings of the 8th International Conference on Bioinspired Information and Communications Technologies (BICT '14)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), Boston, Massachusetts, 231–234. https://doi.org/10.4108/icst.bict.2014.257909

[17] Isao Sasano. 2020. An approach to generate text-based IDEs for syntax completion based on syntax specification. In *Proceedings of the 2020 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (New Orleans, LA, USA) *(PEPM 2020)*. Association for Computing Machinery, New York, NY, USA, 38–44. https://doi.org/10.1145/3372884.3373158

[18] Isao Sasano and Kwanghoon Choi. 2021. A Text-Based Syntax Completion Method Using LR Parsing. In *Proceedings of the 2021 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (Virtual, Denmark) *(PEPM 2021)*. Association for Computing Machinery, New York, NY, USA, 32–43. https://doi.org/10.1145/3441296.3441395

[19] Isao Sasano and Kwanghoon Choi. 2023. A Text-Based Syntax Completion Method using LR Parsing and Its Evaluation. *Science of Computer Programming* (2023), 102957. https://doi.org/10.1016/j.scico.2023.102957

[20] Isao Sasano and Takumi Goto. 2013. An approach to completing variable names for implicitly typed functional languages. *Higher-Order and Symbolic Computation* 25, 1 (2013), 127–163. https://doi.org/10.1007/s10990-013-9095-x

[21] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 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* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1433–1443. https://doi.org/10.1145/3368089.3417058

[22] Federico Tomassetti. 2016. Building autocompletion for an editor based on ANTLR. https://tomassetti.me/autocompletion-editor-antlr/.

[23] Masaru Tomita. 1985. *Efficient parsing for natural language: A fast algorithm for practical systems.* Kluwer Academic Publishers. https://doi.org/10.1007/978-1-4757-1885-0