# A Text-Based Syntax Completion Method using LR Parsing

Isao Sasano
Department of Computer Science and Engineering
Shibaura Institute of Technology
Tokyo, Japan
sasano@sic.shibaura-it.ac.jp

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

## Abstract

This paper presents a text-based syntax completion method using an LR parser. We propose formal definitions of candidate text to be completed based on the sentential forms, and we design algorithms for computing candidates through reductions in the LR parsing. This is in contrast to the existing methods that have not clearly stated what candidates they intend to produce. This is also different from a transformation approach using an LR parser, which transforms the grammar of the programming language, a burdensome task at this moment. The advantage of our method is that LR parsers can be adopted without modification, and a syntax completion system can be built using them, without incurring efforts. We implemented the algorithms as an Emacs server to demonstrate the feasibility of their application.

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

***Keywords:*** syntax completion, LR parsing, parser generator, sentential forms, reduction, integrated development environments

**ACM Reference Format:**
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 (PEPM '21), January 18–19, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3441296.3441395


*PEPM '21, January 18–19, 2021, Virtual, Denmark*

ACM ISBN 978-1-4503-8305-9/21/01…$15.00
https://doi.org/10.1145/3441296.3441395

## 1 Introduction

Integrated development environments (IDEs) such as Eclipse provide syntax completion, which is used for editing an incomplete program text. Developers of an IDE have to implement features, such as syntax completion, for each language. It reduces cost to help developers to implement such features from a specification in a systematic way .

In this study, we investigate how to generate syntactic candidates for completion from the grammar of the language. For simplicity, we only use the program text up to the cursor position on the editor.

We describe this idea using a running example when assuming grammar of a subset of Standard ML as follows:

| | | |
|---|---|---|
| *start* | := | *exp* |
| *exp* | := | *appexp* |
| | \| | `fn ID =>` *exp* |
| *appexp* | := | *atexp* |
| | \| | *appexp atexp* |
| *atexp* | := | `ID` |
| | \| | `(`*exp*`)` |
| | \| | `let` *dec* `in` *exp* `end` |
| *dec* | := | `val ID =` *exp* |
| | \| | `fun ID ID =` *exp* |

Here, *start* is the start symbol. The symbols in italics represent non-terminal symbols, whereas the remaining symbols directly represent the terminal symbols or lexemes. In addition, `ID` represents an identifier, `fn ID =>` *exp* represents a function abstraction, *appexp atexp* represents a function application, and `let` *dec* `in` *exp* `end` represents a let expression, where *dec* is a declaration. There are two types of declarations, *i.e.*, value and function declarations. We included the parenthesized expression `(`*exp*`)` in the syntax to express situations in which open parentheses that are not yet closed are present.

When the program text up to the cursor position is

```
let val add = fn x =>,
```

a program fragment '`fn x =>`' is a partially input function abstraction. A candidate may be *exp*, which makes the function abstraction complete. This is defined as a simple candidate in Section 2.

Because the let expression is also partially input, users may expect a candidate including `in` and `end`. Thus, we consider *nested candidates*. For the program text above, there are two nested candidates, which will be described in Section 4.

The formal treatment of syntax-aware editing has been supported by so-called structured or projectional editors, such as Synthesizer Generator [23] and MENTOR [8]. The program source code is represented as a tree structure in a structured editor. A tree structure is maintained by restructuring it when the program is modified. The behaviors of the editors can be formally specified, and some features, such as the syntax completion, can be easily implemented. We say an editor is *text-based* when it does not prevent the programmers from editing the program text on a character basis. When compared with text-based editors, the majority of structured editors prevent programmers from freely editing the program text, with the exception of a system called LRC, implemented by Kuiper *et al.* [14, 27]. Owing to such inconvenience, programmers may not like to apply structured editors for editing the program source code despite their suitability for editing structured data apart from the program source code.

An alternative approach may be to use the internal information of the LR parser, which was recently taken by a system called Merlin [3]. Merlin provides various functionalities, including syntax completion, for OCaml language, by using the internal information of the LR parser. The idea is basically to fill in the missing part of the partial AST, possibly with error recovery, although technical details about filling in the missing part were not given.

The study presented herein takes this approach to propose an algorithm for generating syntax completion functionality from a grammar using the internal information of the LR parser and argues the technical details. Based on the algorithm we implement a system for syntax completion by using a general-purpose LR parser generator [16], which, given a grammar, builds an LR parser that can access the internal information of the parser. We extend the LR parser generator with a functionality to compute candidates for a syntax completion. The extended LR parser generator automatically provides a syntax completion system, given a specific grammar.

We herein specify syntactic candidates to be completed in terms of a given grammar for a particular language. Subsequently, we present algorithms based on LR parsing, which enable completion on text-based editors. Furthermore, we implement a completion system for Emacs.

The contributions of this paper are as follows.

- We proposed a formal definition of simple and nested candidates for a text-based syntax completion system.
- We designed algorithms for computing the syntax completion candidates based on an LR parser.
- To show its feasibility, we implemented the algorithm as a server that interacts with Emacs.
- We developed a tool to provide a text-based syntax completion system automatically from a specific grammar.

The rest of this paper is organized as follows. In Section 2, we propose a formal definition for simple candidates. In Section 3, we design an LR parser based algorithm for simple candidates. Section 4 extends the definition to nested candidates, and describes a backtracking algorithm used for computing such candidates. Section 5 describes an implementation for the algorithms. After a discussion in Section 6 and related studies in Section 7, we conclude with remarks in Section 8.

## 2 Specifications of Simple Candidates

In this section we specify simple candidates. First, the *postfix sentential form*, which intuitively corresponds to the remaining part of the program text being input up to the cursor position, is defined.

**Definition 2.1** (Postfix sentential forms)**.** Let $\alpha$ be a sequence of (terminal or non-terminal) symbols. When $\alpha\beta$ is a sentential form, the sequence $\beta$ is referred to as a *postfix sentential form* with respect to $\alpha$.

Intuitively, the candidates that the users expect to complete with are those that *must follow* or *close* some of the syntactic components in the program text up to the cursor position. First, we formalize the definition of a *simple candidate* based on this observation. Subsequently, we design an algorithm for computing the candidate. Later, we use this definition as a basic block for constructing larger candidates to be defined as *nested candidates*.

**Definition 2.2** (Simple candidates)**.** Let $\alpha$ be a prefix of a sentential form. A sequence of (terminal or non-terminal) symbols $\gamma$ is a simple candidate with respect to $\alpha$ when the following conditions hold.

1. $\gamma$ is a prefix of a postfix sentential form with respect to $\alpha$.
2. A postfix of $\alpha\gamma$ constitutes the right-hand side of a production $A \longrightarrow rhs$ in the grammar, where $|rhs| > |\gamma|$.

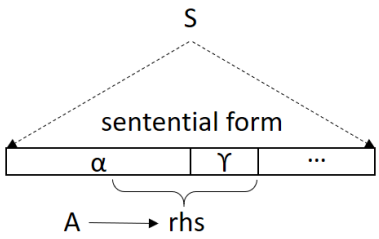


**Figure 1.** Definition of a simple candidate $\gamma$

In this definition, $|s|$ represents the length of a sequence of (terminal or non-terminal) symbols $s$. Intuitively, the first condition states that $\gamma$ does not introduce any syntax errors when $\alpha$ is a prefix of a sentential form. The second condition states that $\gamma$ is a postfix of the right hand side (rhs) of a production such that $\gamma$ plays a role in *closing* some of the syntactic components in $\alpha$, i.e, $|rhs| > |\gamma|$. The conditions are illustrated in Fig. 1.

Note that there may be more than one simple candidate $\gamma$ for a sequence $\alpha$. For example, let us consider the following grammar of an arithmetic language.

$$\begin{array}{lcl} E & := & A \mid E + A \mid E - A \\ A & := & \texttt{num} \mid (E) \end{array}$$

Here, num is a sequence of digits. When the program text up to the cursor position is '(2+3', there are three simple candidates.

$$+A,\ -A,\ )$$

These candidates are popped up on Emacs as in Fig. 2, where

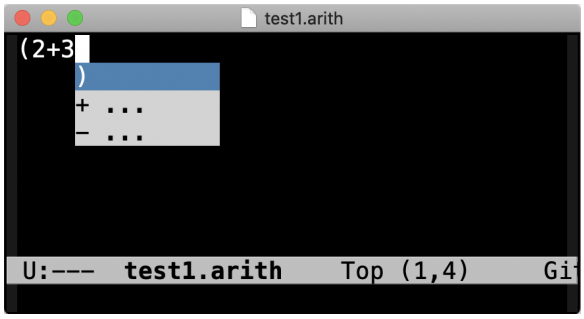


**Figure 2.** A screen shot of the three simple candidates popped up for the program text '(2+3' in an arithmetic language

$A$ is replaced with the ellipsis. The implementation will be discussed in Section 5.

First, we describe why they are the simple candidates for '(2+3'. Let us start by deriving sentential forms constrained by the program text, from the starting symbol, $E$. Of the three productions with $E$ on the left-hand side, consider $E \longrightarrow A$ and then $A \longrightarrow (E)$ resulting in the following derivation.

$$E \Rightarrow A \Rightarrow (E)$$

We must now choose either $E \longrightarrow E{+}A$ or $E \longrightarrow E{-}A$. Otherwise, the program text will not be derivable.

$$(E) \Rightarrow (E{+}A) \quad \text{or} \quad (E) \Rightarrow (E{-}A).$$

After each derivation, there are several ways to derive sentential forms with the program text as a prefix. The shortest way is as follows.

$$(E) \Rightarrow (E{+}A) \Rightarrow (E{+}3) \Rightarrow (A{+}3) \Rightarrow (2{+}3).$$

Here, we find a sentential form, $(E)$, whose prefix, ($E$, *derives* the program text, (2+3. We analyze this using Definition 1 as follows: Here, $\alpha$ is ($E$, $\gamma$ is ), the rest of the symbols after $\gamma$ are empty, and the production is $A \longrightarrow (E)$. This explains why ) is a simple candidate for a syntax completion of (2+3.

Let us derive another sentential form using $E \longrightarrow E + A$ from $(E{+}A)$ as follows:

$$(E{+}A) \Rightarrow (E{+}A{+}A) \Rightarrow \cdots \Rightarrow (2{+}3{+}A)$$

where the sentential form found is $(E{+}A)$, whose prefix, ($E$, derives the program text, (2+3. An analysis using Definition 1 reveals that $\alpha$ is ($E$, $\gamma$ is $+A$, the rest of the symbols after $\gamma$ are ), and the production is $E \longrightarrow E{+}A$. This gives us the second simple candidate, $+\cdots$, obtained by replacing a non-terminal $A$ with the ellipsis in $+A$.

The same argument, but using $E \longrightarrow E{-}A$, provides an account for the third simple candidate, $-\cdots$, in the following derivation.

$$(E) \Rightarrow (E{-}A) \Rightarrow (E{+}A{-}A) \Rightarrow \cdots (2{+}3{-}A)$$

Second, we argue that there are no more than these three simple candidates by showing that no new simple candidates can be obtained from any of the sentential forms that are yet to be derived. The remaining methods of derivation must start with one of the following:

$$E \Rightarrow A \Rightarrow (E) \Rightarrow (E{+}A) \Rightarrow (E{-}A{+}A)$$

and

$$E \Rightarrow A \Rightarrow (E) \Rightarrow (E{-}A) \Rightarrow (E{-}A{-}A).$$

The symbol - appears immediately after $E$ in any of two sentential forms, $(E{-}A{+}A)$ and $(E{-}A{-}A)$, but is not in the program text. Thus, for the program text to be a prefix of a sentential form, it must be derived by the non-terminal $E$. Therefore, any of the symbols right after $E - A$, that is, $+A$ and $-A$, must occur after $\gamma$. As a result, a problem of finding simple candidates with the two sentential forms becomes the same as that with $(E{-}A\cdots)$ because the part with the ellipsis is irrelevant to simple candidates. The problem can then be solved in the same way as the derivation $E \Rightarrow A \Rightarrow (E) \Rightarrow (E{-}A)$ used to compute the third simple candidate above.

Here let us note that $\gamma$ in Definition 2.2 may be an empty sequence, in which case the empty sequence is just not displayed to the users as a candidate. Allowing an empty sequence as a simple candidate is necessary when considering a nested candidate as a concatenation of simple candidates, as is shown in Section 4.

## 3 Algorithm for Computing Simple Candidates

In this section, we present an algorithm for computing simple candidates. The algorithm is based on LR parsers, and thus we assume the basic knowledge behind them.

As one basic idea behind the algorithm, the input of the algorithm is the stack of the LR parser when the parser reads terminal symbols for the program text up to the cursor, and the output is a set of simple candidates. The algorithm tries every sequence of the parsing actions after processing the program text up to the cursor, deriving all possible sentential forms to find simple candidates according to Definition 2.2.

The algorithm is as follows. Suppose the lexical analysis has succeeded for the input program up to the cursor and obtained the sequence of tokens. First, we parse the input sequence of tokens to the end. When successfully parsed, there are no candidates. When a parse error occurs, we start the following computation. Let $\alpha$ be the sequence of (terminal or non-terminal) symbols at this moment. We suppose that the stack of the LR parser alternately contains states and (terminal or non-terminal) symbols. In the algorithm, we "pick up" terminal or non-terminal symbols corresponding to "go to" and "shift", and picked-up symbols are made into a sequence when "reducing", which becomes a candidate.

(1) When there are reducing actions (possibly for different lookahead symbols and different productions) in the current state, one of them is attempted as follows.
`(* Return here later by backtracking and take some other reducing action. When all reducing actions are attempted, backtrack. *)`
Let $A \longrightarrow rhs$ be the production associated with the reducing action. We then apply the reducing action. Immediately before the reducing action, a postfix of the sequence of symbols on the stack coincides with $rhs$. Then, the postfix is replaced with the non-terminal symbol $A$. Let $\alpha'$ be the sequence of symbols on the stack immediately after the replacement.

(1-1) When $|\alpha| \geq |\alpha'|$, create a sequence of picked-up symbols, add it to the candidate list, and backtrack.

(1-2) Otherwise, backtrack.

(2) When there is no reducing action in the current state,

(2-1) When there are entries in the current state in the goto table, take one of them.
`(* Return here later by backtracking and take some other goto entry. When all goto entries have been tried, backtrack. *)`
Pick up the (non-terminal) symbol associated with the goto entry, "go to" the state according to the entry, and go to (1).

(2-2) When there is no entry in the current state in the goto table,

(2-2-1) When there is an accept entry in the current state, backtrack.

(2-2-2) When there are entries as shift actions (possibly for different lookahead symbols) in the current state, take one of them.
`(* Return here later by backtracking and take some other shifting action. When all shifting actions have been attempted, backtrack. *)`
Pick up the lookahead (terminal) symbol, "shift" according to the entry, and go back to (1).

Note that the condition $|\alpha| \geq |\alpha'|$ in (1-1) is equivalent to the condition $|rhs| > |\gamma|$ in Definition 2.2.

In the algorithm, the terminal and non-terminal symbols are picked up during shift and goto actions, respectively, and later used for creating a sequence of the picked-up symbols, which is $\gamma_i$. When backtracking, the sequence is reset. Otherwise, the sequence remains.

Here, let us make a remark regarding the algorithm.

**Remark 1.** Given a sequence of terminal or non-terminal symbols $\alpha$, the algorithm computes all simple candidates with respect to $\alpha$ according to Definition 2.2.

## 4 Nested Syntax Completion

In this section, we extend the simple candidates defined in Section 2 to *nested candidates*. To define the nested candidates, let us recall a notion of the rightmost derivation. We use the symbol $\underset{rm}{\Rightarrow}$ used in a textbook [1] for representing the rightmost derivation.

**Definition 4.1** (Rightmost derivation)**.** $\alpha \underset{rm}{\Rightarrow} \beta$ holds when the following conditions hold.

1. The rightmost non-terminal symbol in $\alpha$ is $A$.
2. $A \longrightarrow \gamma$ is a production, and
3. $\beta$ is a sequence obtained by replacing the rightmost non-terminal symbol $A$ with $\gamma$ in $\alpha$.

Using the notion of the rightmost derivation and simple candidates defined in Section 2, we define the nested candidates.

**Definition 4.2** (Nested candidates)**.** Let $\alpha_0$ be a prefix of a sentential form. For $i \geq 1$, let $\gamma_i$ be a simple candidate with respect to $\alpha_i$ if there is any and $\alpha_{i+1}$ be a sequence of (terminal or non-terminal) symbols that satisfies $\alpha_{i+1} \underset{rm}{\Rightarrow} \alpha_i\gamma_i$. A concatenation of $\gamma_0, \ldots, \gamma_j$ for some $j \geq 0$ with $\gamma_j$ being defined, is a *nested candidate* with respect to $\alpha_0$.

In general, there may be more than one simple candidate $\gamma_i$ for each $\alpha_i$, so the set of nested candidates constitute a tree structure. Definition of nested candidates in Definition 4.2 is illustrated in Fig. 3. Note that, in Definition 4.2, $\gamma_i$ may be an empty sequence, which will be illustrated in the example below.

Before describing an example, let us make a remark regarding $\alpha_i$ in Definition 4.2 for clarifying the definition of nested candidates.

**Remark 2.** In Definition 4.2, $\alpha_i$ ends in a non-terminal symbol for $i \geq 1$.

This remark is apparent from Definition 4.2 and illustrated in Fig. 3.

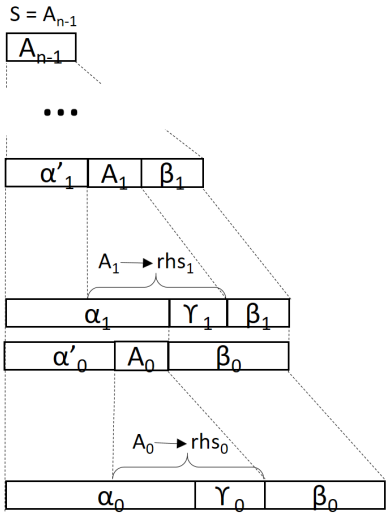


**Figure 3.** Definition of a set of nested candidates, $\{\gamma_0, \gamma_0\gamma_1, \ldots, \gamma_0 \cdots \gamma_{n-1}\}$

Now let us see the nested candidates for the example in Section 1.

**Example 4.3** (Nested candidates)**.** When the program text up to the cursor position is

`let val add = fn x =>`,

users are provided '. . .' and '. . . `in` . . . `end`' obtained by replacing all non-terminals in the nested candidates with ellipsis as follows.

*exp*, *exp* `in` *exp* `end`

Now, let us describe two nested candidates in the example according to the definition. To derive a sentential form in which the program text is a prefix, we must have the following derivation.

$$exp \Rightarrow appexp \Rightarrow atexp \Rightarrow \texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end}$$
$$\Rightarrow \texttt{let val}\ ID = exp\ \texttt{in}\ exp\ \texttt{end}$$
$$\Rightarrow \texttt{let val}\ ID = \texttt{fn}\ ID\ \texttt{=>}\ exp\ \texttt{in}\ exp\ \texttt{end}$$

The last sentential form is derived using a production

$$exp \longrightarrow \texttt{fn}\ ID\ \texttt{=>}\ exp$$

such that $\alpha_0$, $\gamma_0$, and $\beta_0$ are symbols as in the table:

| $\alpha_0$ | $\gamma_0$ | $\beta_0$ |
|---|---|---|
| `let val` $ID$ `= fn` $ID$ `=>` | $exp$ | `in` $exp$ `end` |

The second to last sentential form is derived using a production

$$dec \longrightarrow \texttt{val}\ ID = exp$$

where this sentential form is decomposed into $\alpha_1$, $\gamma_1$, and $\beta_1$ as in the table:

| $\alpha_1$ | $\gamma_1$ | $\beta_1$ |
|---|---|---|
| `let val` $ID$ `=` $exp$ | $\epsilon$ | `in` $exp$ `end` |

For the third to last sentential form, `let` *dec* `in` *exp* `end`, we use another production

$$atexp \longrightarrow \texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end}$$

with $\alpha_2$, $\gamma_2$, and $\beta_2$ as in the table:

| $\alpha_2$ | $\gamma_2$ | $\beta_2$ |
|---|---|---|
| `let` *dec* | `in` *exp* `end` | $\epsilon$ |

For the remaining sentential forms, $\alpha_3 = atexp$, $\alpha_4 = appexp$, $\alpha_5 = exp$, and $\gamma_3 = \gamma_4 = \beta_3 = \beta_4 = \epsilon$.

Following the definition, the nested candidates for the program text are $\gamma_0, \gamma_0\gamma_1, \ldots, \gamma_0\gamma_1\gamma_2\gamma_3\gamma_4$. By removing the redundancy, we obtain the following two nested candidates:

*exp*, *exp* `in` *exp* `end`.

These are displayed as '. . .' and '. . . `in` . . . `end`' to the user.

The above candidates are produced by the algorithm for computing nested candidates presented in Section 4.1. Note that *exp* `in` *exp* is not a nested candidate because

`let val add = fn x =>` *exp* `in` *exp*

does not have any postfix, which is the right-hand side of a production.

In Example 4.3, there is a finite number of nested candidates, whereas in some other cases, there are infinitely many nested candidates constituting a graph structure. Consider the example under the grammar for an arithmetic language in Section 2:

$$E := A \mid E \texttt{ + } A \mid E \texttt{ - } A$$
$$A := \texttt{num} \mid \texttt{(}E\texttt{)}.$$

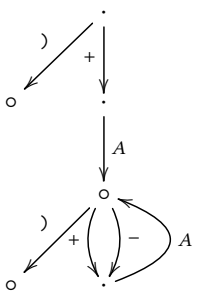


**Figure 4.** Graph structure of the nested candidates

**Example 4.4** (Nested Candidates)**.** When the program text up to the cursor position is '`(2+3`', there are infinitely many nested candidates, which is depicted in a graph in Fig. 4. All nested candidates for a sequence of terminal and non-terminal symbols actually constitute a tree structure, and a graph can efficiently represent this by sharing some subtrees. In the graph, each circle corresponds to a nested candidate.

Note that the length of each nested candidate is finite according to Definition 4.2, although there are infinitely many

nested candidates for '(2+3'. Thus, only finite number of candidates should be taken from the infinitely many candidates represented by the graph in Fig. 4.

In the implementation, we employ a heuristic to compute only a finite subset of the nested candidates per each user request, as described in Section 5. We may consider nested candidates in terms of graph structures, instead of tree structures, in the future.

We also comment on the relationship between simple and nested candidates.

**Remark 3.** Suppose $\alpha$ is a sequence of terminal or nonterminal symbols. The set of simple candidates with respect to $\alpha$ is a subset of the set of nested candidates with respect to $\alpha$.

### 4.1 Algorithm for Computing Nested Candidates

In this section, we present an algorithm for computing nested candidates. The algorithm is the same as that for computing simple candidates except for case (1-1), which is shown here.

(1-1) When $|\alpha| \geq |\alpha'|$, create a sequence of picked-up symbols, add it to the candidate list, and go back to (1).

Note that the algorithm may not terminate when there are infinitely many nested candidates. Let us make an assumption here.

**Conjecture 4.5.** Given a sequence of terminal or nonterminal symbols $\alpha_0$, the algorithm computes all nested candidates with respect to $\alpha_0$ according to Definition 4.2. More precisely, $\gamma$ is a nested candidate according to Definition 4.2 if and only if $\gamma$ is produced as a candidate during the finite steps applied by the algorithm.

For the conjecture to hold, the algorithm should be rewritten in a breadth-first manner, although the algorithm is described in a depth-first manner for better readability.

In the implemented system, we compute a finite subset of nested candidates, as shown in Section 5.

## 5 Implementation

This section describes an implementation of the two algorithms for computing the syntax completion candidates. Every user views our system as an editor to request a syntax completion and a server to respond with candidates. Note that we chose Emacs as the editor for evaluation of the implementation. We implemented the algorithms on a server that receives a request with a program text from an editor and responds to the editor with a list of strings for a syntactic completion candidate of the program text. Emacs displays the computed lists as indicated in Fig. 5 for the simple mode and as shown in Fig. 6 for the nested mode. When there is only one candidate, it is inserted at the cursor position directly without a selection, as shown in Fig. 5, where the ellipsis is the candidate.


**Figure 5.** A screen shot of syntax-completion mode (simple candidate)

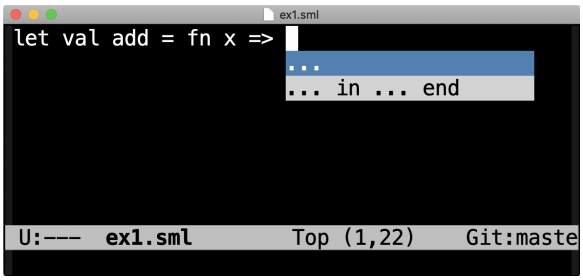


**Figure 6.** A screen shot of syntax-completion-nested mode (nested candidate)

The server can also respond to the editor with either `SuccessfullyParsed` or `LexicalError`. The former response indicates that the program text is successfully parsed and thus there is nothing left to complete. The latter suggests that programmers should correct unacceptable lexical symbols in the program text before they ask for syntax completion.

The source code of our implementation and two example uses are available at https://github.com/kwanghoon/{yapb,arith,smllike}.

Internally, the algorithms have been implemented as an extension of the LALR(1) parser, which is one of the most popular parsing methods. After the server receives a request from the editor, the LALR(1) parser is applied to a program text in the request. When the program text is *complete*, it is successfully parsed by the parser. When the program text contains incorrect lexical symbols, it is stuck in the lexical analysis. Otherwise, the complete text is parsed but will still not be in the final accepted state. The automaton stack is the reverse of the prefix of a sentential form interspersed with automaton states. At this time, the syntax completion algorithms is applied.

The input of the algorithm implementation is the state of the LALR(1) parser (parser stack), and the parsing automaton (action table, goto table, and production rules). Given the

input, the implementation of one of the simple and nested algorithms begins, depending on the simple and nested modes in the editor set by programmers.

### 5.1 A Running Example

As a running example of the implementation, let us now see how the simple syntax candidates for the arithmetic program text shown in Section 2 are computed.

In the following, we describe how the LALR(1) automaton in our implementation parses the text up to the cursor position represented by ∧. For notation, each step has a dot to indicate the point of parsing.

```
      . ( 2 + 3 ∧    by shift over (
⟶  ( . 2 + 3 ∧    by shift over num
⟶  ( 2 . + 3 ∧    by reduce A → num
⟶  ( A . + 3 ∧    by reduce E → A
⟶  ( E . + 3 ∧    by shift over (
⟶  ( E + . 3 ∧    by shift over num
⟶  ( E + 3 . ∧    by reduce A → num
⟶  ( E + A . ∧    by reduce E → E + A
⟶  ( E . ∧
```

Immediately after the last step above, the candidate computing algorithm starts working with the automaton stack that contains `E` and `(`, a reverse of the symbols in the step, interspersed by the automaton states. This state can be described as the following set of LR(1) items, each element of which is a pair of a production with a dot at a certain position on the right side and a lookahead:

```
[A → ( E.), end]
[A → ( E.), +]
[A → ( E.), -]
[A → ( E.), )]
[E → E.+ A, )]
[E → E.+ A, +]
[E → E.+ A, -]
[E → E.- A, )]
[E → E.- A, +]
[E → E.- A, -]
```

Recall that simple candidates are defined as a sequence of symbols up to the point of the first reduce action immediately after parsing a given program text. The set of LR(1) items as the current state shows three possible paths to the point of a reduce action.

- The first four LR(1) items indicate '`)`' as a simple candidate by reducing a production $A \rightarrow (E)$.
- The next three LR(1) items represent '`+ ...`' as the second simple candidate obtained from replacing a non-terminal symbol `A` with the ellipsis in '`+ A`' by reducing a production $A \rightarrow E$+$A$.
- For the remaining three LR(1) items, the same argument produces '`- ...`' as the third simple candidate.

The implementation of the algorithm actually utilizes an action table and a goto table in the LALR(1) parser. As shown in the example, the algorithm collects all simple (and nested) candidates using backtracking that searches for all next states that are reachable from a given state through reduces/shifts in the action table and gotos in the goto table.

### 5.2 A Heuristic

A naive implementation of the algorithm for syntax completion candidates may not be terminating whenever the number of candidates is infinite. In practice, there must be a way to work around this problem for a user to avoid being blocked by computing infinite number of candidates and to be guaranteed to receive only a finite number of candidates on each request.

We design a heuristic to avoid generating an infinite number of candidates. The heuristic is simple: the algorithm is extended to maintain a history of the backtracking. Using the history, the implementation of the extended algorithms can detect a cycle while trying to search as many candidates as possible.

The data structure for the history is a list of triples, namely, a state, a parsing stack, and a feature of a parsing operation applied at the state using the stack. The feature is a triple of a production for a reduce action, a non-terminal for a goto table, and a terminal for a shift action. It refines a pair of a state and parsing stack with the features such that the backtracking can search over all different goto/shift symbols in the same state. In the example state, without the inclusion of such a feature, only the first four LR(1) items are considered for the search and the remaining items are not searched.

Although this data structure is sufficient to stop our implementation from entering into an infinite loop, the heuristic occasionally generates an unbalanced set of candidates. We discuss this issue later.

As an alternative heuristic to what is implemented, the breadth-first search (BFS) could be used . A BFS-based algorithm would compute nested candidates within a limit of levels that programmers set for search. Such a level number could be provided by a user or by some static analysis of the parser. Also, different levels might be specified for different syntactic constructions for better completions.

### 5.3 YAPB: A Tool for Building a Text-Based Syntax Completion System from a Parser for Free

We developed a tool that provides a text-based syntax completion system from a given parser for free. This tool is known as *Yet Another Parser Builder (YAPB)*. Basically, it is a programmable parser builder system in which every LALR(1) parser (specification) is written in Haskell and is immediately executable. Hence we call it a parser builder rather than a parser generator that generates an executable parser. Once a parser is written in YAPB, a syntax completion system using the proposed algorithms is available without incurring any cost. Figure 7 shows a list of (abbreviated) Haskell type

```
{- For parser -}
lexing :: LexerSpec token-> String
                       ->IO[Terminal token]
parsing :: ParserSpec token ast
       -> [Terminal token]->IO ast

{- For syntax completion -}
successfullyParsed :: IO[EmacsDataItem]
handleLexError :: IO[EmacsDataItem]
handleParseError :: Bool ->
     ParseError token ast -> IO[EmacsDataItem]
```

**Figure 7.** List of (abbreviated) type declarations of common library functions in YAPB for parsing and syntax completion

declarations for a few important library functions provided by YAPB.

Using the YAPB library, as in the following, a typical Main module can be written in Haskell for a syntax completion server. Given the parser and lexer specifications (`parserSpec` and `lexerSpec`), YAPB library functions, `lexing` and `parsing`, are used to conduct a lexical analysis and a syntax analysis over a program text `prog`. This acts exactly as a parser for constructing an abstract syntax tree, `ast`, from a string using a list of terminals (or tokens). As a syntax completion server, when it successfully parses the program text, another library function `successfullyParsed` is used to return an empty list of candidates to the Emacs editor.

```
main :: IO ()
main = emacsServer compCand

compCand::String->Bool->IO[EmacsDataItem]
compCand prog mode = ((do
   terminals <- lexing lexerSpec prog
   ast <- parsing parserSpec terminals
   successfullyParsed)`catch`\e ->
    case e::LexError of
      _ -> handleLexError) `catch`\e ->
     case e::ParseError Token AST of
        _ -> handleParseError mode e
```

The remaining parts are more interesting. When parsing an incomplete program text, a parser throws either a lexical error exception or a parser error exception. Every lexical error is handled by the YAPB library function `handleLexError`.

Every parse error occurs for a partially complete program text. The YAPB library function `handleParseError` that ultimately implements our algorithms handles the parse error under a specified mode of simple and nested candidates. The exception `e` at that moment contains all information necessary for the algorithm to compute the syntax completion candidates, and is composed of dynamic information, that is, the current state of the LALR(1) parser and the parsing stack, as well as static information, that is, the action table, goto table, and production rules.

Our tool is advantageous in that developers can reuse a parser specification with no change for building a syntax completion server, as shown in the Haskell example program above.

## 6 Discussions

In this section, we discuss several aspects such as analyzing the complexity of the algorithms.

### 6.1 Complexity of the Algorithms

Here, we analyze the complexity of the algorithms. First, we consider the algorithm for computing simple candidates. Note that there are only a finite number of simple candidates because of the length condition in Definition 2.2.

Simple candidates with respect to a sequence of symbols $\alpha$ constitute a tree structure because we add symbols using shifts and gotos until reaching the state in which a reduce is possible, and thus there is no cycle in the structure. A path from the root to each leaf in the tree is a postfix of the right-hand side of a production in the grammar. A path from the root to an internal node corresponds to the same prefix of some postfixes.

Thus, the number of leaves in the tree is at most the number of the prefixes on the right-hand side of the productions in the grammar. The depth of the tree is at most *maxrhs*, which is the maximum among the lengths of the right-hand sides of the productions in the grammar. Let $n$ be the number of productions in the grammar. Then $p$ is at most $n \cdot maxrhs$. The number of nodes in the tree is at most

$$p \cdot maxrhs = n \cdot maxrhs^2.$$

The time complexity of the algorithm for computing simple candidates is $O(n \cdot maxrhs^2)$. In practice, it is expected that the number of nodes in the tree will be much less than $n \cdot maxhrs^2$.

The algorithm for computing the nested candidates may produce infinitely many nested candidates, and thus the implementation uses certain heuristics. Various heuristics should be applicable, with the complexity analysis left for future study.

### 6.2 Comparison with the Approach Based on Grammar Transformation

In [29], a given grammar for a (full) language is transformed into a grammar for the prefixes, the program text is parsed up to the cursor position to produce a partial parse tree, and the candidates are computed by traversing this tree, although the transformation of the grammar is done manually. The author specifies the syntactic candidates to be completed based on a grammar similarly to the present work. The specifications in [29] correspond to the simple candidates without the length condition. Using the simple candidates, we specify the nested candidates in the present study. In [29], there is too much

freedom in selecting a subset from the set of candidates conforming to the specification, whereas we specified the candidates more precisely.

There should be a certain relation between the approach using the grammar transformation and the approach by using the internal information of the LR parser described in the present study. The algorithm in [29] refers to the production for obtaining the sequence of symbols, constituting a part of a candidate, when visiting each node in the partial parse tree, which corresponds to a simple candidate obtained by a sequence of actions consisting of shifts and gotos followed by a reduce. We leave the exact comparison as a future study.

### 6.3 Partial Parsers by Derivatives

We may be able to directly write parsers for a program text up to the cursor based on derivatives, not through a grammar transformation described in Section 6.2. A study [19] on writing partial parsers using a functional parser combinator was conducted. This study extends research [4] on the derivatives of regular expressions by utilizing a lazy evaluation and memoization.

Suppose that $L$ is the language and $c$ is a character (or a token). In addition, $D_c(L)$ is defined as the derivative of the language $L$ with respect to $c$.

$$D_c(L) = \{w \mid cw \in L\}$$

In words, $D_c(L)$ is the set of all cdr values of the strings, each of which has $c$ as its first character. Suppose a language $L$ is specified by a partial parser $p$ that is constructed by the parser combinators. The partial parser, which takes the remaining string after the cursor position, can then be obtained by recursively computing the derivative of the parser $p$ with respect to each character in the input up to the cursor position, individually. We might be able to use the obtained partial parser for computing candidates at the cursor position. If implemented, the computational complexity is expected to be much larger compared with our approach because simply parsing a string $s$ requires a time exponential to the length of the string $s$ by computing the derivatives of the parsers [19].

### 6.4 A Role of Ellipsis . . . on the Editor

In the current implementation, non-terminal symbols are represented as ellipsis . . . , which can be anything that is not a legal lexeme in the language. Although at present the ellipsis are not used for any specific purpose, one possible role of the ellipsis . . . in the Emacs editor is that, when a user presses a key, such as the enter key, over the ellipsis, some candidates that replace the ellipsis may pop up.

A sequence of symbols shown as a nested candidate, possibly containing ellipsis, may represent an internal state, and not a leaf, of the breadth-first search algorithm for computing the nested candidates presented in Section 4.1, because the search is cut short by some of the heuristics in the implementation, as described in Section 5. A special lexeme, such as a placerholder in the generic framework for syntactic code completion [2], may be used for representing the truncated part of a candidate. In this case a possible role of the special lexeme is that, when the user presses a key, such as the enter key, over the special lexeme, the remaining portion of the candidates that was truncated during the previous completion pops up by communicating with the server. Under this situation, it may be stated that the candidates are computed gradually according to the user's operation on the editor like structured editing.

### 6.5 Limitations

This section discusses a few limitations in our approach to the syntax completion problem. Firstly, this study used only the program text up to the cursor position; however, programmers may not necessarily write the code sequentially from left to right. In the future, we may consider syntax completion based on the program text before and after the cursor position.

Secondly, this study did not consider the completion of identifiers, which is one of important features in editors. It requires a semantic analysis over the whole program, such as type checking, rather than the LR-based syntactic analysis over a prefix of the program. To support the completion of identifiers, static semantics of the target language should be considered, which would be challenging [21]. For implicitly-typed languages with static scope, we could combine the algorithms in the present work with an algorithm, proposed by Sasano and Goto [30], for completing identifiers when the program text up to the cursor position is consistent with respect to the syntax and types.

Thirdly, the result of the syntax completion in this study might highly depend on the shape of the grammar. LR grammars must sometimes be expressed in non-intuitive ways to fit into the LR constraints. This could lead to completions that do not have any particular semantic meaning. We need to investigate how the shape of the grammars affects completion candidates in our approach.

## 7 Related Work

Several types of methodologies and tools have been presented for syntax, expression, and identifier completion, and we describe some of them in this section. Some aspects of parsing or grammar concern syntax completion, so we also argue here some studies about program generation, word (or string) generation, incremental parsing, and grammar transformation.

A recent study [2] investigated a syntax completion that has been referred to by the authors as *syntactic code completion*. They represent an incomplete program text using placeholders. On their IDE, users select a placeholder for

various language constructs, e.g., expressions, and syntactic candidates pop up. The IDE inserts a syntactic component by selecting one of them. In their approach, an incomplete program text is (syntactically) complete except for the placeholders. They also presented *placeholder inference*, which infers placeholders corresponding to missing part in the source code being input. Further, they argued soundness and completeness of their syntax completion. By contrast, we do not use placeholders in the same sense and allow an arbitrary program text to be present after the cursor position. We leave as a future study to argue soundness or completeness of our algorithm.

Further, in [32], the authors investigated *robust* editing using projectional editors, *i.e.*, editing a well-formed program results in a well-formed program. Two DSLs are used by the authors. One is for specifying whether it is appropriately formed, whereas the other is used to specify the robust editing patterns. By contrast, we allow a syntactically incomplete program text, as well as syntax completion resulting in such a syntactically incomplete program text.

Kuiper *et al.* [14, 27] implemented a system called LRC, which generates graphical language-oriented tools; however, this LRC system is no longer available. The LRC system takes a higher order attribute grammar (HAG) [34] that specifies a language as its input and generates purely functional and incremental attribute evaluators. For example, the LRC system can generate editors. Although the editors generated by the LRC system allow users to edit the source code either through a GUI or on a character basis, editors exhibiting a syntax completion functionality have not been generated using the LRC.

In 2012, Perelman *et al.* [22] proposed an algorithm to complete expressions, but not for completing the syntax in general, in an object-oriented language. This approach requires programmers to write an expression with holes, which is referred to as a *partial expression*, and subsequently computes several candidates to fill in the holes with some ordering. When computing expressions to fill in the holes, types are taken into account so that the completed expression type-checks. The Agda language has provided typed holes since the 1990s; more recently, GHC, a Haskell compiler, has enabled programmers to use typed holes allowing them to obtain information that may help in properly filling the holes. By contrast, our system does not require programmers to write such expressions using holes, although types are not considered in the present study.

Several studies have investigated identifier completion. In 2013, Sasano and Goto [30] proposed an algorithm to complete an identifier when the program text up to the cursor position is consistent with respect to the syntax and types. Later, Sasano [28] presented an approach to complete the identifiers by coping with an incomplete program text based on LR parsing error recovery. In this study, the error recovery behavior is manually specified using the Yacc specification. In addition, de Jonge *et al.* [5] developed a method for deriving error recovery rules based on the grammar specification. In the future, these studies may be utilized for syntax completion in the case of an incomplete program text that may not necessarily be syntactically complete up to the cursor position. Rittri [24] developed a method for searching for an identifier in a program library using types as search keys. Runciman *et al.* [26] independently developed a method for the same problem by considering the types that can be unified.

Gvero *et al.* [9] presented an approach for code completion using languages with a parametrically polymorphic type system. They developed a solution with respect to the type inhabitant problem: For a given type environment $\Gamma$ and a type $\tau$, an expression $e$ is searched such that the type judgment $\Gamma \triangleright e : \tau$ holds. They compute $\Gamma$ based on the cursor position and $\tau$ by examining the declared types appearing up to the cursor position. Our system does not use the type information. Although the authors use the types, their approach does not cope with the partial terms.

Robbes *et al.* [25] claim that finding a candidate in the popup window is cumbersome or even slower than typing the full name when using a completion engine, such as content assist in Eclipse. They limited the number of the candidates and developed an algorithm to compute the candidates based on the program editing history. By contrast, our system does not limit the number of candidates so there may be many candidates when the partial syntax tree is deep, which should be coped with in some way in the future.

Several studies have investigated generating a portion of the programs: Hashimoto [10] constructed an ML-style programming language with first-class contexts, *i.e.*, expressions with holes. Our study also uses holes in a certain sense; however, the language used a construct for filling in the holes. Other tools have been developed to generate terms under type constraints. Djinn[1] generates a term having a given type. The tool proposed by Katayama [12] generates a minimum term having a given type. Jeuring *et al.* [11] published a technique for generating generic functions, and a previous study [13] investigated the synthesis of functions matching a set of input-output pairs. These studies are not directly related to our problem, however.

Related to program generation, some studies such as [17] investigated string (or word) generation from context-free grammars, the motivation of which is for testing parsers. The study [17] by McKenzie investigates a problem of generating strings of a given length $n$ at random, with each string of length $n$ being generated in the same probability, derivable from a given context-free grammar $G$. Another study [7] about string generation investigated a problem of generating at random strings of some context-free language

[1] http://www.augustsson.net/Darcs/Djinn/

$L$ with respect to a given distribution of the number of occurrences of the alphabets of the language $L$. String generation at random with some requirements may be a possible way to be used in generating syntax candidates when reducing the number of syntax candidates begin generated.

Incremental parsing was investigated in many studies, such as the study [35] for avoiding reparsing of the whole program after each modification and, as an application, the programming tool Merlin [3], as is mentioned in Section 1, for OCaml uses incremental parsing. In source code editors supporting syntax completion, duplicated computation naturally occurs in parsing. In usual situations, programmers edit source code in a few files at once, each of which has several hundreds of lines or so. LR parsing, fully to the end or partially up to the cursor, takes time linear to the length of the input string and actually fast, compared with the computation of syntax candidates. Considering this situation, we did not pay attention to the incremental parsing, for the purpose of avoiding reparsing, at this moment.

Several studies have investigated a grammar transformation. A grammar transformation has been investigated with respect to various systems, e.g., the TXL system [6], which the authors refer to as *grammar programming*. TXL supports a structural program transformation. A grammar transformation concerns the syntax completion when candidates are computed simply by using the program text up to the cursor position. In this case, we may obtain the grammar for partial programs up to the cursor by transforming the grammar of the full language. One way [29] was presented to generate a completion system through a grammar transformation, which was discussed in Section 6.

## 8 Conclusions and Future Work

In this study, we presented a novel approach to generating a tool for syntax completion. With this approach, we utilize an LR parser to parse the program text up to the cursor position and then use the internal information of the LR parser to compute the candidates. Syntax completion functionality is generated from a description of the grammar, which is expected to reduce the cost of implementing one for each language. We believe that the language need not be restricted when using our approach as long as the grammar of the language is LALR(1).

Several additional investigations must be conducted in the future, some of which are listed as follows.

- We plan to give a proof that the algorithms for computing the candidates conform to the candidate specifications.
- In this study, we do not complete the identifiers. Identifier completion concerns the scope rule and the type system of the language, and thus we need to consider how to incorporate the identifier completion into the tool presented in this paper.
- Using attribute grammar, we may be able to uniformly achieve syntax completion and identifier completion. Additionally, we may consider the use of parsers apart from LR, such as the ANTLR [20] and GLR [15, 18, 31, 33] parsers.
- We have not compared the approach by using the internal information of the LR parser presented in the present study with the approach through a grammar transformation [29]. Although we gave a rough comparison in Section 6, we leave the exact comparison as a future study.
- We plan to argue the time complexity for computing nested candidates, which depends on the heuristics.
- Although the current implementation employed a cycle detection heuristic to generate only a finite number of candidates, it has not been evaluated thoroughly regarding whether the chosen candidates are useful. We leave this as a future study.
- We would like to do some evaluation to show more practical results, such as how many possible completions are computed for typical languages, how usable and relevant candidates are produced in practice, and how scalable the proposed method is in terms of performance as well.

## Acknowledgments

We would like to thank the anonymous reviewers of PEPM 2021 for their many helpful comments. This work was partially supported by JSPS KAKENHI under Grant Number 20K11752 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. 2007. *Compilers — principles, techniques, and tools, 2nd edition.* Addison Wesley.

[2] Luís Eduardo de Souza Amorim, Sebastian Erdweg, Guido Wachsmuth, and Eelco Visser. 2016. Principled syntactic code completion using placeholders. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering (SLE 2016).* ACM, Amsterdam, Netherlands, 163–175. https://doi.org/10.1145/2997364.2997374

[3] Frédéric Bour, Thomas Refis, and Gabriel Scherer. 2018. Merlin: A language server for OCaml (Experience report). *Proc. ACM Program. Lang.* 2, ICFP, Article 103 (July 2018), 15 pages. https://doi.org/10.1145/3236798

[4] Janusz A. Brzozowski. 1964. Derivatives of regular expressions. *J. ACM* 11, 4 (1964), 481–494. https://doi.org/10.1145/321239.321249

[5] Maartje de Jonge, Lennart C. L. Kats, Eelco Visser, and Emma Söderberg. 2012. Natural and flexible error recovery for generated modular language environments. *ACM Trans. Program. Lang. Syst.* 34, 4, Article 15 (2012), 15:1–15:50 pages. https://doi.org/10.1145/2400676.2400678

[6] Thomas R. Dean, James R. Cordy, Andrew J. Malton, and Kevin A. Schneider. 2002. Grammar programming in TXL. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM '02).* IEEE Computer Society, 93. https://doi.org/10.1109/SCAM.2002.1134109

[7] Alain Denise, Olivier Roques, and Michel Termier. 2000. Random generation of words of context-free languages according to the frequencies of letters. In *Mathematics and Computer Science*, Danièle Gardy and Abdelkader Mokkadem (Eds.). Birkhäuser Basel, Basel, 113–125. https://doi.org/10.1007/978-3-0348-8405-1_10

[8] Veronique Donzeau-Gouge, Gerard Huet, Bernard Lang, and Gilles Kahn. 1980. *Programming environments based on structured editors: the MENTOR experience.* Technical Report RR-0026. INRIA.

[9] Tihomir Gvero, Viktor Kuncak, and Ruzica Piskac. 2011. *Code completion using quantitative type inhabitation.* Technical Report EPFL-REPORT-170040. Ecole Polytechnique Federale de Lausanne.

[10] Masatomo Hashimoto. 1998. First-class contexts in ML. In *Asian Computing Science Conference (Lecture Notes in Computer Science, Vol. 1538)*. Springer, 206–223. https://doi.org/10.1007/3-540-49366-2_16

[11] Johan Jeuring, Alexey Rodriguez, and Gideon Smeding. 2006. Generating generic functions. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Generic Programming (WGP '06)*. Portland, Oregon, USA, 23–32. https://doi.org/10.1145/1159861.1159865

[12] Susumu Katayama. 2005. Systematic search for lambda expressions. In *Trends in Functional Programming*. 111–126.

[13] Pieter W. M. Koopman and Rinus Plasmeijer. 2006. Systematic synthesis of functions. In *Trends in Functional Programming (Trends in Functional Programming, Vol. 7)*, Henrik Nilsson (Ed.). Intellect, 35–54.

[14] Matthijs F. Kuiper and João Saraiva. 1998. LRC - A generator for incremental language-oriented tools. In *Proceedings of the 7th International Conference on Compiler Construction (CC '98)*. Springer-Verlag, London, UK, 298–301. https://doi.org/10.1007/BFb0026440

[15] Bernard Lang. 1974. Deterministic techniques for efficient non-deterministic parsers. In *Proceedings of the 2nd International Colloquium on Automata, Languages and Programming (ICALP '74) (Lecture Notes in Computer Science, Vol. 14)*. Springer-Verlag, 255–269. https://doi.org/10.1007/978-3-662-21545-6_18

[16] 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).

[17] Bruce McKenzie. 1997. *Generating strings at random from a context free grammar.* Technical Report TR-COSC 10/97. University of Canterbury.

[18] Scott McPeak and George C. Necula. 2004. Elkhound: A fast, practical GLR parser generator. In *Compiler Construction*, Evelyn Duesterwald (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 73–88. https://doi.org/10.1007/978-3-540-24723-4_6

[19] Matthew Might, David Darais, and Daniel Spiewak. 2011. Parsing with derivatives: A functional pearl. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming* (Tokyo, Japan) *(ICFP '11)*. Association for Computing Machinery, New York, NY, USA, 189–195. https://doi.org/10.1145/2034773.2034801

[20] 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

[21] Daniel A. A. Pelsmaeker, Hendrik van Antwerpen, and Eelco Visser. 2019. Towards language-parametric semantic editor services based on declarative type system specifications (Brave new idea paper). In *33rd European Conference on Object-Oriented Programming (ECOOP 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 134)*, Alastair F. Donaldson (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 26:1–26:18. https://doi.org/10.1145/3359061.3362782

[22] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. 2012. Type-directed completion of partial expressions. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI '12)*. ACM Press, Beijing, China, 275–286. https://doi.org/10.1145/2254064.2254098

[23] Thomas Reps and Tim Teitelbaum. 1984. The Synthesizer Generator. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*. 42–48. https://doi.org/10.1145/800020.808247

[24] Mikael Rittri. 1989. Using types as search keys in function libraries. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture (FPCA '89)*. 174–183. https://doi.org/10.1145/99370.99384

[25] 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

[26] Colin Runciman and Ian Toyn. 1989. Retrieving re-usable software components by polymorphic type. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture (FPCA '89)*. ACM Press, Imperial College, London, United Kingdom, 166–173. https://doi.org/10.1145/99370.99383

[27] João Saraiva. 2002. Component-based programming for higher-order attribute grammars. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE '02)*. Springer-Verlag, London, UK, 268–282. https://doi.org/10.1007/3-540-45821-2_17

[28] 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

[29] 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

[30] 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

[31] Maarten P. Sijm. 2019. Incremental scannerless generalized LR parsing. In *Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity* (Athens, Greece) *(SPLASH Companion 2019)*. Association for Computing Machinery, New York, NY, USA, 54–56. https://doi.org/10.1145/3359061.3361085

[32] Friedrich Steimann, Marcus Frenkel, and Markus Voelter. 2017. Robust projectional editing. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2017)*. ACM, Vancouver, BC, Canada, 79–90. https://doi.org/10.1145/3136014.3136034

[33] 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

[34] Harald Vogt, Doaitse Swierstra, and Matthijs Kuiper. 1989. Higher order attribute grammars. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation* (Portland, Oregon, USA) *(PLDI '89)*. ACM, 131–145. https://doi.org/10.1145/73141.74830

[35] Tim A. Wagner and Susan L. Graham. 1998. Efficient and flexible incremental parsing. *ACM Transactions on Programming Languages and Systems* 20, 5 (1998), 980–1013. https://doi.org/10.1145/293677.293678