An Approach to Generate Text-Based IDEs for Syntax Completion Based on Syntax Specification

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

Abstract

The integrated development environments provide several types of functionalities. Herein, we intend to generate a syntax completion functionality from the grammar of the target language as long as the sentences of the language can be analyzed via LR parsing. We specify the syntax candidates to be completed based on the sentential forms and reductions in LR parsing. Furthermore, we implement a prototype system for computing the syntax candidates to be completed at the cursor position in the source code written in a small subset of Standard ML; the system only uses the program text up to the cursor position to ensure simplicity.

CCS Concepts • **Software and its engineering → Syntax**; **Integrated and visual development environments**; *General programming languages.*

Keywords syntax completion, integrated development environments, LR parsing, sentential forms, reduction

ACM Reference Format:
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 (PEPM '20), January 20, 2020, New Orleans, LA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3372884.3373158

1 Introduction

The functionalities of integrated development environments (IDEs), including identifier or syntax completion, are used for the incomplete program text being edited. Further, the IDE developers may have their own policies to deal with incomplete program text and may not specify or explicitly describe the functionalities. However, programmers may hesitate to use IDEs if the specification of their functionalities is unclear. Proficient programmers desire to know the functionalities so that they can precisely predict their behavior.

Furthermore, the so-called structured editors or projectional editors, such as Synthesizer Generator [16] and MENTOR [4], have been developed. The program source code is represented as some tree structure in structured editors. They maintain a tree structure by restructuring it when the program is modified. They exhibit some good properties, especially that the behaviors of the editors can be formally specified and that their features, such as syntax completion, can be easily implemented. An editor is considered to be *text-based* when it does not prevent the programmers from editing the program text on a character basis. When compared with the text-based editors, majority of the structured editors prevent programmers from freely editing the program text. Because of the this inconvenience, most people do not like to use structured editors for editing the program source code even though they may be suitable for editing structured data apart from the program source code.

Therefore, we develop text-based editors that can provide formally specified functionalities, especially syntax completion. Further, we use LR parsers to achieve formally specified, yet practical, syntax completion based on the grammar.

In this study, we present an approach to achieve syntax completion based on the following features:

- syntax completion is formally specified based on LR parsing;
- syntax completion is implemented by reusing the code in compilers, especially the syntax description in LR parsing, as much as possible;
- the syntax completion behaviors can be predicted by programmers having some knowledge of LR parsing if they would like to understand the precise syntax completion behaviors; and
- syntax completion is *text-based* in the aforementioned manner.

Although the initial three features may be observed in many structured editors, the fourth feature will not be observed, except in the editors generated by the LRC system

PEPM '20, January 20, 2020, New Orleans, LA, USA

ACM ISBN 978-1-4503-7096-7/20/01…$15.00
https://doi.org/10.1145/3372884.3373158

[11, 20]. To the best of our knowledge, none of the developed editors possess all the four aforementioned features.

To ensure simplicity, we only use the program text up to the cursor position for syntax completion, *i.e.*, ignore the text after the cursor position if any; further, we assume that there are no syntax errors up to the cursor position. Throughout the paper, we use the LR parsing terminologies in a previous study conducted by Donald Knuth [9].

2 Syntax Completion Specification

In this section, we present the problem that we intend to solve in this study. We use the following language as an instance on which the problem is specified. It is a functional language, but it is not essential for our approach, which can be applied to any language as far as its syntax description suits our approach.

$$\begin{array}{rcl} M & := & \texttt{ID} \mid \texttt{fn ID => } M \mid M\ M \mid (M) \\ & \mid & \texttt{let val ID = } M \texttt{ in } M \texttt{ end} \end{array}$$

Here, the italic M represents a nonterminal symbol, whereas the remaining symbols directly represent the terminal symbols or lexemes. `ID` represents an identifier, `fn ID =>` M represents a function abstraction, $M\ M$ represents a function application, and `let val ID =` M `in` M `end` represents a let expression. We have included the parenthesized expression (M) in the syntax to express situations in which open parentheses that are not yet closed are present.

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 α be the sequence of tokens (terminal symbols) up to the cursor position. When $\alpha\beta$ is a sentential form, the sequence β is referred to as a *postfix sentential form* with respect to α.

Usually, the candidates that are expected to be completed by the users are those which *correspond to* or *close* some of the syntactic components in the program text up to the cursor position. Based on this observation, the candidates to be completed can be formally specified as follows.

Definition 2.2 (Candidates to be completed)**.** Let α be the sequence of tokens (terminal symbols) up to the cursor position. A sequence of symbols γ is a candidate to be completed when $\alpha\gamma$ is a sequence of symbols with a postfix that can be reduced by some production in the grammar.

Based on the aforementioned definitions, the following remark holds.

Remark 1. *Let α be the sequence of tokens (terminal symbols) up to the cursor position. A candidate to be completed at the cursor position is some prefix of a postfix sentential form w.r.t α.*

Based on the aforementioned definitions, in general, there are infinitely many candidates to be completed for a program text that is syntactically valid up to the cursor position. Therefore, a suitable finite subset should be selected from the set of all the candidates. Here, we provide some examples to denote the manner in which we select candidates.

When the program text up to the cursor position is

`let val add = fn x =>`,

the following three candidates can be selected from infinitely many candidates:

`ID`, M, M `in` M `end`.

The completion results are given as follows:

`let val add = fn x => ID`,
`let val add = fn x =>` M,
`let val add = fn x =>` M `in` M `end`.

When the program text up to the cursor position is

`let val add = fn y`,

we select the following two candidates:

`=>` M, `=>` M `in` M `end`.

The completion results are given as follows:

`let val add = fn y =>` M,
`let val add = fn y =>` M `in` M `end`

Basically, the candidates are accumulated according to the nesting structures of the program text up to the cursor position that have not yet been closed. We make an exceptional treatment for `ID` in the *use* positions, because many candidates may be present for `ID`, which will be discussed in Section 4. For `ID` in the *declaration* positions, which is `ID` immediately after `fn` or `val` in the definition of M, we do not compute the candidates, since it is nonsense. Following sections detail the manner in which candidates are computed.

For simplicity, we assume that the cursor position is immediately after a white space, even though we have not formally defined a white space in this study. We believe a cursor can be easily allowed to remain in the middle of a lexeme, which will be investigated in a future study. The parsing phase is assumed to comprise lexical and syntax analyses.

3 Generation of Definitions for Prefix Syntax

As was mentioned in the previous section, we parse the program up to the cursor position and compute some candidates that are to be completed. One method by which this can be achieved is by using the internal information of parsers, which may be generated using parser generators like Yacc. Another method is to transform the given syntax definition to one for the program text up to the cursor position. In this study, we adopt the latter method.

In this section, we present an algorithm for transforming the syntax description in BNF to one in BNF for the program text up to the cursor position, which is referred to as *prefix syntax*. To avoid cluttering, only one nonterminal symbol is assumed to be present in the syntax of the language in this section.

Suppose a nonterminal symbol N is defined in BNF as follows:

$$\begin{array}{rcl} N & := & \ldots \\ & | & X_1 \ldots X_k \\ & | & \ldots . \end{array}$$

In this section, only one nonterminal symbol is assumed to be present; therefore, the above is all the definition of the syntax of the language. Here, the symbol X with some subscripts denotes either terminal or nonterminal symbols.

First, we present a methodology to generate the productions of prefix syntax. We make a nonterminal symbol P corresponding to the nonterminal symbol N. For each production for N

$$N := X_1 \ldots X_k,$$

we generate some productions for P in the following manner. For each $X_j = N$, we generate a production of the form

$$P := X_1 \ldots X_{j-1}\ P.$$

For $X_j \neq N$, we generate a production of the form

$$P := X_1 \ldots X_{j-1}\ ␣,$$

where ␣ represents the cursor position. The generated productions for P include the following production:

$$P := ␣$$

When more than one nonterminal symbols are present in the original syntax description, a nonterminal symbol is created for each one; subsequently, the productions for the created symbols are generated in a similar manner.

4 Computing Candidates

In our approach, we use Yacc-like systems to implement a system for computing the candidates that are to be completed. We write an action for each production to construct a parse tree. After constructing the parse tree for the program text up to the cursor position, we compute the candidates that are to be completed. We add a candidate when visiting each node in the parse tree. Instead of presenting a general algorithm, we illustrate it by using some constructs in the language M in Section 2. We compute candidates in accordance with the program structure.

Initially, we consider the following production:

$$P := (P.$$

Let the list of candidates for P on the right-hand side be $[a_1, \ldots, a_n]$. We construct the list of candidates for P on the left-hand side by adding a candidate a_0 to the list. Here, we construct a_0 by concatenating a_1 with a close parenthesis `)`. Then, the candidates of P on the left-hand side are given as follows:

$$[a_0, a_1, \ldots, a_n].$$

This construction is derived from the original production:

$$M := (M).$$

Next, consider the following production:

$$P := \texttt{let val ID =}\ P.$$

Let the candidates for P on the right-hand side be $[a_1, \ldots, a_n]$. We construct a_0 by concatenating a_1 with the sequence `in` M `end`. Then, the candidates for P on the left-hand side are given as follows:

$$[a_0, a_1, \ldots, a_n].$$

This construction is derived from the original production:

$$M := \texttt{let val ID =}\ M\ \texttt{in}\ M\ \texttt{end}.$$

Until now, we presented the manner in which candidates can be obtained for some syntactic components apart from terminals. Now, let us consider the following production for the cursor.

$$P := ␣$$

When the identifiers $x_1, \ldots, x_n$ are within scope at the cursor position, let the list of candidates for the cursor ␣ be

$$[[\,], x_1, \ldots, x_n],$$

where [ ] represents a hole (not the empty list) which should later be rewritten by programmers. In the language M in Section 2, a hole [ ] corresponds to M. In the implementation, a hole [ ] must be represented as some string like `...`, which will be argued in Section 5.2. By putting [ ] at the head of the list, identifiers do not appear in case of a candidate possessing more than one token, as presented in Figures 1 and 2.

At times, many identifiers are present within their scope at the cursor position. In the language considered in this section, the number of identifiers that can be completed are the depth of nesting of the `fn` expressions (function abstractions) and let expressions at the cursor position. If the identifiers are combined with some other syntactic components in the candidates, the number of candidates would be the product obtained by multiplying the depth of nesting of the syntactic components at the cursor position with the number of identifiers within their scope at the cursor position. However, the programmers do not expect these types of candidates.

In the current implementation, as described above, candidates are accumulated from the bottom to the top of the parse tree of the program text up to the cursor position, excluding identifiers. We collect the identifiers from the top to the cursor node and then add a string like `...` to the list of the identifiers. The obtained list is the candidates for the cursor ␣. Because the identifiers in a buffer can be easily collected without considering the type consistency or other program

modules, such as libraries, the detailed description can be omitted. When completing the identifiers in a large source code containing libraries or modules, many things should be considered; therefore, we will present some relevant previous work in Section 6.

5 Implementation

Based on the approach presented in the previous sections, a prototype system is implemented to complete the syntax in a small subset of Standard ML as an Emacs mode. The source code for the mode is available on the web page[1]. The implementation uses Yacc to generate a parser for the program text up to the cursor position.

5.1 An Example

We consider the following syntax as an example.

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

Here, *start* is the start symbol. Although the language generated based on the aforementioned definition is the same as that generated based on the definition in Section 2, the former is used to avoid conflicts when generating the parser table in Yacc. The grammar in Section 2 is ambiguous because `x y z` can be parsed as either `(x y) z` or `x (y z)` and the grammar described above is unambiguous. Although any LR(k) grammar is unambiguous [9], people may allow some ambiguity, typically in cases in which shift/reduce conflicts occur when generating parser tables using Yacc.

5.2 Holes

The syntax to be completed in the previous section may contain nonterminal symbols, including *exp* and *dec*. They are *holes* that should be rewritten by programmers. We represent the nonterminal symbols in candidates by some string such as three dots ..., which should not be a legal lexeme in the language. Therefore, the string must be manually rewritten by the programmers.

Also, the syntax may contain terminal symbols, such as `ID`, to which many lexemes correspond. When `ID` occurs at the position of *use* and does not occur at the position of *declaration*, we include identifiers as candidates to be completed by considering the scope rule.

[1] http://www.cs.ise.shibaura-it.ac.jp/pepm2020/

5.3 Cursor

Syntax completion is implemented as an Emacs mode. The cursor ␣ is recognized based on the following fragment in Lex specification.

```
[ ]+ {
num_chars+=yyleng;
if (num_chars == position-1) {
 cursorlen=0;
 return (CURSOR);}
}
```

Currently, as we mentioned in Section 1, syntax completion occurs only when the cursor is located immediately after one or more white characters, which is checked using the aforementioned code.

5.4 Structure of the Emacs Mode

The Emacs mode is implemented using Emacs Lisp and C, where the program in C serves as a server to compute the candidates and the program in Emacs Lisp serves as a client to send all the program text in the current buffer as well as the location of the cursor to the server through TCP/IP. It is common to write a relatively complex computation in languages apart from Emacs Lisp. Lex and Yacc are used to generate the program in C.

5.5 Action Description

We illustrate the manner in which actions can be written using the small language in Section 5.1. The following is part of the syntax and action description for the prefixes of the *atexp* expressions. Currently, this is written manually.

```
patexp
: CURSOR
{
  $$ = make_patexp1_CURSOR ();
}
| '(' pexp
{
  $$ = make_patexp2 ($2);
}
| LET dec IN pexp
{
  $$ = make_patexp3 ($2, $4);
}
```

In the action part, the parse trees of prefix syntax are constructed. After the construction of the parse tree, the candidates are computed, as described in Section 4. Although the actions and the functions to compute the candidates are hand-written, we expect that they can be mechanically generated.

5.6 User Interaction

We implement an Emacs mode named smlcomplete mode, using the Emacs Lisp program `popup.el`, which is included in

the auto-complete mode[2], for popping up candidates. By typing the TAB key, the mode computes the candidates that are to be popped up and subsequently calls a function `popup-menu*` defined in `popup.el`. When some candidates are obtained as the result, they are popped up.

5.7 Examples

We present a screenshot of the mode In Figure 1, which was obtained when the programmer had just typed the TAB key on the cursor, located after the space that follows the final occurrence of y.

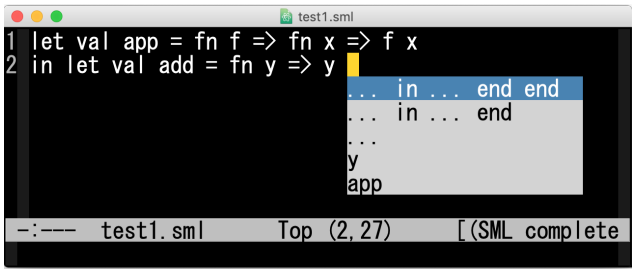

Figure 1. A screen shot of smlcomplete mode

We present another screenshot in Fig. 2 for completing parentheses, which is one of the simplest syntax completion. The close parentheses are popped up for the open parentheses that have not yet been closed.

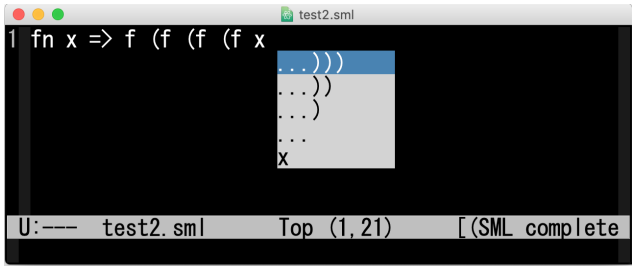

Figure 2. A screen shot of smlcomplete mode for completing parentheses

6 Related Work

In this study, we showed a method to transform a syntax to some of its prefix syntax. We would not like to claim that the transformation is novel. Several studies have investigated syntax transformation (or grammar transformation). Syntax transformation has been investigated with respect to various systems, e.g., the TXL system [3], where the authors refer to it as *grammar programming*. TXL supports structural program transformation.

[2] http://www.emacswiki.org/emacs/AutoComplete

Several methodologies have been presented for syntax completion apart from the syntax transformation methodology presented in this study. Here, we list some of those methodologies.

A recent study [1] has investigated syntax completion, which has been referred to by the authors as *syntactic code completion*. They represent incomplete program text using placeholders. On their IDE, users select a placeholder for various language constructs, e.g., expressions, and syntactic candidates are popped up. The IDE inserts a syntactic component by selecting one of them. In their approach, the incomplete program text is (syntactically) complete except for the placeholders. In contrast, we do not use placeholders in their sense and allow an arbitrary program text to be present after the cursor position.

Further, a previously conducted study [23] has 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. In contrast, we allow syntactically incomplete program text and syntax completion, resulting in a syntactically incomplete program text.

Kuiper *et al.* [11, 20] implemented a system named LRC, which generates graphical language-oriented tools; however, the LRC system is no longer available. The LRC system takes a higher order attribute grammar (HAG) [25] 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 will allow users to edit the source code either via GUI or on a character basis, editors exhibiting a syntax completion functionality have not been generated by LRC.

In 2012, Perelman *et al.* [15] proposed an algorithm to complete expressions, which is not for completing syntax, in object-oriented languages. The work requires programmers to write an expression with holes, which can be referred to as a *partial expression*, and subsequently computes several candidates to be filled in the holes exhibiting some ordering. The Agda language provides typed holes since the 1990s; more recently, GHC, a Haskell compiler, enables programmers to use typed holes so that they can obtain some information that may help in filling the hole. In contrast, our system does not require programmers to write such expressions using holes.

Several studies have investigated identifier completion. In 2013, Sasano and Goto [22] proposed an algorithm to complete identifiers when the program text up to the cursor position is consistent with respect to the syntax and types. Later, Sasano [21] presented an approach to complete identifiers by coping with the incomplete program text based on LR parsing error recovery. In his study, the error recovery behavior is manually specified using the Yacc specification. De Jonge *et al.* [2] developed a method to derive error recovery rules

based on grammar specification. In the future, these studies may be utilized for syntax completion in case of incomplete program text that may not necessarily be syntactically complete up to the cursor position. Rittri [17] developed a method to search for an identifier in a program library using types as search keys. Runciman *et al.* [19] independently developed a method for the same problem by considering the types that can be unified.

Gvero *et al.* [5] presented an approach for code completion using languages with a parametrically polymorphic type system. They developed a solution with respect to the type inhabitation problem: for a given type environment Γ and a type τ, try to find an expression e such that the type judgment $\Gamma \triangleright e : \tau$ holds. They compute Γ based on the cursor position and τ by examining the declared types appearing up to the cursor position. Our system does not use the typing information. Even though they use types, their approach did not cope with the partial terms.

Robbes *et al.* [18] claims that finding a candidate in the popup window is cumbersome or even slower than typing the full name when using completion engines, 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. In contrast, our system does not limit the number of candidates.

Several studies have investigated generating a portion of programs; some of these studies are discussed herein. Hashimoto [6] constructed an ML-style programming language with first-class contexts, *i.e.*, expressions with holes. Our study also uses holes in some sense; however, his language has a language construct for filling holes. Some tools have been developed to generate terms under type constraints. Djinn[3] generates a term having a given type. The tool proposed by Katayama [8] generates a minimum term having a given type. Jeuring *et al.* [7] published a technique for generating generic functions, and a previous study [10] investigates the synthesis of functions matching a set of input-output pairs. These studies are not directly related to our problem.

7 Conclusions and Future Work

We believe that this study is the first step toward obtaining formally specified functionalities in IDE's, including syntax completion, which solves the inconvenience associated with the usage of the so-called structured or projectional editors. We also believe that the syntax of the language need not be restricted when using our approach as long as the programs in the language can be parsed using LR parsers. Several investigations have to be conducted in the future.

One investigation is related to the manner in which the program text is parsed and the candidates are computed. Two approaches have been presented in the beginning of Section 3.

One approach is to use the internal information of parsers. A recent study [13] has investigated the usage of libraries for writing LR parsers using a single language, where the users need not use some other languages, such as BNF in `.y` files in Yacc, similarly to parser combinators, e.g., parsing expression grammars, or PLY, which is a Lex/Yacc for Python language implemented by utilizing the reflection or introspection functionality in Python. Libraries [13] provide methodologies to access the internal information in parsers. Therefore, using the libraries, we expect to implement syntax completion by accessing the internal information of the parser based on the specification presented in Section 2.

Another approach is to transform the syntax of the language to some of its prefix syntax. Although the prefix syntax and the program to compute the candidates are manually written in our implementation, we expect that we can implement the syntax transformation based on the approach presented in the Section 3.

Further, another approach of using attribute grammars may also be used. In this study, we have manually written the code to collect identifiers, which are within scope at the cursor position. Using attribute grammars, we may be able to uniformly achieve syntax completion and identifier completion. We may also consider the use of parsers apart from LR, such as the ANTLR [14] and GLR [12, 24] parsers.

Some future studies are as follows:

- We would like to present an algorithm for syntax completion as well as properties of the candidates.
- In this study, we 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.

Acknowledgments

We would like to thank Fritz Henglein and an anonymous reviewer of a domestic workshop PPL 2019 in Japan, both for suggesting the use of sentential forms in LR parsing. We would also like to thank Kwanghoon Choi for discussing the use of the libraries, recently developed by him for writing LR parsers. Further, we would like to thank the anonymous reviewers of PEPM 2020 for their many helpful comments, including one on the LRC system and one on the usage of attribute grammars to uniformly achieve both syntax and identifier completion. We would also like to thank João Saraiva for answering questions about LRC and denoting a paper that mentions editors generated by LRC. This work was partially supported by JSPS KAKENHI under Grant Number 16K00106.

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

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

[6] Masatomo Hashimoto. 1998. First-class contexts in ML. In *Asian Computing Science Conference (Lecture Notes in Computer Science)*, Vol. 1538. Springer, 206–223.

[7] 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.

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

[9] Donald E. Knuth. 1965. On the translation of languages from left to right. *Information and Control* 8, 6 (1965), 607–639.

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

[11] 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.

[12] 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.

[13] Jintaeck Lim, Gayoung Kim, Seunghyun Shin, Kwanghoon Choi, and Iksoon Kim. 2019. A method for efficient development of parsers via modular LR automaton generation (in Korean). In *Proceedings of Korea Computer Congress (KCC 2019)*. Korea, 1578–1580.

[14] 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 (PLDI '11)*. 425–436.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] Masaru Tomita. 1985. *Efficient parsing for natural language: A fast algorithm for practical systems.* Kluwer Academic Publishers.

[25] 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 (PLDI '89)*. ACM, 131–145.