Silverchain: A Fluent API Generator

Tomoki Nakamaru
The University of Tokyo
Japan
nakamaru@csg.ci.i.u-tokyo.ac.jp

Kazuhiro Ichikawa
The University of Tokyo
Japan
ichikawa@csg.ci.i.u-tokyo.ac.jp

Tetsuro Yamazaki
The University of Tokyo
Japan
yamazaki@csg.ci.i.u-tokyo.ac.jp

Shigeru Chiba
The University of Tokyo
Japan
chiba@acm.org

Abstract

This paper presents a tool named *Silverchain*, which generates class definitions for a fluent API from the grammar of the API. A fluent API is an API that is used by method chaining and its grammar is a BNF-like set of rules that defines method chains accepted in type checking. Fluent APIs generated by Silverchain provide two styles of APIs: One is for building a chain by concatenating all method calls in series. The other is for building a chain from partial chains by passing child chains to method calls in the parent chain as their arguments. To generate such a fluent API, Silverchain first translates given grammar into a set of deterministic pushdown automata without ϵ-transitions, then encodes these automata into class definitions. Each constructed automata corresponds to a nonterminal in given grammar and recognizes symbol sequences produced from its corresponding nonterminal.

CCS Concepts • **Software and its engineering** → **Domain specific languages**; *Source code generation*;

Keywords Fluent API, Embedded DSL

ACM Reference Format:
Tomoki Nakamaru, Kazuhiro Ichikawa, Tetsuro Yamazaki, and Shigeru Chiba. 2017. Silverchain: A Fluent API Generator. In *Proceedings of 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3136040.3136041

1 Introduction

A fluent API [10] is an API that is used by method chaining and a number of well-known class libraries provide their fluent APIs such as Hamcrest [15] and jMock [21]. Method chaining is a coding style that concatenates consecutive method calls together like a chain. By method chaining, programmers can group semantically related method calls into a single expression and keep code easy to understand. Fluent APIs are a popular way of designing embedded domain-specific languages (EDSLs), a form of DSLs that is implemented as a library in a general-purpose host language [17]. Using a *fluent EDSL*, DSL sentences can be emulated well without any change of the host language. For instance, Java programmers can compose an SQL query with jOOQ [12] as follows:

```
// SELECT * FROM book
//     WHERE book.published_in = 2016;
String query = create.select()
        .from(BOOK)
        .where(BOOK.PUBLISHED_IN.eq(2016)).getSQL();
```

Fluent APIs usually support *subchaining style*, which is a style of method chaining that passes a chain to a method call in another chain as its argument. In the case of the example above, subchaining style is used in the `where` clause. This feature improves the usability of fluent APIs in that programmers can build an entire chain from semantically grouped partial chains. Note that subchaining style is effective for a long or complex chain but not for a small or simple chain. *Non-subchaining style*, a style that concatenates all method calls in series, is useful when building a small or simple chain. To improve the usability further, fluent APIs are frequently designed to be *type-safe* by setting the return type of each method based on which methods can be written next [2, 10]. The users of a type-safe fluent API can find misuse easily from thrown type errors and autocompletion can provide necessary and sufficient candidate methods to the users.

Creating a well-designed fluent API, however, is not as easy as using it. To make a fluent API type-safe, API developers need to define a lot of classes and choose the return type of each method carefully. Further, each class definition gets complicated when making fluent APIs support both subchaining and non-subchaining style. In fact, manually written fluent APIs often fail to follow their expected rules perfectly. The users of such incomplete APIs accidentally

GPCE'17, October 23–24, 2017, Vancouver, Canada
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5524-7/17/10...$15.00
https://doi.org/10.1145/3136040.3136041

build a chain causing runtime errors, and autocompletion provides unnecessary or insufficient candidates to the users.

In this paper, we present a tool named *Silverchain* for generating type-safe fluent APIs that support both subchaining and non-subchaining style. The input to Silverchain is the grammar of a target fluent API, a BNF-like set of rules that defines method chains accepted in type checking. From given grammar, Silverchain first constructs a set of deterministic pushdown automata without ϵ-transitions (called real-time deterministic pushdown automaton, or RDPDA for short), each of which recognizes symbol sequences produced from its corresponding nonterminal in the grammar. These constructed RDPDAs are then encoded into class definitions for the target fluent API. Silverchain can generate a fluent API for any context-free grammar (CFG) but cannot generate an API that fully supports non-subchaining style for some kinds of grammars. This limitation is due to our construction method of RDPDAs. Silverchain differs from other generators (or generating algorithms) in that generated APIs support subchaining and non-subchaining styles. We describe those other generators later in Section 5.

2 Motivation

As a motivating example, consider creating a Java fluent API for *IDoc*, a small DSL for writing itemized documents. The following code shows an example use of IDoc and its translation into TEX:

```
begin
  "Item 1"
  "Item 2"
    begin
      "Item 2.1"
      "Item 2.2"
    end
end

// Translation to TeX
\begin{itemize} % Just "begin" in IDoc
  \item Item 1
  \item Item 2 % Quoted string in IDoc
    \begin{itemize}
      \item Item 2.1
      \item Item 2.2
    \end{itemize} % Just "end" in IDoc
\end{itemize}
```

In IDoc, itemized documents can be written more simply since IDoc is specialized to itemized documents.

Using a Java fluent API for IDoc, programmers can compose an itemized document in Java as if they write it in IDoc. For instance, the IDoc sentence above is emulated in Java by method chaining as follows:

```
String tex = idoc()
    .begin()
    .text("Item 1")
    .text("Item 2")
    .list(begin()
        .text("Item 2.1")
        .text("Item 2.2")
        .end())
    .end().toTeX();
```

With a fluent API, DSL sentences are embedded without any change of the host language while those embedded sentences maintain domain-specific syntax. When using a *non-fluent* API, the document is emulated in a different way as follows:

```
IDoc idoc = new IDoc();
idoc.addItem(new Item("Item 1"));
Item item2 = new Item("Item 2");
item2.addItem(new Item("Item 2.1"));
item2.addItem(new Item("Item 2.2"));
idoc.addItem(item2);
String tex = idoc.toTeX();
```

Without method chaining, code tends to contain meaningless intermediate variables and does not resemble original DSL sentences. *String encoding* and *syntax extension* are also techniques to embed DSL sentences into Java source code. By string encoding, DSL sentences are directly written as string literals in Java. Those sentences maintain domain-specific syntax but are often unreadable especially when they are long or contain escape sequences. Furthermore, no syntax checking is applied to DSL sentences at compile-time. By extending Java syntax, programmers can compose a DSL sentence in Java as it is but introducing a syntax extension mechanism is still a research topic [9, 19]. Fluent APIs can be activated easily by importing them since they are just libraries.

From the viewpoint of usability, it is essential to support both subchaining and non-subchaining style in one fluent API. Without subchaining, the example document is emulated as follows:

```
String tex = idoc()
    .begin()
    .text("Item 1")
    .text("Item 2")
    .begin() // Nesting structure is flattened.
    .text("Item 2.1")
    .text("Item 2.2")
    .end()
    .end().toTeX();
```

The readability gets worse as the length or the complexity of a chain increases if a fluent API does not support subchaining style. The readability gets better by indentation that reflects original nesting structures of the sentence, but such indentation is not naturally obtained with support from an integrated development environment (IDE). Subchaining style is also effective when changing a part of a chain dynamically as follows:

```
List list1 = begin() ... .end();
List list2 = begin() ... .end();
String tex1 = idoc()
    .begin()
    .text("Item")
    .list(bool? list1 : list2)
    .end().toTeX();

// If only non-subchaining style is supported.
IncompChain1 c1 = idoc().begin().text("Item");
IncompChain2 c2;
if (bool) {
  c2 = c1.begin() ... .end(); // list1
} else {
  c2 = c1.begin() ... .end(); // list2
}
```

```
String tex2 = c2.end().toTeX();
```

When subchaining style is supported, the users can build a chain by combining the base chain and semantically grouped subchains according to satisfied conditions. On the other hand, when subchaining is not supported, intermediate variables get meaningless and the users cannot make good use of method chaining. Non-subchaining style, however, is also important as well as subchaining style. Non-subchaining style can reduce redundant parts in a chain while maintaining the readability if a chain is simple enough:

```
// Simple chain in non-subchaining style
String tex1 = idoc()
    .begin().text("Item").end().toTeX();

// If only subchaining style is supported.
String tex2 = idoc()
    .list(begin().item(text("Item")).end()).toTeX();
```

As seen from the above, which style should be used depends on the situation and two styles are often mixed in a chain. Fluent APIs with only subchaining support (or only non-subchaining support) prevent their users from utilizing method chaining effectively.

Fluent APIs achieve safety by setting the return type of each method based on which methods can be written next. The users of such type-safe fluent APIs can find misuse easily from thrown type errors:

```
// Missing "end()" call causes a type error.
String tex = idoc()
    .begin()
    .text("Item") // Returns a type that
                  //   does not have "toTeX()".
    .toTeX();
```

The type-safety also enhances the usability in that autocompletion of an IDE can provide necessary and sufficient hints. The users can build correct chains easily just by following provided hints and do not have to pay attention to accidental runtime errors. In other words, an existing IDE for a host language serves as an IDE for the DSL if a fluent API is type-safe. Here, type-safety indicates the syntactic correctness from the perspective of method arrangement, not of the semantics of written chains.

Although well-designed fluent APIs have many advantages, creating such APIs is a tedious task. API developers need to define a lot of classes and connect them appropriately by choosing the return type of each method. Moreover, the developers need to repeat this tedious task for each language in which they provide the API. In fact, many well-known fluent APIs fail to follow their desired rules perfectly. For instance, the following chains written with popular fluent APIs are accepted by Java type checker but those should not be accepted from the viewpoint of the validity of DSL sentences:

```
// <div src="./image.png"></div> with j2html
div().withSrc("./image.png").render();

// "SELECT * ORDER BY book.id" with jOOQ
select().orderBy(BOOK.id).getSQL();
```

```
1 idoc -> list* ;
2 list -> "begin" item+ "end" ;
3 item -> text list? ;
4
5 START = idoc ;
6 text : STRING ;
```

Figure 1. The grammar of the fluent API for IDoc

The incomplete type-safety not only is unsafe but also worsen the user experiences in that autocompletion suggests unnecessary or insufficient methods for hints.

3 Silverchain

The input to Silverchain is the grammar of a fluent API written in a BNF-like DSL such as shown in Figure 1, which is the grammar of a fluent API for IDoc. In the grammar, a valid chain of methods (a chain accepted in type checking) is defined by a set of production rules. Each production rule specifies a valid method chain that begins with the method corresponding to the left-hand side of the rule. For example, three production rules in Line 1 to 3 in Figure 1 specify that the following chains in subchaining style are accepted (or rejected) by a type checker:

```
// "..." represents a subchain.
// Line 1
idoc().list(...).list(...); // OK

// Line 2
list().begin().item(...).end(); // OK
list().begin().end(); // Error

// Line 3
item().text("Item").list(...); // OK
item().list(...); // Error
```

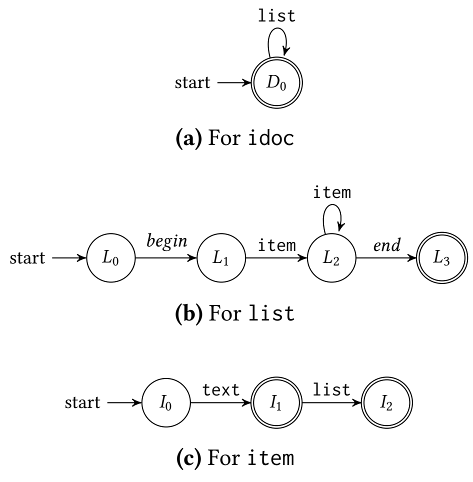
Those rules also specify that the following chains in non-subchaining style are accepted (or rejected):

```
// Line 1 and Line 2
idoc()
  .begin().item(...).end()
  .begin().item(...).end();; // OK

// Line 2 and Line 3
list().begin().text("Item").end(); // OK
list().begin().list(...); // Error

// Line 1, Line 2, and Line 3
idoc()
  .begin().text("Item 1").end()
  .begin().text("Item 2").end(); // OK
```

Line 5 and Line 6 are supplementary rules. The rule in Line 5 specifies that only a chain beginning with `idoc()` can be a top-level chain. A top-level chain can be terminated by `eval()` as follows:

```
idoc().list(...).eval(); // OK
list().begin().item(...).end().eval(); // Error
item().text("Item").eval(); // Error
```

Here, `eval()` is a method call to evaluate (or materialize) a chain. The rule in Line 6 specifies that the symbol `text` is translated into a method call taking a string as its argument.



(a) For `idoc`

(b) For `list`

(c) For `item`

Figure 2. DFAs for nonterminals in Figure 1

Our translation from grammar to a fluent API consists of four steps. First three steps are for constructing a set of real-time deterministic pushdown automata (RDPDAs). Each RDPDA corresponds to a nonterminal in given grammar and recognizes all symbol sequences produced from its corresponding nonterminal. To construct such RDPDAs, we construct a set of DFAs from given grammar and then incrementally extend those DFAs to RDPDAs. At the final step, we encode constructed RDPDAs into class definitions.

Silverchain can generate a fluent API for any context-free grammar but, for some kinds of grammars, cannot generate an API that fully supports non-subchaining style. In these cases, Silverchain generates a fluent API that requires subchaining style to build certain parts of a chain. This limitation is due to the third step of our translation and it is discussed in Section 3.3.

3.1 Constructing DFAs from Production Rules

At the first step, Silverchain constructs a DFA for each nonterminal in given grammar by converting the right-hand side of the production rule for the nonterminal. In the case of the grammar in Figure 1, Silverchain constructs three DFAs in Figure 2 at this step. Each DFA is constructed so as to recognize symbol sequences produced *directly* from the corresponding nonterminal. For example, the DFA for `list` recognizes the sequence "*begin* `item` `item` *end*" as seen from Figure 2b. Since the right-hand side of a production rule is a regular expression consisting of terminals and nonterminals, such a DFA can be constructed from a production rule by well-known algorithms (e.g. Brzozowski's [4] and Thompson's [28]).

By encoding each state of DFAs into a class and each transition to a method, a type-safe fluent API is obtained though it only supports subchaining style. For example, the Java classes in Figure 3 are generated from the DFAs in

```
// Class grouping the classes related to idoc.
class Idoc {
  void eval() {}

  // Groups static methods to begin a chain.
  class StartingMethods {
    static State0 idoc() { ... }
  }
  // An accepting state is encoded
  //         into a subclass of Idoc.
  class State0 extends Idoc {
    // Takes a subchain for list.
    State0 list(List l) { ... }
  }
}

class List {
  class StartingMethods {
    static State0 list() { ... }
  }
  class State0 {
    State1 begin() { ... }
  }
  class State1 {
    State2 item(Item i) { ... }
  }
  class State2 {
    State2 item(Item i) { ... }
    State3 end() { ... }
  }
  class State3 extends List {
  }
}

class Item {
  class StartingMethods {
    static State0 item() { ... }
  }
  class State0 {
    State1 text(String t) { ... }
  }
  class State1 extends Item {
    State2 list(List l) { ... }
  }
  class State2 extends Item {
  }
}
```

Figure 3. Java classes generated by encoding Figure 2

Figure 2 by this encoding scheme. (The encoding scheme is described again in Section 3.4.) The users of the API defined in Figure 3 can compose an itemized document by method chaining as follows:

```
// Subchaining style
idoc().list(
    list().begin().item(
        item().text("Item")
    ).end()
).eval(); // begin "Item" end
```

3.2 Extending DFAs

The fluent API obtained at the previous step does not support non-subchaining style as follows:

```
// Partially non-subchaining style
idoc().list(
    list().begin().text("Item").end()).eval();
```

To support method chaining above, Silverchain extends the DFAs that were constructed at the previous step. A DFA is extended so as to recognize symbol sequences produced

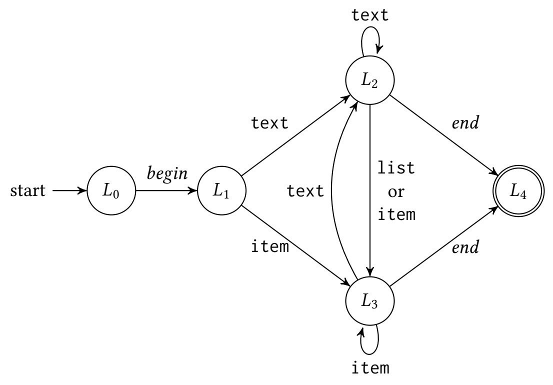


Figure 4. New DFA for `list`

```
class List {
  class StartingMethods {
    static State0 list() { ... }
  }
  class State0 {
    State1 begin() { ... }
  }
  class State1 {
    State2 text(String t) { ... }
    State3 item(Item i) { ... }
  }
  class State2 {
    State2 text(String t) { ... }
    State3 list(List l) { ... }
    State3 item(Item i) { ... }
    State4 end() { ... }
  }
  class State3 {
    State2 text(String t) { ... }
    State3 item(Item i) { ... }
    State4 end() { ... }
  }
  class State4 extends List {
  }
}
```

Figure 5. Java classes generated by encoding Figure 4

not only directly but also *indirectly* from the corresponding nonterminal. For example, the DFA for `list` (Figure 2b) is extended as shown in Figure 4 at this step. As seen from Figure 4, the extended DFA recognizes sequences produced indirectly from `list` such as “*begin* `text` *end*” while it also recognizes sequences produced directly from `list` such as “*begin* `item` *end*”. The fluent API generated from the extended DFAs supports non-subchaining style. The classes in Figure 5 are generated from the DFA in Figure 4 and they allow the following code:

```
// Newly supported style by the extension
List l1 = list().begin().text("Item").end();

// Subchaining style is also supported.
List l2 = list().begin().item(...).end();
```

The extension of a DFA can be regarded as inline expansion of nonterminals in the production rule that corresponds to the DFA. For example, the extension from Figure 2b to Figure 4 is modeled as follows:

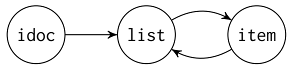
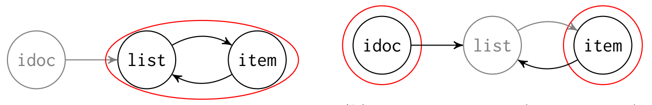


Figure 6. The initial reference relation graph for Figure 1



(a) Examining (`idoc`, `list`). This edge is not selected since the SCC containing `list` consists of two nodes.

(b) Examining (`list`, `item`). This edge is selected since the SCC containing `item` consists only of one node without a self-loop.

Figure 7. Selecting an edge for inline expansion

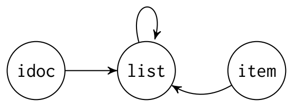
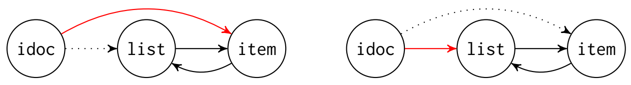


Figure 8. The updated graph after expanding `item` in the rule for `list`



(a) Expanding (`idoc`, `list`). **(b)** Expanding (`idoc`, `item`).

Figure 9. Without decomposition into SCCs

```
// Before expansion
list -> "begin" item+ "end";

// After expansion
list -> "begin" (item | (text list?))+ "end" ;
```

A nonterminal on the right-hand side of a production rule is replaced by the union of that nonterminal and its directly producing sequence. In fact, Figure 4 is constructed from the expanded rule above. Note that the inline expansion is applied to nonterminals only when they have not been expanded yet. Redundant inline expansion of nonterminals does not extend the corresponding DFA.

To avoid infinite regression caused by inline expansion, Silverchain selects a nonterminal to be expanded as follows. It first examines reference relations among nonterminals, where a reference from a nonterminal n_s to a nonterminal n_d represents that the rule for n_s contains one or more unexpanded n_d on its right-hand side. For example, Figure 1 contains a reference from `list` to `item`. Silverchain considers these reference relations when selecting an expanding nonterminals:

(1) Create a graph G. A node in G corresponds to each nonterminal in given grammar. An edge (n_s, n_d) in G corresponds to a relation from n_s to n_d. Figure 6 shows a graph created from the grammar in Figure 1.

(2) Select an edge (n_s, n_d) such that the following subprocess returns true for (n_s, n_d):
   a. Create a subgraph G' of G by removing n_s and the edges with n_s on their source or destination.
   b. Decompose G' into strongly connected components (SCCs) and let the component containing n_d be C.
   c. Return true if C consists of a single node without a self-loop, and false otherwise.

   If no edge is found, Silverchain finishes the extension of DFAs. Silverchain examines edges in the order that an edge further from the node for the *start nonterminal* is examined earlier. Here, the start nonterminal is a nonterminal that is specified on the right-hand side of the supplementary rule with = in given grammar. In the case of G in Figure 6, Silverchain selects (`list`, `item`) through the subprocesses shown in Figure 7.

(3) Apply inline expansion to the production rule for n_s. All the occurrences of n_d in the rule are expanded. For example, when n_s is `list` and n_d is `item`, then the rule for `list` will be expanded into:

```
list -> "begin" (item|(text list?))+ "end" ;
```

   as we have seen before.

(4) Update G to reflect the inline expansion above. The selected edge (n_s, n_d) is removed. New edges are added to G to represent reference relations in the expanded production rule. Then, go back to (2). G is updated to the graph in Figure 8 when (`list`, `item`) is selected for expansion.

Decomposition into SCC at (2) is required to avoid an infinite regression of inline expansion caused by mutual recursion in grammar. Suppose that the edge (`idoc`, `list`) in Figure 6 is selected for expansion. The dotted edge is removed and the red edge is added as shown in Figure 9a. Then the edge (`idoc`, `item`) in Figure 9a is selected for expansion and the graph is updated to Figure 9b, where the dotted edge is removed and the red edge is added. Figure 9b is equivalent to Figure 6. Further inline expansion will cause infinite regression. By the decomposition, Silverchain selects an edge composing a mutual-recursion cycle earlier than other edges. Such an edge is expanded and the cycle is transformed into non-cyclic edges and self-loops. Which edge is first selected among cyclic edges does not matter to resolve infinite regression.

3.3 Constructing RDPDAs from DFAs

With the fluent API obtained at the previous step, the users still cannot compose an itemized document as follows:

```
// Fully non-subchaining style
idoc().begin().text("Item").end().eval();
```

Table 1. Constructing a set of RDPDAs

(a) The table representation of DFAs after the second step

		begin	*end*	`list`	`item`	`text`
`idoc`	$\rightleftharpoons D_0$	-	-	D_0	-	-
`list`	$\rightarrow L_0$	L_1	-	-	-	-
	L_1	-	-	-	L_3	L_2
	L_2	-	L_4	L_3	L_3	L_2
	L_3	-	L_4	-	L_3	L_2
	$\leftarrow L_4$	-	-	-	-	-
`item`	$\rightarrow I_0$	-	-	-	-	I_1
	$\leftarrow I_1$	-	-	I_2	-	-
	$\leftarrow I_2$	-	-	-	-	-

(b) The table representation of the constructed RDPDAs at this step

		begin	*end*	`list`	`item`	`text`
`idoc`	$\rightleftharpoons D_0$	~~-~~ $L'_1 D_0$	-	D_0	-	-
`list`	$\rightarrow L_0$	L_1	-	-	-	-
	L_1	-	-	-	L_3	L_2
	L_2	~~-~~ $L'_1 L_3$	L_4	L_3	L_3	L_2
	L_3	-	L_4	-	L_3	L_2
	$\leftarrow L_4$	-	-	-	-	-
`item`	$\rightarrow I_0$	-	-	-	-	I_1
	$\leftarrow I_1$	~~-~~ $L'_1 I_2$	-	I_2	-	-
	$\leftarrow I_2$	-	-	-	-	-
`list'`	L'_0	L'_1	-	-	-	-
	L'_1	-	-	-	L'_3	L'_2
	L'_2	~~-~~ $L'_1 L'_3$	~~L'_4~~ ϵ	L'_3	L'_3	L'_2
	L'_3	-	~~L'_4~~ ϵ	-	L'_3	L'_2
	L'_4	-	-	-	-	-

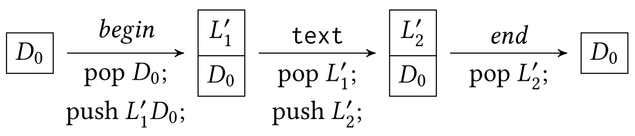


Figure 10. An example of state transitions of the RDPDA for `idoc`

To support method chaining above, Silverchain extends DFAs to RDPDAs (real-time deterministic pushdown automata, deterministic pushdown automata without ϵ-transitions). Since an RDPDA has a stack, it can recognize languages that may contain recursively defined nonterminals in their grammar. Real-timeness, the absence of ϵ-transitions, is important since the encoding scheme mentioned in Section 3.1 cannot generate a method for an ϵ-transition.

To explain the extension at this step, we introduce table representation of a set of automata. In the table representation, all states of an automaton and all input symbols for the automaton are enumerated across the rows and the columns of the table, respectively. Table 1a is the table representation of the three DFAs obtained after the second step (the DFAs in Figure 2a, 2c, and 4). The rows in the table are grouped for each DFA and separated by horizontal lines. A row with $\longrightarrow$ represents the initial state of the DFA corresponding to the group that the row belongs to, and a row with $\longleftarrow$ represents an accepting state of the DFA. Each cell represents the destination of a state after consuming the symbol in the column of the cell.

Table 1b is the table representation of three RDPDAs that Silverchain constructs from three DFAs in Table 1a at this step. In the table representation of RDPDAs, a row with $\longrightarrow$ represents the state stacked at the initialization of the RDPDA for the nonterminal corresponding to the group that the row belongs to. For example, the stack of the RDPDA for `idoc` is initialized with D_0 as specified in the first row in Table 1b. Each cell in the table for RDPDAs represents the states pushed into the stack when consuming the symbol in the column of the cell. Figure 10 shows an example of state transitions of the RDPDA for `idoc` when "*begin* `text` *end*" is input. Every time an RDPDA consumes an input symbol, it pops the top element in its stack and pushes zero or more elements to its stack according to the table. An RDPDA recognizes a symbol sequence if it has the state for a row with $\longleftarrow$ on the top of the stack after consuming the final symbol.

The extension by Silverchain from DFAs to RDPDAs is explained as modifying the table for the DFAs. During the modification, Silverchain duplicates rows in the table. In Table 1b, the lowest five rows are such duplicated rows. The duplication represents recursion in grammar. The cells in red are modified to connect the existing rows (i.e. automata) and duplication. The table is modified as follows:

(i) Find a nonterminal that is not expanded at the previous extension step in Section 3.2 and let the nonterminal be v. If no nonterminal is found, Silverchain finishes the extension from DFAs to RDPDAs. In Table 1a, `list` can be v.

(ii) Duplicate the rows for the DFA for v if the accepting states of the DFA has no transitions. If any of the accepting states has a transition, go back to (i). The five rows at the bottom of Table 1b are such duplications since an accepting state L_4 has no transition.

(iii) Replace the value in a cell with ϵ if the value is a duplicated row for the accepting states of the DFA for v. ϵ does not push anything to a stack. The transition only pops from a stack. The two changes in column *end* are performed at this step.

Table 2. The RDPDAs constructed without checking the emptiness of rows or cells

(a) Duplicating the rows for `T` at (ii)

		begin	*end*	*extra*	`T`
`S`	$\longrightarrow S_0$	$\cancel{-}\,T'_1S_1$	-	-	S_1
	$\longleftarrow S_1$	-	-	-	-
$\vdots$					
`T'`	T'_0	T'_1	-	-	-
	T'_1	$\cancel{-}\,T'_1T'_2$	$\cancel{T'_3}\ \epsilon?$	-	T'_2
	T'_2	-	$\cancel{T'_3}\ \epsilon?$	-	-
	T'_3	-	-	$\cancel{T'_4}\ \epsilon?$	-
	T'_4	-	-	-	-

(b) Overwriting U_0 with V'_1U_0 at (v)

		begin	*end*	`V`
`U`	$\longrightarrow U_0$	$\cancel{U_0}\ V'_1U_0?$	-	U_1
	$\longleftarrow U_1$	-	-	-
$\vdots$				
`V'`	V'_0	V'_1	-	-
	V'_1	$\cancel{V'_3}\ \epsilon$	-	V'_2
	V'_2	-	$\cancel{V'_3}\ \epsilon$	-
	V'_3	-	-	-

(iv) Find a non-empty cell for the initial state of the DFA in the duplicated row. Let the value in that cell be α and its column be γ. When v is `list`, α is L'_1 and γ is *begin*.

(v) For each row ρ that has a non-empty value β in column v, put the series $\alpha\beta$ in row ρ of column γ if the cell in row ρ of column γ is empty. The four changes in column *begin* in Table 1b are applied at this step.

In Table 1b, the transition $L_2 \times begin \longrightarrow L'_1L_3$ added at (v) specifies that, when *begin* is read, a nested nonterminal `list'` starts and the automaton moves to L'_1. The nested nonterminal `list'` ends at the transitions added at (iii), for example, which is $L'_2 \times end \longrightarrow \epsilon$. L'_2 is popped and moved to L_3, which has been pushed at (v).

3.3.1 Limitation

At (ii), Silverchain does not duplicate rows if any of the rows for accepting states is not empty. Hence, the non-subchaining style will not be available for the corresponding nonterminal. At (v), Silverchain does not put the series $\alpha\beta$ if the target cell is not empty. Since the automaton never moves to duplicated rows, the non-subchaining style will not be available either. These cases occur in the following grammars:

```
// (ii)
S -> T ;
T -> "begin" T "end" "extra"?
   | "begin" "end" "extra"? ;

// (v)
U -> "begin"* V ;
V -> "begin" "end" | "begin" V "end" ;
```

Table 2a and 2b are the table representation of the RDPDAs for these grammars after Silverchain modifies tables without considering the issues above. As seen in the tables, the constructed RDPDAs do not recognize sequences that should be recognized according to grammar. For example, the sequence "*begin end extra*" should be recognized but the constructed RDPDA for S in Table 2a does not recognize it. Similary, the constructed RDPDA for U in Table 2b does not recognize "*begin begin end*".

The inline expansion mentioned in Section 3.2 was introduced to avoid the occurrence of the cases mentioned above as much as possible. If the inline expansion is not applied, Silverchain fails to generate a non-subchaining API even for the following regular grammar:

```
W -> "a" ( X | Y ) ;
X -> "b" "c" ;
Y -> "b" "d" ;
```

When given grammar represents a visibly pushdown language (VPL) [1], the cases Silverchain cannot generate non-subchainging API, like the example illustrated by Table 2a and 2b, will never happen. This is because a VPL has the property that symbols to begin/end a nesting structure are not used anywhere else. Further formal discussion and proof are necessary to posit this claim.

3.4 Encoding RDPDAs to Class Definitions

Silverchain generates class definitions by encoding each state into a class and each transition into a method. In the table representation of automata, Silverchain encodes each row into a class and each cell into a method. Stacked states in an RDPDA are represented by a nested generic class. For example, the class `X<Y<Z<BOTTOM>>>` represents a stack that contains `X` on the top and the two states `Y` and `Z` under `X`.

Figure 11 shows Java classes generated from Table 1b. Each top-level class corresponds to a group of rows separated by horizontal lines (e.g. Line 1). It has inner classes for the rows in the group corresponding to it (e.g. Line 12). An inner class extends its outer class if the corresponding row is attributed with ← (e.g. Line 20). The inner class for a row has methods that correspond to cells in the row and each method is named after its corresponding cell's column. The argument type of a method is also determined by the corresponding cell's column. A method takes no arguments when the column is a terminal (e.g. Line 14). It takes a top-level class when the column is a nonterminal (e.g. Line 15). It takes a native type when the column is a symbol that is typed in given grammar (e.g. Line 17).

```
 1 class Idoc {
 2   void eval() { }
 3
 4   class StartingMethods {
 5     static State0<BOTTOM> idoc() { ... }
 6   }
 7   ...
 8 }
 9 ...
10 class List {
11   ...
12   class State2<T> {
13     _List.State1<State3<T>> begin() { ... }
14     State4<T> end() { ... }
15     State3<T> list(List l) { ... }
16     State3<T> item(Item i) { ... }
17     State2<T> text(String t) { ... }
18   }
19   ...
20   class State4<T> extends List {
21   }
22 }
23 ...
24 class _List {
25   ...
26   class State2<T> {
27     State1<State3<T>> begin() { ... }
28     T end() { ... }
29     ...
30   }
31   ...
32 }
33
34 class BOTTOM { }
```

Figure 11. Encoding Table 1b

The return type of a method is determined by the value of the corresponding cell. It returns a nested generic type if the corresponding cell has a value other than ϵ (e.g. Line 16). The outermost type is the class for the first letter in the cell. The inner type is determined by the number of letters in the corresponding cell. If the cell contains two letters, the inner type is a nested generic type consisting of the class for the second letter and the type argument. If the cell contains only one letter, the inner type is the type argument. If a cell is ϵ, the corresponding method returns the type argument (e.g. Line 28).

Top-level classes except ones originated in duplicated rows have the inner class named `StartingMethods` (e.g. Line 4), which has a static method invoked at the beginning of a method chain. Each static method is named the same as its top-level enclosing class. It returns a nested generic type that consists of the inner class for the cell pointed to by → and the class `BOTTOM`. The top-level class for the start nonterminal has `eval`, the method to end a method chain (Line 2). In Figure 11, the body of a method is omitted. How Silverchain generates the method body is explained next.

3.4.1 The Bodies of Methods

Silverchain generates a method so that it will build a list of method names (or an object passed in its argument) invoked by a method chain. The users of Silverchain have only to implement an evaluation method, which interprets the given list of method names. The following shows the bodies of the generated methods in Figure 11:

```
// Line 5 in Figure 11
// Invoked at the beginning of a chain
static State0<BOTTOM> idoc() {
  return new State0<>(new Context());
}

// Line 14 in Figure 11
State4<T> end() {
  context.symbols.add("end");
  return new State4<>(context);
}

// Line 15 in Figure 11
State3<T> list(List list) {
  // Importing sub-context
  context.symbols.addAll(list.context.symbols);
  return new State3<>(context);
}
```

A new context object is created in the static method in `StartingMethods` and other methods append their names to the list `context.symbols`. Silverchain generates an evaluation method `eval` but its body is left empty. The users of Silverchain can customize the evaluation method by referring to `context` as follows:

```
String toTeX() { // void eval() {
  String tex = "";
  for (Object o: context.symbols) {
    // Translate each symbol into TeX
  }
  return tex;
}
```

By using a parser generator in the body of the evaluation method, the users can easily obtain ASTs from `context.symbols`. We need a little trick for generated methods to return a nested parametric type:

```
// Line 27 in Figure 11
State1<State3<T>> begin() {
  context.symbols.add("begin");
  // Store pushed class in the stack.
  context.classes.push(State3.class);
  return new State1<>(context);
}

// Line 28 in Figure 11
@SuppressWarnings("unchecked")
T end() {
  context.symbols.add("end");
  try {
    return (T) context.classes
        .pop() // Pop the stored class.
        .getDeclaredConstructor(Context.class)
        .newInstance(context);
  } catch (Exception e) {
    throw new RuntimeException(e);
  }
}
```

Since a type parameter is not a first class entity in Java, Silverchain uses the reflection API and explicitly passes a type parameter as a class object. Instead of executing such an expression the `new T()`, Silverchain calls `newInstance` on a class object representing `T`.

By fitting all semantic actions into the `eval` method, the users of Silverchain do not need to modify scattered parts over generated classes. This helps the users preserve their own code for semantic actions when re-generating a type-safe interface. However, this scheme requires the users to implement a complicated compiler to evaluate an entire method chain in the `eval` method. This work will be tedious.

3.4.2 Improving Usability

Since a manually developed fluent API often provides several convenience methods, Silverchain also generates such convenience methods. The API generated so far in Figure 11 does not include such convenience methods. Hence, programmers have to write somewhat redundant code like this:

```
String tex = idoc()
  .begin()
  .text("item 1")
  .item(item()    // Redundant call to item()
      .text("item 2")
      .begin()
      .text("item 2.1")
      .text("item 2.2")  // Consecutive calls
                         //     to text()
      .end())
  .end().toTex();
```

`item()` is a static method to start a new subchain but it is fairly obvious that a `list` subchain is placed here as the argument of the enclosing `item(List)` method. The `item()` call might look redundant. These problems can be mitigated by providing convenience methods. For example, programmers could write as follows:

```
String tex = idoc()
  .begin()
  .text("item 1")
  .item(text("item 2")
        .begin()
        .text("item 2.1", "item 2.2")
        .end())
  .end().toTex();
```

`text` in the fourth line is a convenience static method, which starts a new subchain. `text` in the sixth line is another convenience method with variable length arguments, which is equivalent to multiple consecutive calls to `text`.

Silverchain uses the RDPDAs to generate convenience methods. For each RDPDA, it collects the input symbols that the automaton can read at the initial state. Then it generates a static method sharing the same name with each input symbol. In Table 1b, the initial state for `item` is I_0, which is indicated by $\rightarrow$. At I_0, the automaton can read only `text` as an input symbol. Hence, Silverchain generates a static method `text` as a convenience method. It is declared in the `Item.StartingMethods` class. Note that two static methods named `begin` are defined across the API for IDoc. Those methods cannot be used in a single file unless a host language supports alias importing.

When a transition of the RDPDAs is a loop (it is a transition to the same state), Silverchain generates a method taking variable-length arguments. For example, it generates `text` as a method taking a variable-length arguments. In Table 1b, when an automaton at the state L'_1 reads an input symbol `text`, it moves to the state L'_2. When the automaton at the state L'_2 reads `text`, its state L'_2 does not change. Hence, Silverchain generates a method with a variable-length arguments for the transition $L'_1 \times \texttt{text} \rightarrow L'_2$.

4 Use Case

To compare a fluent API generated by Silverchain and a manually written fluent API, we gave the grammar of DOT[1] to Silverchain. DOT is an external DSL for describing a graph and used as follows:

```
// Directed graph with two edges
digraph G {
    A -> B
    B -> A
}
```

The terminals that cannot be converted into method names are replaced as appropriate, for example, "{" is replaced with `beginGraph`. With the fluent API generated by Silverchain, programmers can describe the graph above in Java as follows:

```
digraph().id("G")
    .beginGraph()
    .edge(
        nodeId("A").arrow().nodeId("B"),
        nodeId("B").arrow().nodeId("A")
    )
    .endGraph().eval();
```

With graphviz-java[2], a manually written fluent API, the same graph can be described as follows:

```
graph("G").directed()
    .with(
        node("A").link(node("B"))
        node("B").link(node("A"))
    )
    .toFile(...);
```

As seen from these examples, the generated API can be used as the graphviz-java API. In addition, programmers can describe the graph in non-subchaining style as follows:

```
digraph().id("G")
    .beginGraph()
    .node("A").arrow().node("B")
    .node("B").arrow().node("A")
    .endGraph().eval();
```

The first drawback of the generated API is that redundant method calls are required such as `beginGraph()`. These methods can be omitted by using information obtained from the table representation of RDPDAs. A method can be omitted if it is the only transition that exists between states and no ambiguity arises after omitting that transition. However, if such automatic omission is applied naively, a method chain with the generated API might be unreadable for programmers. Some redundant symbols in a DSL are necessary for programmers to read and understand written sentences. To avoid this problem, Silverchain does not apply that automatic omission. Another drawback of the generated API is that the API uses a lot of mechanically named classes such as `State1` and `State2`. Programmers can hardly edit those classes to provide more enhanced APIs that can be seen in manually written APIs:

```
// With graphviz-java
node("X").with(Shape.RECTANGLE, Style.FILLED);

// With the generated API
node("X").beginAttr()
    .attr("shape=rectangle")
    .attr("style=filled").endAttr():
```

The API provided by graphviz-java uses the domain-specific knowledge of DOT that is not represented in the grammar of DOT. Therefore, it cannot be generated by Silverchain but it is also difficult to add such an extra API to the generated API.

As another use case, we gave the grammar of the DSL used by Silverchain for describing input grammar. The generated API is a fluent API that defines a fluent API. With the API, the grammar of IDoc (Figure 1) is written as shown in Figure 12. Figure 12a shows a chain that expresses the IDoc's grammar without subchaining. Figure 12b shows a chain that uses subchaining style to group semantically related subchains. Figure 12c shows a chain that uses subchaining style to dynamically change a part of the entire chain. The users of the generated API can use any style that suits their usage.

5 Related Work

"Fluent interface" is coined by Fowler et al. [10] and a library with its fluent interface is called a fluent API. The idea to make fluent APIs type-safe is also pointed out by Fowler et al. [10] and its origin can be seen in the idea called *typestate* [27]. There are several studies on fluent APIs that support only non-subchaining style [11, 30]. Although such APIs are not common in practice, they can be regarded as a technique to embed a DSL into a general-purpose language that does not have features to extend its own syntax.

Gil et al. proposed an algorithm to generate a fluent API that is type-safe and supports only non-subchaining style (hereinafter referred to as *no-subchaining API* for short) [11, 24]. The algorithm generates such an API by encoding a jump-stack real-time deterministic pushdown automaton (JRDPDA) into class definitions. Since JRDPDAs can recognize deterministic context-free languages [7] and this class of languages is much larger than the class that RDPDAs can recognize [16], the algorithm of Gil et al. can generate no-subchaining APIs for grammars that Silverchain cannot generate them for.

With a no-subchaining API generated by the algorithm of Gil et al., however, the compilation time of a chain grows exponentially to the length of the chain in the worst case. This is caused by the exponential growth of the size of the type at the end of a chain, as Gil et al. showed in their experiment using Java 8 [11]. Here, the size of a type is defined by the number of type names in the textual representation of the type. The exponential growth of compilation time occurs not only in Java 8 but also in C++ and in Java 9 EA, which has

[1] http://www.graphviz.org/content/dot-language
[2] https://github.com/nidi3/graphviz-java

```
String javaSrc = grammar()
    .START().eq().symbol("idoc").semicolon()
    .symbol("idoc").arrow().symbol("list").star().semicolon()
    .symbol("list").arrow().literal("begin").symbol("item").plus().literal("end").semicolon()
    .symbol("item").arrow().symbol("text").symbol("list").question().semicolon()
    .symbol("text").colon().STRING().semicolon()
    .toJava();
```

(a) Without subchaining

```
Rule start = START().eq().symbol("idoc").semicolon();
Rule idoc = symbol("idoc").arrow().symbol("list").star().semicolon();
Rule list = ... ;
Rule item = ... ;
Rule text = ... ;
String javaSrc = grammar().rule(start, idoc, list, item, text).toJava();
```

(b) Grouping semantically related chains by subchaining

```
Factor items = allowEmptyList? symbol("item").star() : symbol("item").plus() ;
String javaSrc = grammar()
    .START()
    ...
    .symbol("list").arrow().literal("begin").factor(items).literal("end").semicolon()
    ...
    .toJava();
```

(c) Dynamically changing a part of the expression

Figure 12. Code examples using the generated API for the DSL used in Silverchain

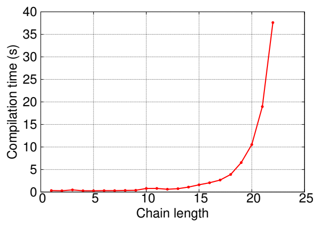
a new type-checking strategy [8, 25]. (Figure 13 shows the experiment in [11] performed on our machine using C++ and Java 9 EA.) Considering these facts, checking an extremely large type essentially takes a long time and the improvement of compiler implementation is unlikely to solve this problem. On the other hand, with APIs generated by Silverchain, the size of the type at the end of a chain grows linearly even in the worst case. For that reason, the compilation time grows at most linearly to the length of a chain. It is also problematic from a practical point of view that the algorithm of Gil et al. does not take subchaining style into account.

In theory, a no-subchaining API can be built for any grammar (including grammars that are not context-free) if the API is implemented in a language with a Turing-complete type system. For instance, if given grammar is context-free, a no-subchaining API can be built by creating a CYK parser [5, 22, 31] implemented by a Turing machine emulated on a type system [14]. However, using a Turing machine on the back is overly complex for most DSLs and often causes practical problems. For example, `javac` throws a `StackOverflowError` even for a small chain of two or three methods with a no-subchaining API generated by Grigore's algorithm [14], one of algorithms that uses a Turing machine emulated on Java type system.

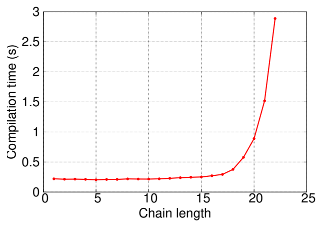
ScaLALR [18] generates a no-subchaining API in Scala from grammar by encoding an LALR parser into class definitions. With APIs generated by ScaLALR, the size of the type at the end of a chain does not grow exponentially to the length of the chain even in the worst case but that API internally uses uncommon features such as implicit classes and implicit conversions to avoid the exponential growth. Therefore, the technique used in ScaLALR is not portable to other languages such as Java and C++. On the other hand, the technique used in Silverchain is relatively simple and is portable to languages whose type system has generic classes.

EriLex [30] is a tool to generate a no-subchaining API from grammar by encoding an RDPDA into class definitions. The encoding method in EriLex is similar to the one in Silverchain and is portable to other languages. The drawback of EriLex is that input grammar needs to be LL(1) in Greibach normal form [13], which is a form that most of manually written grammars do not follow. Since the users of EriLex need to rewrite grammar into that form, EriLex cannot generate a subchaining style API for each nonterminal. Furthermore, there is no algorithm to rewrite grammar into the form required by EriLex as far as we know. Whether given grammar can be rewritten into that form is undecidable [26]. There are several tools known as fluent API generators such as CLARA [2], TS4J [3], and fluflu [29]. These tools, however, generate only subchaining style APIs.

The role of type parameters in Silverchain differs from ones in manually written fluent APIs such as AssertJ [6], j2html [20], and jOOQ [12]. In those manually written fluent APIs, type parameters are used to eliminate boilerplate code in classes for APIs. For example in AssertJ, type parameters



(a) The result in Java 9 EA



(b) The result in C++

```
class Exp<T, S> {
  Exp<Exp<T, S>, Exp<T, S>> m() { ... }
}

// 4-sized chain using the class "Exp"
new Exp<Object, Object>().m().m() ... .m().m();
```

(c) The Java class used in the experiment. We measured the compilation time for method chains of varying length using this class. The Java compiler we used is `javac` 9.

```
template <class T, class S> class Exp {
  Exp<Exp<T, S>, Exp<T, S> > *m() { ... }
};

// 4-sized chain using the class "Exp"
(new Exp<X, X>())->m()->m()->m()->m();
```

(d) The class used in the experiment for C++. We used `clang` 802.0.42 to compile method chains using this class.

Figure 13. The worst case behavior of the algorithm of Gil et al.

The experiment is performed on a machine with Intel Core i7 3.3 GHz processor and 16 GB memory.

are used to store the type of `this` in Java [23] and help the developers to implement methods that have the same signature but a different return type. On the other hand, Silverchain uses type parameters to express an infinite number of states that are produced from CFG. The algorithm of Gil et al. and EriLex also use type parameters in the same way.

6 Conclusion

We present Silverchain, which generates type-safe fluent APIs that support both subchaining and non-subchaining style in one fluent API. The input to Silverchain is the grammar of a generating fluent API. Since the DSL for the input grammar is similar to BNF, a *fluent EDSL* for an external DSL can be easily generated from the grammar of the DSL with Silverchain. However, for some grammars, Silverchain fails to generate a fluent API that fully supports non-subchaining style. First future work is to extend the range of grammars that Silverchain can generate non-subchaining style API for without exponential growth of compilation time.

Another future work is to improve the emulation of DSLs by using the mechanism in the host language. For example, DSL sentences are currently emulated only by method chaining, but they could be emulated in a better way by mapping a part of DSL's syntax to the similar syntax in a host general-purpose language (e.g. if-else syntax in a DSL to the same syntax in a host language). The properties of an emulated DSL other than syntactic rules could also be statically checked if the DSL's type system or name binding system is also mapped to host language mechanism.

References

[1] Rajeev Alur and Parthasarathy Madhusudan. 2004. Visibly Pushdown Languages. In *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing (STOC '04)*. ACM, New York, NY, USA, 202–211. https://doi.org/10.1145/1007352.1007390

[2] Eric Bodden. 2010. Efficient Hybrid Typestate Analysis by Determining Continuation-equivalent States. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 5–14. https://doi.org/10.1145/1806799.1806805

[3] Eric Bodden. 2014. TS4J: A Fluent Interface for Defining and Computing Typestate Analyses. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis (SOAP '14)*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/2614628.2614629

[4] Janusz Brzozowski. 1962. Canonical Regular Expressions and Minimal State Graphs for Definite Events. (1962), 529–561.

[5] John Cocke. 1969. *Programming Languages and Their Compilers: Preliminary Notes.* Courant Institute of Mathematical Sciences, New York University.

[6] Joel Costigliola. 2014. AssertJ / Fluent assertions for java. (December 2014). Retrieved September 4, 2017 from http://joel-costigliola.github.io/assertj/

[7] Bruno Courcelle. 1977. On Jump-Deterministic Pushdown Automata. *Mathematical systems theory* 11, 1 (Dec 1977), 87–109. https://doi.org/10.1007/BF01768471

[8] Joseph Darcy. 2016. JDK 9 Language, Tooling, and Library Features. (September 2016). Retrieved September 4, 2017 from https://cdn.app.compendium.com/uploads/user/e7c690e8-6ff9-102a-ac6d-e4aebca50425/f4a5b21d-66fa-4885-92bf-c4e81c06d916/File/e1950f4e52d2b112757b70cf28caa117/j1_2016_jdk9_lang_tools_libs.pdf

[9] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. 2011. SugarJ: Library-based Syntactic Language Extensibility. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*. ACM, New York, NY, USA, 391–406. https://doi.org/10.1145/2048066.2048099

[10] Martin Fowler. 2005. FluentInterface. (December 2005). Retrieved September 4, 2017 from https://www.martinfowler.com/bliki/FluentInterface.html

[11] Yossi Gil and Tomer Levy. 2016. Formal Language Recognition with the Java Type Checker. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.), Vol. 56. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 10:1–10:27. https://doi.org/10.4230/LIPIcs.ECOOP.2016.10

[12] Data Geekery GmbH. 2017. jOOQ: The easiest way to write SQL in Java. (September 2017). Retrieved September 4, 2017 from https://www.jooq.org/

[13] Sheila Greibach. 1965. A New Normal-Form Theorem for Context-Free Phrase Structure Grammars. *J. ACM* 12, 1 (Jan. 1965), 42–52.

https://doi.org/10.1145/321250.321254

[14] Radu Grigore. 2017. Java Generics Are Turing Complete. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 73–85. https://doi.org/10.1145/3009837.3009871

[15] Hamcrest. 2012. Hamcrest. (August 2012). Retrieved September 4, 2017 from http://hamcrest.org/

[16] Michael Harrison and Ivan Havel. 1972. Real-Time Strict Deterministic Languages. *SIAM J. Comput.* 1, 4 (1972), 333–349. https://doi.org/10.1137/0201024

[17] Paul Hudak. 1996. Building Domain-specific Embedded Languages. *ACM Comput. Surv.* 28, 4es, Article 196 (Dec. 1996). https://doi.org/10.1145/242224.242477

[18] Kazuhiro Ichikawa. 2016. phenan/scalalr: ScaLALR : LALR parser generator for embedded DSLs in Scala. (December 2016). Retrieved September 4, 2017 from https://github.com/phenan/scalalr

[19] Kazuhiro Ichikawa and Shigeru Chiba. 2017. User-Defined Operators Including Name Binding for New Language Constructs. *The Art, Science, and Engineering of Programming* 1, 2, Article 15 (2017).

[20] j2html. 2017. Fast and fluent Java HTML5 builder - Java HTML builder. (September 2017). Retrieved September 4, 2017 from https://j2html.com/

[21] jMock. 2012. jMock - An Expressive Mock Object Library for Java. (December 2012). Retrieved September 4, 2017 from http://www.jmock.org/

[22] Tadao Kasami. 1965. *An Efficient Recognition and Syntax-Analysis Algorithm for Context-Free Languages.* Technical Report. DTIC Document.

[23] Ansgar Konermann. 2010. Emulating "self types" using Java Generics to simplify fluent API implementation. (November 2010). Retrieved September 4, 2017 from http://web.archive.org/web/20130721224442/http://passion.forco.de/content/emulating-self-types-using-java-generics-simplify-fluent-api-implementation

[24] Tomer Levy. 2017. *A Fluent API for Automatic Generation of Fluent APIs in Java.* Ph.D. Dissertation. Israel Institute of Technology.

[25] Vicente Romero and Maurizio Cimadamore. 2016. JEP 215: Tiered Attribution for javac. (July 2016). Retrieved September 4, 2017 from http://openjdk.java.net/jeps/215

[26] Daniel Rosenkrantz and Richard Stearns. 1969. Properties of Deterministic Top Down Grammars. In *Proceedings of the First Annual ACM Symposium on Theory of Computing (STOC '69)*. ACM, New York, NY, USA, 165–180. https://doi.org/10.1145/800169.805431

[27] Robert Strom and Shaula Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering* SE-12, 1 (Jan 1986), 157–171. https://doi.org/10.1109/TSE.1986.6312929

[28] Ken Thompson. 1968. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM* 11, 6 (June 1968), 419–422. https://doi.org/10.1145/363347.363387

[29] Peter Verhas. 2014. verhas/fluflu: Fluent Api Creator. (July 2014). Retrieved September 4, 2017 from https://github.com/verhas/fluflu

[30] Hao Xu. 2010. *EriLex: An Embedded Domain Specific Language Generator.* Springer Berlin Heidelberg, Berlin, Heidelberg. 192–212 pages. https://doi.org/10.1007/978-3-642-13953-6_11

[31] Daniel Younger. 1967. Recognition and parsing of context-free languages in time n3. *Information and Control* 10, 2 (1967), 189 – 208. https://doi.org/10.1016/S0019-9958(67)80007-X