Fling – A Fluent API Generator (Artifact)

Ori Roth
Technion I.I.T Computer Science Dept., Haifa, Israel
ori.rothh@gmail.com

Yossi Gil
Technion I.I.T Computer Science Dept., Haifa, Israel
yogi@cs.technion.ac.il

Abstract

The first general and practical solution of the fluent API problem is presented. We give an algorithm that given a deterministic context free language (equivalently, LR(k), $k \geq 0$ language) encodes it in an unbounded parametric polymorphism type system employing only a polynomial number of types. The theoretical result is employed in an actual tool FLING– a fluent API compiler-compiler in the style of YACC, tailored for embedding DSLs in JAVA.

2012 ACM Subject Classification Software and its engineering → General programming languages; Software and its engineering → Domain specific languages

Keywords and phrases Fluent API, compilation, generics, code generation

Digital Object Identifier 10.4230/DARTS.5.2.12

Funding *Ori Roth*: Technion I.I.T
Yossi Gil: Technion I.I.T

Related Article Yossi Gil and Ori Roth, "Fling - A Fluent API Generator", in 33rd European Conference on Object-Oriented Programming (ECOOP 2019), LIPIcs, Vol. 134, pp. 13:1–13:25, 2019.
https://dx.doi.org/10.4230/LIPIcs.ECOOP.2019.13

Related Conference 33rd European Conference on Object-Oriented Programming (ECOOP 2019), July 15–19, 2019, London, United Kingdom

1 Scope

Fluent APIs are often used a implement a domain specific language (DSL). In List. 1 we see an example of fluent API that specifies a regular expression.

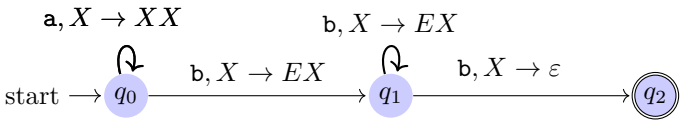
Listing 1 Fluent API specification of the regular expression $(ab?)^* \mid \left([0-9][0-9]^+\right)$.

```
1  re().// this starts the fluent API chain here
2  noneOrMore(exactly("a").and().option(exactly("b"))).//
3  or().oneOrMore(anyDigit()).//
4  $(); // End the chain
```

In the main ECOOP'19 article, we argued that an efficient and modestly sized fluent API can be generated for any *deterministic language* (languages accepted by a deterministic pushdown automaton), or equivalently, all LR(k) languages. Our theoretical result was supported by a proof and an implementation. Here we describe on the implementation and its derivatives; we contribute the code behind the algorithm, and demonstrate that the result is not merely theoretical, but can applied for embedding a DSL as a fluent API in programming languages with only rudimentary support of genericity.

FLING (**FL** uent **I**nterfaces **G**enerator) is a fluent API compiler-compiler (in the venue of YACC) that does this and more. It accepts a specification of a formal language, e.g., the language of regular expressions, and compiles this definition into a set of classes and methods that realize a fluent API for the given language. The formal language may specified in one of two ways:

Figure 1 A DPDA for the language $a^n b^n$, $n > 0$.



1. *DPDA* (**D**eterministic **P**ush**D**own **A**utomaton) including a definition of the set of stack symbols, automaton symbols, start symbol, transition function, and accepting symbols.
   Consider for example, a DPDA for the language $a^n b^n$, $n > 0$ in Fig. 1.
   The automaton in the figure uses two stack symbols: E for denoting the stack's bottom, and X, serving as a marker on the stack for each a character that does not found a b that matches it. List. 2 shows how this automaton is specified in FLING.

   Listing 2 Fling specification, in a fluent API fashion, of the DPDA of Fig. 1.

```
enum Σ implements Terminal {a, b } // Define the alphabet of input characters.
enum Γ implements Named { E, X }// Define the alphabet of stack symbols.
public static final DPDA<Named, Verb, Named> dpda =
  Grammar.cast(dpda(Q.class, Σ.class, Γ.class) // Boiler plate code
    .q0(q0) // Define the start state
    .F(q2) // Define the accepting state
    .γ0(E) // Define the initial stack symbol
    .δ(q0, a, E, q0, E, X) // Transition for pushing the first $X$
    .δ(q0, a, X, q0, X, X) // Transition for pushing additional $X$s
    .δ(q0, b, X, q1) // Transition for consuming first $b$s
    .δ(q1, b, X, q1) // Transition for consuming subsequent $b$s
    .δ(q1, ε(), E, q2) // Finish when bottom of the stack is reached
    .go()); // Let the builder of the DPDA start its work.
```

   Once the automaton is built, we can request FLING to generate a fluent API for it. List. 3 shows how this is done.

   Listing 3 Using FLING to create a fluent API for JAVA and C++.

```
public static final String JavaFluentAPI =
  new JavaAPIAdapter(
    "fling.examples.generated", // Package name
    "AnBn", // The class name.
      "$", // The terminating symbol
    new NaiveNamer("fling.examples.generated", "AnBn")) //
       .printFluentAPI(new ReliableAPICompiler(dpda).compileFluentAPI());
public static final String CppFluentAPI = new CppAPIAdapter(
  "$",
  new NaiveNamer("AnBn")) //
    .printFluentAPI(new ReliableAPICompiler(dpda).compileFluentAPI());
```

   (This and other examples can be found in `fling.examples.automata`.)
2. *LL(1) EBNF Grammar*, including the sets of terminals and non-terminals, the start symbol and derivation rules. (See examples in `fling.examples.languages` and the walk-through below.)

Use cases of the generated fluent API are found in `fling.examples.usecases`.

Incidentally, the specification of the input formal language, be it as DPDA or EBNF, is done done using fluent API. Moreover, these two fluent APIs that FLING offers to its clients were generated by FLING itself.

In the case that language is specified using a grammar, FLING also generates code to create the AST of the parsed input. In particular, FLING defines classes that correspond to non-terminal symbols of the grammar specification, with class containment and inheritance relationships in accordance with the derivation rules of the grammar. The fluent API that FLING generates in this case is such that a fluent API call chain returns an instance of the class corresponding to the start symbol of the grammar. This instance represents the AST of the particular chain that returned it.

FLING can generate fluent API code for different programming languages. The current implementation may produce Java or C++ code (DPDA only). FLING may be extended by language adapters (see package `fling.adapters` for examples) to support other programming languages. The target language must support unbounded polymorphism of classes, i.e., classes that may receive type parameters, but it is not required that the language allows constraints on these parameters, as found in e.g., ML.

For example, let us use FLING to create a JAVA [1] fluent API of DATALOG [2], and then use it to define DATALOG programs and run them. A definition of a DSL by its grammar begins with definitions of the set of grammar terminals and non-terminas, as shown in List. 4 (drawn from file `Datalog.java` in the code).

Listing 4 Terminals and non-terminals for DATALOG grammar.

```
/** Set of terminals, i.e., method names of generated fluent API. */
public enum Σ implements Terminal {
  infer, fact, query, of, and, when, always, v, l
}

/**
 * Set of non-terminals, i.e., abstract concepts of fluent API; these names
 * will be translated into names of classes of abstract syntax tree that \Fling
 * generates, i.e., this AST will have class {@link Program} which will have a
 * list of {@link Statement}, etc.
 */
public enum V implements Variable {
  Program, Statement, Rule, Query, Fact, Bodyless, WithBody,
  RuleHead, RuleBody, FirstClause, AdditionalClause, Term
}
```

The language grammar is then defined by List. 5.

Listing 5 DATALOG BNF.

```
/** Datalog's grammar in Backus-Naur form. */
public static final BNF bnf = bnf(). //
  start(Program). // This is the start symbol
  derive(Program).to(oneOrMore(Statement)). // Program ::= Statement*
  specialize(Statement).into(Fact, Rule, Query).
  /* Defines the rule Statement ::= Fact |Rule | Query, but also defines
   * that classes {@link Fact}, {@link Rule} and {@link Query} extend class
   * {@link Statement} */
  derive(Fact).to(fact.with(S), of.many(S)). // Fact ::= fact(S*) of(S*)
  derive(Query).to(query.with(S), of.many(Term)). //
  specialize(Rule).into(Bodyless, WithBody). //
  derive(Bodyless).to(always.with(S), of.many(Term)). //
  derive(WithBody).to(RuleHead, RuleBody). //
  derive(RuleHead).to(infer.with(S), of.many(Term)). //
  derive(RuleBody).to(FirstClause, noneOrMore(AdditionalClause)). //
  derive(FirstClause).to(when.with(S), of.many(Term)). //
  derive(AdditionalClause).to(and.with(S), of.many(Term)). //
  derive(Term).to(l.with(S)).or(v.with(S)). //
  build();
```

To complete the definition, we define `S` as an alias to the class String as in List. 6

Listing 6 A definition of `S` is required for List. 5.

```
/**
 * Short name of {@link String}.class, used to specify the type of parameters
 * to fluent API methods in grammar specification.
 */
private static final Class<String> S = String.class;
```

Now, DATALOG language specification, given in Backus-Naur form (BNF), is embedded in the JAVA program shown in List. 5.

Listing 7 DATALOG program.

```
parent(john, bob).
parent(bob, donald).
ancestor(A, B) :- parent(A, B).
ancestor(A, B) :- parent(A, C), ancestor(C, B).
ancestor(john, X)?
```

Listing 8 Embedded DATALOG program.

```
Program program =
  fact("parent").of("john", "bob").
  fact("parent").of("bob", "donald").
  infer("ancestor").of(v("A"), v("B")).
    when("parent").of(v("A"), v("B")).
  infer("ancestor").of(v("A"), v("B")).
    when("parent").of(v("A"), v("C")).
    and("ancestor").of(v("C"), v("B")).
  query("ancestor").of(l("john"), v("X")).$();
```

After calling FLING with the DATALOG BNF as its input, we can use the generated fluent API to define DATALOG programs. The native DATALOG program shown in List. 7 can now be embedded in JAVA as a chain of method calls in fluent API style, as shown in List. 8.

This chain yields a `Program` object representing the abstract syntax tree (AST) of the program, later to be traversed by the client library, executing the program and printing its result.

The supplied JAVA code runs the embedded DATALOG program by visiting the said AST.

Listing 9 Visiting a Datalog program using an AST visitor.

```
    new DatalogRunner().visit(program);
```

The `DatalogRunner` class is implemented as a standard visitor, with dedicated methods for each node type. Here is an excerpt of the code

Listing 10 An AST visitor running a Datalog program using Jatalog.

```
public static class DatalogRunner extends DatalogAST.Visitor {
  final Jatalog j = new Jatalog(); // Use Jatalog, a \Datalog engine

  @Override public void whileVisiting(final Fact fact) throws DatalogException {
    j.fact(fact.fact, fact.of);
    print(fact);
  }
  @Override public void whileVisiting(final WithBody withBody) throws Exception {
    j.rule(Expr.expr(withBody.ruleHead.infer, toStrings(withBody.ruleHead.of)), //
        getExprRightHandSide(withBody));
    print(withBody);
  }
  //...
}
```

2 Content

The key packages and classes in FLING are:

- **Language input**
  - `fling.BNF`: Language specification in Backus-Naur form. To create a `BNF` one may use the fluent API in `fling.grammars.api`[1].
  - `fling.DPDA`: Language specification in the form of DPDA.

[1] this fluent API was originally generated by FLING

- `fling.internal.grammar.Grammar`: base class for a grammar family. Such family declares its method to convert a BNF to DPDA. Currently, only $LL(1)$ grammars are supported.

- **API compilation**
  - `fling.internal.compiler.api.APICompiler`: compiles DPDA into a fluent API. The fluent API computed in an internal abstract form. This internal form can be then be translated to the target language. There are two compilers available:
    - `fling.compilers.api.PolynomialAPICompiler`: generates a polynomial number of types, but an illegal API method call returns "bottom", empty type instead of raising a compilation exception[2].
    - `fling.compilers.api.RelizableAPICompiler`: generates a (possibly) exponential number of types, yet all illegal API method calls raise a compilation exception.
- **AST compilation**
  - `fling.compilers.ast.ASTCompiler`: compiles grammar into types representing its AST nodes. The AST types class is given as an AST, to be translated to a program of the target language.
- **Language adaption**
  - `fling.adapters`: contains target language adapters.
    - `fling.adapters.JavaMediator`: integrate the generated API and AST to a Java program. Its input must be an $LL(1)$ grammar.
    - `fling.adapters.CppAPIAdapter`: compiles API to C++ [3] code. Its input must be a DPDA. The generated C++ fluent API supports method chains of the form $a() \to b().c().\ldots.\$()$[3].
    - `fling.internal.compiler.Namer`: responsible of naming entities in the generate code. Currently there is only one namer available, `fling.namers.NaiveNamer`.
- **Testing**
  - `fling.examples.automata`: language examples given as DPDA
  - `fling.examples.languages`: language examples given as BNF
  - `fling.examples.usecases`: examples of using the language examples. These use cases refer to code generated by Fling, located in `fling.examples.generated`. To activate the examples run `fling.examples.ExamplesMainRunMeFirst`.

2.1 Walk-through

We now demonstrate the common use-case of Fling, by defining a grammar, generating a fluent API of its language and analyzing the ASTs created at run-time. Let us create a fluent API for arithmetic expressions supporting addition and multiplication, containing words such as "`2 * (3 + 4)`".

1. **Choose a grammar for your language:** A possible grammar for the language of arithmetic expression is defined as follows:

```
E -> E + T | T
T -> T * F | F
F -> (E) | int
```

   Fling currently supports only $LL(1)$ grammars, so this grammar should be rewritten to fit the $LL(1)$ class[4]:

[2] this is an implementation of the API generation algorithm discussed in the paper
[4] exlanation about $LL(1)$ grammars is given here: https://andrewbegel.com/cs164/ll1.html

```
E -> T E'
E' -> + T E' | ε
T -> F T'
T' -> * F T' | ε
F -> (E) | int
```

2. **Encode the grammar in Java:** Write your grammar in a JAVA class using FLING's API. First define the terminal and non-terminal symbols of your language in their respecting `enum`s: Keep in mind you are limitted by JAVA's naming rules, so replacements might be needed.

```
import fling.*;
...
public enum Σ implements Terminal {
  plus, mult, begin, end, i
}
public enum V implements Variable {
  E, E_, T, T_, F
}
```

Next encode the grammar rules, given in BNF, using FLING's fluent API:

- Declare the start symbol
- Compose at least one derivation rule for each non-terminal
- The derivation rules may make use of extended BNF notations
- You may specialize terminals with parameter declarations: These parameter will later be accepted by the fluent API methods[5].

```
import fling.BNF;
import static fling.grammars.api.BNFAPI.bnf;
...
public static final BNF bnf = bnf(). // Start defining BNF
  start(E). // Declare the start symbol
  derive(E).to(T, E_). // E ::= T E'
  derive(E_).to(plus, T, E_).orNone(). // E' ::= + T E' | ε
  derive(T).to(F, T_). // T ::= F T'
  derive(T_).to(mult, F, T_).orNone(). // T' ::= * F T' | ε
  derive(F).to(begin, E, end).or(i.with(Integer.class)). // F ::= (E) | int
  // The terminal i representing a number is specialized with a parameter of type Integer
  build(); // Yield BNF
```

3. **Generate the fluent API**: Use a `JavaModerator` to create the fluent API classes. The moderator links together the various compilers offered by FLING, making the process handleable. The contents of the fluent API classes are computed upon initialing the moderator, and are available each in a class field: Thus writting the classes into the corresponding files is made easy.

```
import fling.adapters.JavaMediator;
...
JavaMediator jm = new JavaMediator(bnf, // The grammar defined above
  "fling.examples.generated", // Output package name
  "SimpleArithmetic", // Output base class name
  Σ.class); // Class of terminal symbols
// Write classes to filesystem:
writeToFile("SimpleArithmetic", jm.apiClass);
writeToFile("SimpleArithmeticAST", jm.astClass);
writeToFile("SimpleArithmeticCompiler", jm.astCompilerClass);
```

Here, class `SimpleArithmetic` contains the fluent API interfaces declarations. The declarations used for the AST generated at runtime are in `SimpleArithmeticAST`. The $LL(1)$ run-time compiler used to create the ASTs of fluent API method chains is found in class `SimpleArithmeticCompiler`

4. **Use your fluent API:** After saving the generated code into `.java` files and compiling this, the fluent API is ready for use. The class containing the fluent API declarations, `SimpleArithmetic`, contains a static method for each terminal which with which an arithmetic expression can start. In the present case, these are `begin` and `i`. A fluent API call chain

[5] A specialized terminal counts as a *new terminal*, in respect to the grammar rules

then starts with `SimpleArithmetic.begin()` (or `SimpleArithmetic.i(···)`) and ends with a `.$()`. The value returned by the chain is the AST of the declared arithmetic expression. Types of AST nodes are either non-abstract class or an interface:

- A class represents an "is-a" derivation rule of the form $X ::= AbcD$, containing a field for each of its non-vacuous descendants.
- An interface represents an "is-either" derivation rule of the form $X ::= A|B|C|D$, implemented by two or more interfaces and classes.

Class `SimpleArithmetic` in package `fling.examples.languages` defines the grammar in a fluent API style

```
package fling.examples.languages;
// Imports omitted for brevity
public class SimpleArithmetic {
  // Terminal symbols:
  public enum Σ implements Terminal {plus, mult, begin, end, i}
  // Non−terminal symbols:
  public enum V implements Variable {E, E_, T, T_, F}
  public static final BNF bnf = bnf(). // Start defining BNF
    start(E). // Declare the start symbol
    derive(E).to(T, E_). // E ::= T E'
    derive(E_).to(plus, T, E_).orNone(). // E' ::= + T E' | ε
    derive(T).to(F, T_). // T ::= F T'
    derive(T_).to(mult, F, T_).orNone(). // T' ::= ∗ F T' | ε
    derive(F).to(begin, E, end).or(i.with(Integer.class)). // F ::= (E) | int
    build(); // Yield BNF
  public static void main(String[] args) {
    JavaMediator jm = new JavaMediator(bnf, // Grammar definition
      "fling.examples.generated", // Output package name
      "SimpleArithmetic", // Output base class name
      Σ.class); // Class of terminal symbols
    writeToFile("SimpleArithmetic", jm.apiClass);
    writeToFile("SimpleArithmeticAST", jm.astClass);
    writeToFile("SimpleArithmeticCompiler", jm.astCompilerClass);
  }
  private static void writeToFile(String fileName, String fileContent) {
    // Code for formatting Java code and saving it omitted for brevity
  }
}
```

Compiling and running this class will generate three files:

1. `./src/test/java/fling/examples/generated/SimpleArithmetic.java`
2. `./src/test/java/fling/examples/generated/SimpleArithmeticAST.java`
3. `./src/test/java/fling/examples/generated/SimpleArithmeticCompiler`

You must compile and run these files to be able to test run the fluent API. Within Eclipse, this means refreshing the files, by hitting F5. Here is a simple use case which puts the generated code to use:

```
package fling.examples.usecases;
import static fling.examples.generated.SimpleArithmetic.i;
import fling.examples.generated.SimpleArithmeticAST.*;
public class SimpleArithmeticUseCase {
  public static void main(String[] args) {
    // A method calls chain declaring the arithmetic expression 2+(3∗4)
    E e = i(2).mult().begin().i(3).plus().i(4).end().$();
    System.out.println("2 * (3 + 4) = " + evaluate(e)); // Output is "2 ∗ (3 + 4) = 14"
  }
  private static int evaluate(E e)   { return evaluate(e.t) + evaluate(e.e_); }
  private static int evaluate(E_ e)  { return e instanceof E_1 ? evaluate((E_1) e) : evaluate((E_2) e); }
  private static int evaluate(E_1 e) { return evaluate(e.t) + evaluate(e.e_); } // E ::= T E'
  private static int evaluate(E_2 e) { return 0; } // E ::= ε
  private static int evaluate(T t)   { return evaluate(t.f) * evaluate(t.t_); }
  private static int evaluate(T_ t)  { return t instanceof T_1 ? evaluate((T_1) t) : evaluate((T_2) t); }
  private static int evaluate(T_1 t) { return evaluate(t.f) * evaluate(t.t_); } // T ::= F T'
  private static int evaluate(T_2 t) { return 1; } // T ::= ε
  private static int evaluate(F f)   { return f instanceof F1 ? evaluate((F1) f) : evaluate((F2) f); }
  private static int evaluate(F1 f)  { return evaluate(f.e); } // F ::= (E)
  private static int evaluate(F2 f)  { return f.i; } // F ::= int
}
```

2.2 Troubleshooting

Error handling in FLING is minimal. Some exceptions with useful error message do occur. Note

- While standard compilers provide suggestions to fixing improperly designed grammars, FLING does not offer such a service (although it could). If you get an exception while building the BNF, make sure it adheres the $LL(1)$ grammar class rules before proceeding. Sending a proper $LL(1)$ grammar to a `JavaMediator` should generate the fluent API classes successfully.
- Put in mind a code using the fluent API will not compile until it has been generated. While Eclipse support partial compilation, other IDEs might encounter problems while trying to compile, for instance, the above example in a single pass.
- Project FLING has a few external dependencies. Before using the project, make sure to build it using Maven. An optional compilation script is given below:

```
#!/bin/bash
# Clone repository
git clone https://github.com/OriRoth/fling.git
cd fling
# Compile without fling.examples.usecases (will produce errors)
mvn test-compile
# Generate example fluent-APIs
mvn -Dtest=fling.examples.ExamplesMainRunMeFirst surefire:test
# Compile with fling.examples.usecases (without errors)
mvn test-compile
# Copy dependencies to local directory
mvn dependency:copy-dependencies
# Run "SimpleArithmetic" use-case example
java -Dfile.encoding=UTF8 -cp target/classes/:target/test-classes/:target/dependency/* \
fling.examples.usecases.SimpleArithmeticUseCase
# Should output "2 * (3 + 4) = 14"
```

  The script above clones the project from the repository, compiles it in two passes and runs the use case above class `SimpleArithmeticUseCase` class depicted above.
- The project files should be UTF-8 encoded, or otherwise will produce errors.

3 Getting the artifact

The artifact endorsed by the Artifact Evaluation Committee is available free of charge on the Dagstuhl Research Online Publication Server (DROPS). In addition, the artifact is also available at `https://github.com/OriRoth/fling`. Clone this repository and open it with Eclipse to gain access to the sources and examples. Be sure to compile and run as JAVA application the file named `ExamplesMainRunMeFirst` to generate the classes used by the tests.

4 Tested platforms

Product was tested on Windows and Linux under the Eclipse Orion environment. Requires JAVA 1.8 or above to compile.

5 License

The artifact is available under MIT license.

6 MD5 sum of the artifact

5064eb01b5f238cd081fc9c63e604cbf

7 Size of the artifact

1,113,854 Bytes

References

1 Ken Arnold and James Gosling. *The* Java *Programming Language*. Addison Wesley, 1996.

2 Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic programming and databases*. SVNY, 1990.

3 Stroustrup. *The C++ Programming Language*. AW, awad, third edition, 1997.