Fajita

A Fluent API
for Automatic Generation
of Fluent APIs
in Java

Research Thesis

Submitted in Partial Fulfillment of the Requirements for the
Degree of Master of Science in Computer Science

Tomer Levy

Submitted to the Senate of the Technion - Israel Institute of Technology
Nisan, 5776, Haifa, April, 2016

Acknowledgements

This Research Thesis was Done under the Supervision of Prof. Joseph Gil in the Faculty of Computer Science

First of all, I would like to thank Tamar, for the infinite patience and understanding, and her unconditional love and support during this insane semester. I could not have done it without you.

In addition, I want to thank my family and close friends ; even if you don't feel that way, you keep me sane, fresh and ready for more.

I would also like to thank my friend Iddo Zmiry, whom with I always have some tasteful argument regarding this and that, spicing up my days.

And last, but not least, I want to express my deepest gratitude to my advisor, Prof. Yossi (Joseph) Gil, for his expert guidance, extraordinary commitment and warm hospitality. He taught me much more then just science, on topics variating from Torah and English literature to "legal buccaneer"s.

The Generous Financial Help of the Technion is Gratefully Acknowledged

Contents

List of Tables

List of Figures

List of Algorithms

Listings

Abstract

The main result of this thesis is Fajita, a practical prototypical tool for the automatic generation of Java *Fluent APIs from their specification.*

We begin with a theoretical study, that explains why the problem's core lies with the expressive power of Java *generics. We show that automatic generation is possible whenever the specification is an instance of the set of deterministic context-free languages, a set which contains most "practical" languages.*

We then advances to present, for the first, an efficient (specifically linear time) algorithm for generating an automaton, implementable within the framework of compile time computation of Java*, which recognizes a given LL(1) language. The generated automaton is time efficient, spending a constant amount of time on each symbol of the "input". Space requirement is also polynomially bounded.*

Other contributions include a collection of techniques and idioms of the limited meta-programming possible with Java *generics, and an empirical measurement demonstrating that the runtime of the "javac" compiler of* Java *may be exponential in the program's length, even for programs composed of a handful of lines and which do not rely on overly complex use of generics. Another theoretical contribution of this work is the RLLp, an automaton for recognizing LL(1) grammars that supplies the realtime property.*

List of Names and Symbol

L	Formal language
τ	Java type
Σ	Alphabet set
σ	Letter from the alphabet
Ξ	Nonterminals set (or variables)
ξ	Nonterminal (or variable)
G	Grammar
P	Productions set
r	Production rule
α	String of terminals or nonterminals
β	String of terminals or nonterminals
Q	States set
q	state
Γ	Stack elements set
γ	stack element
ζ	String of stack elements
δ	Transition function (or prediction function)
Δ	Prediction function
ϵ	Empty string
\$	End-of-input symbol
$\perp$	Undefined result
$\lightning$	error state

List of Abbreviations

Fajita	Fluent API for Java Inspired by Theory of Automata
API	Application Programming Interface
DSL	Domain Specific Language
BNF	Backus–Naur Form
DFA	Deterministic Finite Automata
LL	Left to right, Leftmost derivation
LL	Left to right, Righmost derivation
LLp	LL parser
RLLp	Realtime LL parser
DPDA	Deterministic PushDown Automata
NPDA	Nondeterministic PushDown Automaton
CFG	Context-Free Grammar
DCFG	Deterministic Context-Free Grammar
jDPDA	Single-Jump Single-State Realtime Deterministic PushDown Automton

Chapter 1

Introduction

1.1 Fluent APIs Java

Ever after their inception[1] *fluent APIs* increasingly gain popularity [20, 34, 38] and research interest [15, 35]. In many ways, fluent APIs are a kind of *internal* ***D****omain* ***S****pecific* ***L****anguages* [19, 32, 48]: They make it possible to enrich a host programming language without changing it. Advantages are many: base language tools (compiler, debugger, IDE, etc.) remain applicable, programmers are saved the trouble of learning a new syntax, etc. However, these advantages come at the cost of expressive power; in the words of Fowler: "*Internal DSLs are limited by the syntax and structure of your base language.*"[2]. Indeed, in languages such as C++ [47], fluent APIs often make extensive use of operator overloading (examine, e.g., Ara-Rat [23]), but this capability is not available in Java [3].

Despite this limitation, fluent APIs in Java can be rich, expressive, and in many cases can boost productivity in elegant and neat code snippets such as

```
from("direct:a").choice()
    .when(header("foo").isEqualTo("bar"))
      .to("direct:b")
    .when(header("foo").isEqualTo("cheese"))
      .to("direct:c")
    .otherwise()
      .to("direct:d");
```

(a use case of Apache Camel [33], open-source integration framework), and,

[1] http://martinfowler.com/bliki/FluentInterface.html

[2] M. Fowler, *Language Workbenches: The Killer-App for Domain Specific Languages?*, 2005 http://www.martinfowler.com/articles/languageWorkbench.html

```
create
  .select(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME, count())
  .from(AUTHOR)
  .join(BOOK).on(AUTHOR.ID.equal(BOOK.AUTHOR_ID))
  .where(BOOK.LANGUAGE.eq("DE"))
  .and(BOOK.PUBLISHED.gt(date("2008-01-01")))
  .groupBy(AUTHOR.FIRST_NAME, AUTHOR.LAST_NAME)
  .having(count().gt(5))
  .orderBy(AUTHOR.LAST_NAME.asc().nullsFirst())
  .limit(2)
  .offset(1);
```

(a use case of jOOQ[3], a framework for writing SQL like code in Java, much like LINQ project [42] in the context of C# [30]).

Other examples of fluent APIs in Java are abundant: jMock [20], Hamcrest[4], EasyMock[5], jOOR[6], jRTF[7] and many more.

The actual implementation of these many examples is traditionally not carried out in the neat manner it could possibly take. Our reason for saying this is that the fundamental problem in fluent API design is the decision on the "language". This language is the precise definition of which sequences of method applications are legal and which are not.

As it turns out, the question of whether a BNF definition of such a language can be "compiled" automatically into a fluent API implementation is, a question of the computational power of the underlying language. In Java, the problem is particularly interesting since in Java (unlike e.g., C++ [28]), the type system is (probably) not Turing complete.

The question is then, what can be computed, and what can not be computed by coercing the type system and the type checker of a certain programming language to do abstract computations it was never meant to carry out? And, why should we care?

The previous jOOQ example suggests that jOOQ imitates SQL, but, is it possible at all to produce a fluent API for the entire SQL language, or XPath, HTML, regular expressions, BNFs, EBNFs, etc.?

Of course, with no operator overloading it is impossible to fully emulate tokens; method names though make a good substitute for tokens, as done in the Apache Camel code excerpt above

```
.when(header(foo).isEqualTo("bar")).
```

.

The questions that motivate this research are:

- Given a BNF specification of a DSL, determine whether there exists a fluent API in Java that can be made for this specification?
- In the cases that such fluent API is possible, can it be produced automatically?

[3] http://www.jooq.org
[4] http://hamcrest.org/JavaHamcrest/
[5] http://easymock.org/
[6] https://github.com/jOOQ/jOOR
[7] https://github.com/ullenboom/jrtf

- Is it feasible to produce a *compiler-compiler* such as Bison [16] to convert such language specification into a fluent API?

Inspired by the theory of formal languages and automata, this study explores what can be done with fluent APIs in JAVA.

1.2 A Brief Review of Fajita - our Main Goal

Consider Figure 1.1, a BNF specification of a fluent API for a certain fragment of SQL.

Figure 1.1 A BNF for a fragment of SQL select queries.

$$
\begin{aligned}
\langle \mathit{Query} \rangle &::= \text{select()}\ \langle \mathit{Quant} \rangle\ \ \text{from(Table.class)}\ \langle \mathit{Where} \rangle \\
\langle \mathit{Quant} \rangle &::= \text{all()} \\
&\ \ |\ \text{columns(Column[].class)} \\
\langle \mathit{Where} \rangle &::= \text{where()}\ \ \text{column(Column.class)}\ \langle \mathit{Operator} \rangle \\
&\ \ |\ \epsilon \\
\langle \mathit{Operator} \rangle &::= \text{equals(Expr.class)} \\
&\ \ |\ \text{greaterThan(Expr.class)} \\
&\ \ |\ \text{lowerThan(Expr.class)}
\end{aligned}
$$

To create a JAVA implementation that realizes this fluent API, the designer feeds the grammar to FAJITA, as in Figure 1.2.

Figure 1.2 A JAVA code excerpt defining the BNF specification of the fragment SQL language defined in Figure 1.1.

```
new Fajita(SQLTerminals.class, SQLNonterminals.class)
  .start(Query)
  .derive(Query).to(select)
                .and(Quant).and(from,Table.class).and(Where)
  .derive(Quant).to(all)
                .or(columns,Column[].class)
  .derive(Where).to(where)
                .and(column,Column.class).and(Operator)
                .orNone()
  .derive(Operator).to(equals,Expr.class)
                   .or(greaterThan,Expr.class)
                   .or(lowerThan,Expr.class)
  .go();
```

We see that FAJITA's API is fluent in itself, and the call chain in Figure 1.2, is structured almost exactly as in derivation rules in Figure 1.1. In particular, the code in Figure 1.2 shows how fluent API specification in FAJITA may include parameterless methods (**`select()`**, **`all()`** and **`where`**) as well as methods which take parameters, e.g., method **`column`** taking parameter of type **`Column`** and method **`from`** taking a **`Table`** parameter.

Other than the derivation rules, FAJITA needs to be told the start rule and the sets of terminals and nonterminals. The start symbol is given in the **start** method that receives a nonterminal, and the symbol sets are specified in the first method call in the chain where, the enumerate types **SQLTerminals** and **SQLNonTerminals** are:

```
enum SQLTerminals implements Terminal{
  select,from,all,columns,
  where,column,equals,greaterThan,lowerThan;
}

enum SQLNonterminals implements NonTerminal{
  Query,Quant,Where,Operator;
}
```

The call **.go()**, occurring last in the chain, makes FAJITA generate types and methods realizing the fluent API, in such a way that legal use of the API like in Figure 1.3 is syntactically legal,

Figure 1.3 Legal sequences of calls in the sql fragment example, where **t** is of type **Table**, **c** is of type **Column** and **e** is of type **Expr**

```
new Query().select().all().from(t).$();
new Query().select().all().from(t)
    .where().column(c).equals(e).$();
```

while snippets disobeying the BNF specification in Figure 1.1 like Figure 1.4 , do not type check.

Figure 1.4 Illegal sequences of calls in the sql fragment example, where **t** is of type **Table** and **c** is of type **Column**.

```
new Query().select().select().from(t).$();
new Query().select().all().from(t).where().column(c).$();
```

1.3 The many trials of doing it in Java

Fluent APIs are neat, but design is complicated, involving theory of automata, type theory, language design, etc.

And still, automatic generation of these exist to some extent. One example, is fluflu[8]: a software artifact that uses JAVA annotations to define deterministic finite automata (henceforth, DFA), and then compiles it to a fluent API based on this automaton. Use example of fluflu is depicted at Figure 1.5.

The code excerpt in Figure 1.5 defines a single stated DFA using fluflu. The state is both initial and accepting, with a single self transition, labeled with terminal a. The regular language realized by the automaton is $L = a^*$.

Although fluent API generation was "achieved", this example has many flaws, some of them are:

[8] https://github.com/verhas/fluflu

Figure 1.5 Use example of fluflu, generation a fluent API for the regular expression a^*

```
@Fluentize(className="L" , startState="Q0" , startMethod="a")
public abstract class ToBeFluentized {
  @Transition(from = "Q0", to = "Q0") public void a() { }
}
```

1. Using fluflu is complicated and the resulted code is messy.
2. Defining a DFA is harder then writing, for example, the equivalent regular expression (and in some cases, the size of the DFA is exponentially bigger)
3. Languages defined by DFA can only be regular, a rather small class of languages.

A question rises after seeing FAJITA and fluflu: how come the representation of simple languages such as regular languages is so complex (as seen in Figure 1.5) and the representation of a more complex class of languages, is rather simple (as seen Figure 1.2?) As it turns out, two factors are involved in the answer.

The first is the complexity of *representing the language*. Regular language are mostly defined by regular expression, but the language of all regular expressions is context-free and not regular (Intuitively, since regular expressions uses parenthesizing). On the contrary, the definition of context-free languages is usually done with BNFs (or context-free grammars), and the language of all BNFs is regular.

Thus, in order to define a fluent API for generating regular languages, one needs a fluent API defined by a context-free language, while in order to define a fluent API for context-free languages, only a regular fluent API is needed.

Since fluflu "knows" only how to generate fluent APIs for regular languages, it cannot represent a fluent API for the generation of itself. Respectively, since FAJITA's language is regular, and can generate context-free fluent APIs, FAJITA can generate itself.

The second factor is the *practical complication* of generating the JAVA types to support the fluent API. As covered in Section 3.1 the generation of fluent APIs for regular languages is rather simple, while the generation of those for context-free languages is the main challenge of this thesis.

1.4 Contribution

The main results of this research are:

1. If the DSL specification is that of a deterministic context-free language, then a fluent API exists for the language, but we do not know whether such a fluent API exists for more general languages.

   Recall that there are universal cubic time parsing algorithms [12, 17, 52] which can parse (and recognize) any (non-deterministic) context-free language. What we do not know is whether algorithms of this sort can be encoded within the framework of the JAVA type system.

2. There exists an algorithm to generate a fluent API that realizes any deterministic context-free languages. Moreover, this fluent API can create at runtime,

a parse tree for the given language. This parse tree can then be supplied as input to the library that implements the language's semantics.

3. Unfortunately, a general purpose compiler-compiler is not yet feasible with the current algorithm.
   - One difficulty is usual in the fields of formal languages: The algorithm is complicated and relies on modules implementing complicated theoretical results, which, to the best of our knowledge, have never been implemented.
   - Another difficulty is that a certain design decision in the implementation of the standard `javac` compiler is likely to make it choke on the JAVA code generated by the algorithm.
4. We did implement a prototype for a compiler-compiler that works for LL(1) grammars.

Other concrete contributions made by this work include

- the understanding that the definition of fluent APIs is analogous to the definition of a formal language.
- a lower bound (deterministic pushdown automata) on the theoretical "computational complexity" of the JAVA type system.
- an algorithm for producing a fluent API for deterministic context-free languages (even if impractical).
- a collection of generic programming techniques, developed towards this algorithm.
- a demonstration that the runtime of Oracle's `javac` compiler may be exponential in the program size.
- a new interpretation to the LL parsing algorithm, under a "realtime" constraint.

The theoretical result that any deterministic context-free grammar can be automatically "compiled" to fluent API takes a toll of exponential blowup. Specifically, the construction first builds a deterministic pushdown automaton whose size parameters, g (number of stack symbols), and, q (number of internal states), are polynomial in the size of the input grammar. This automaton is then emulated by a weaker automaton, with as many as

$$O\left(g^{O(1)}\left(q^2 g^{O(1)}\right)^{qg^{O(1)}}\right)$$

stack symbols. This weaker automaton is then "compiled" into a collection of generic JAVA types, where there is at least one type for each of these symbols.

This work present an algorithm to compile an LL grammar of a fluent API language into a JAVA implementation whose size parameters are linear in the size parameters of the LL parser generated by the classical algorithm (Algorithm 3.5) for computing LL parsers, i.e., the performance loss due to implementation within the JAVA type checker is as small as we can hope it to be.

The savings are made possible by the use of a stronger automaton (the RLL_p, described in detail below in Section 3.4.1) for the emulation, and more efficient "compilation" of the RLL_p into the JAVA type system. We also present FAJITA[9] a JAVA tool that implements this algorithm.

1.5 Related Work

It has long been known that C++ templates are Turing complete in the following precise sense:

Proposition 1. *For every Turing machine, m, there exists a C++ program, C_m such that compilation of C_m terminates if and only if Turing-machine m halts. Furthermore, program C_m can be effectively generated from m [28].*

Intuitively, this is due to the fact that templates in C++ feature both recursive invocation and conditionals (in the form of "*template specialization*").

In the same fashion, it should be mundane to make the judgment that JAVA's generics are not Turing-complete since they offer no conditionals. Still, even though there are time complexity results regarding type systems in functional languages, we failed to find similar claims for JAVA.

Specialization, conditionals, `typedef`s and other features of C++ templates, gave rise to many advancements in template/generic/generative programming in the language [4,6,14,43], including e.g., applications in numeric libraries [49, 50], symbolic derivation [22] and a full blown template library [1].

Garcia et al. [21] compared the expressive power of generics in half a dozen major programming languages. In several ways, the JAVA approach [10] did not rank as well as others.

Unlike C++ [1, 4, 6, 14, 22, 43], the literature on meta-programming with JAVA generics is minimal, research concentrating on other means for enriching the language, most importantly annotations [45].

The work on SugarJ [18] is only one of many other attempts to achieve the embedded DSL effect of fluent APIs by language extensions.

Suggestions for semi-automatic generation can be found in the work of Bodden [9] and on numerous locations in the web. None of these materialized into an algorithm or analysis of complexity. However, there is a software artifact (the previously reviewed fluflu[10]) that automatically generates a fluent API that obeys the transitions of a given finite automaton.

Outline.

During the research we faced some interesting theoretical questions regarding automata theory and the JAVA type-checker ; Chapter 2 reviews these issues. Specifically, Section 2.3 presents our main theorem. Section 2.6 gives a proof to that theorem, Section 2.7 extends the theorem, and Section 2.4 provides a toolkit for type-encoding in JAVA.

Using the theoretical results from Chapter 2, Chapter 3 introduces a set of algorithms and data structures to automatically generate fluent APIs from BNF specifications. Specifically,

[9] ***F**luent **A**PI for **J**AVA (**I**nspired by the **T**heory of **A**utomata)*
[10] https://github.com/verhas/fluflu

Section 3.2 introduces Fajita, Section 3.3 reminds the core of LL parsing and gives some intuition to the main algorithm presented in Section 3.4.

Chapter 4 notes some limitations and interesting points regarding the implementation of the prototypical implementation of Fajita, while Chapter 5 concludes and discusses directions for future work.

Chapter 2

Formal Language Recognition with the Java Type Checker

Based on an article accepted to ECOOP'16 [25]

"Java generics are 100% pure syntactic sugar, and do not support meta-programming" [1]

2.1 Method Chaining, Fluent APIs, and, Type States

The pattern "invoke function on variable **sb**", specifically with a function named **append**, occurs six times in the code in Figure 2.1(a), designed to format a clock reading, given as integers hours, minutes and seconds.

Figure 2.1 Recurring invocations of the pattern "invoke function on the same receiver", before, and after method chaining.

```
String time(int hours, int minutes,
    int seconds) {
  final StringBuilder sb = new
   StringBuilder();
  sb.append(hours);
  sb.append(':');
  sb.append(minutes);
  sb.append(':');
  sb.append(seconds);
  return sb.toString();
}
```

(a) before

```
String time(int hours, int minutes,
    int seconds) {
    return new StringBuilder()
      .append(hours).append(':')
      .append(minutes).append(':')
      .append(seconds)
      .toString();
}
```

(b) after

Some languages, e.g., Smalltalk [26] offer syntactic sugar, called *cascading*, for abbreviating this pattern. *Method chaining* is a "programmer made" syntactic sugar serving the same purpose: If a method f returns its receiver, i.e., **this**, then, instead of the series of two commands: `o.`f`(); o.`g`();`, clients can write only one: `o.`f`().`g`();`. Figure 2.1(b) is

[1] Found on stackoverflow: http://programmers.stackexchange.com/questions/95777/generic-programming-how-often-is-it-used-in-industry

the method chaining (also, shorter and arguably clearer) version of Figure 2.1(a). It is made possible thanks to the designer of class `StringBuilder` ensuring that all overloaded variants of `append` return their receiver.

The distinction between *fluent API* and method chaining is the identity of the receiver: In method chaining, all methods are invoked on the same object, whereas in fluent API the receiver of each method in the chain may be arbitrary. Fluent APIs are more interesting for this reason. Consider, e.g., the following JAVA code fragment (drawn from JMock [20])

```
allowing(any(Object.class)).method("get.*").withNoArguments();
```

Let the return type of function `allowing` (respectively `method`) be denoted by τ_1 (respectively τ_2). Then, the fact that $\tau_1 \neq \tau_2$ means that the set of methods that can be placed after the dot in the partial call chain `allowing(any(Object.class)).` is not necessarily the same set of methods that can be placed after the dot in the partial call chain

```
allowing(any(Object.class)).method("get.*").
```

This distinction makes it possible to design expressive and rich fluent APIs, in which a sequence of "chained" calls is not only readable, but also robust, in the sense that the sequence is type correct only when it makes sense semantically.

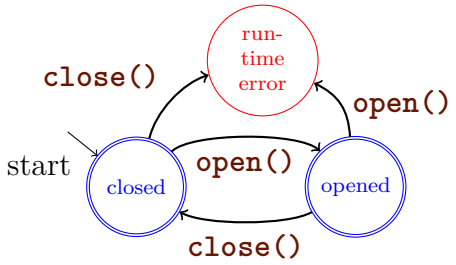
There is a large body of research on *type-states* (See e.g., review articles such as [2, 8]). Informally, an object that belongs to a certain type, has type-states, if not all methods defined in this object's class are applicable to the object in all states it may be in. As it turns out, objects with type states are quite frequent: a recent study [7] estimates that about 7.2% of JAVA classes define protocols, that can be interpreted as type-state.

In a sense, type states define the "language" of the protocol of an object. The protocol of the type-state `Box` class defined in Figure 2.2 admits the chain `new Box().open().close()` but not the chain `new Box().open().open()`.

Figure 2.2 Fluent API of a box object, defined by a DFA and a table

	`open()`	`close()`
"closed"	*become "open"*	*runtime error*
"open"	*runtime error*	*become "closed"*

(a) Definition by table



(b) Definition by DFA

As mentioned above, tools such as fluflu realize type-state based on their finite automaton description. Our approach is a bit more expressive: examine the language L defined by the type-state, e.g., in the box example,

$$L = \big(\texttt{.open().close()}\big)^*\big(\texttt{.open()} \mid \epsilon\big).$$

If L is deterministic context-free, a fluent API can be made for it.

To make the proof concrete, consider this example of fluent API definition: An instance of class `Box` may receive two method invocations: `open()` and `close()`, and can be in either

"open" or "closed" state. Initially the instance is "closed". Its behavior henceforth is defined by Figure 2.2.

To realize this definition, we need a type definition by which `new Box().open().close()`, more generally blue, or accepting states in the figure, type-check. Conversely, with this type definition, compile time type error should occur in `new Box().close()`, and, more generally, in the red state.

Some skill is required to make this type definition: proper design of class `Box`, perhaps with some auxiliary classes extending it, an appropriate method definition here and there, etc.

2.2 Context-Free Languages and Pushdown Automata

Notions discussed here are probably common knowledge (see e.g., [31, 41] for a text book description, or [5] for a scientific review). The purpose here is to set a unifying common vocabulary.

Let Σ be a finite alphabet of *terminals* (often called input characters or tokens). A *language* over Σ is a subset of Σ^*. Keep Σ implicit henceforth.

A ***N**ondeterministic **P**ushdown **A**utomaton* (NPDA) is a device for language recognition, made of a nondeterministic finite automaton and a stack of unbounded depth of (stack) *elements*. A NPDA begins execution with a single copy of the initial element on the stack. In each step, the NPDA examines the next input token, the state of the automaton, and the top of the stack. It then pops the top element from the stack, and nondeterministically chooses which actions of its transition function to perform: Consuming the next input token, moving to a new state, or, pushing any number of elements to the stack. Actually, any combination of these actions may be selected.

The language recognized by a NPDA is the set of strings that it accepts, either by reaching an accepting state or by encountering an empty stack.

A ***C**ontext-**F**ree **G**rammar*(CFG) is a formal description of a language. A CFG G has three components: Ξ a set of *variables* (also called nonterminals), a unique *start variable* $\xi \in \Xi$, and a finite set of (production) *rules*. A rule $r \in G$ describes the derivation of a variable $\xi \in \Xi$ into a string of *symbols*, where symbols are either terminals or variables. Accordingly, rule $r \in G$ is written as $r = \xi \to \beta$, where $\beta \in (\Sigma \cup \Xi)^*$. This description is often called BNF. The *language* of a CFG is the set of strings of terminals (and terminals only) that can be derived from the start symbol, following any sequence of applications of the rules. CFG languages make a proper superset of regular languages, and a proper subset of "context-sensitive" languages [31].

The expressive power of NPDAs and BNFs is the same: For every language defined by a BNF, there exists a NPDA that recognizes it. Conversely, there is a BNF definition for any language recognized by some NPDA.

NPDAs run in exponential deterministic time. A more sane, but weaker, alternative is found in LR(1) parsers, which are deterministic linear time and space. Such parsers employ a stack and a finite automaton structure, to parse the input. More generally, LR(k) parsers, $k > 1$, can be defined. These make their decisions based on the next k input character, rather than just the first of these. General LR(k) parsers are rarely used, since they offer essentially the same expressive power[2], at a greater toll on resources (e.g., size of the automaton). In fact, the

[2]they recognize the same set of languages [36].

expressive power of LR(k), $k \geq 1$ parsers, is that of "***D**eterministic **P**ushdown **A**utomaton*" (DPDA), which are similar to NPDA, except that their conduct is deterministic.

Definition 1 (Deterministic Pushdown Automaton)**.** *A deterministic pushdown automaton (DPDA) is a quintuple* $\langle Q, \Gamma, q_0, A, \delta \rangle$ *where* Q *is a finite set of* states, Γ *is a finite* set of elements, $q_0 \in Q$ *is the initial state, and* $A \subseteq Q$ *is the* set of accepting states *while* δ *is the* partial state transition function $\delta : Q \times \Gamma \times (\Sigma \cup \{\epsilon\}) \nrightarrow Q \times \Gamma^*$.

A DPDA begins its work in state q_0 *with a single designated stack element residing on the stack. At each step, the automaton examines: the current state* $q \in Q$*, the element* $\gamma \in \Gamma$ *at the top of the stack, and* σ*, the next input token, Based on the values of these, it decides how to proceed:*

1. *If* $q \in A$ *and the input is exhausted, the automaton accepts the input and stops.*
2. *Suppose that* $\delta(q, \gamma, \epsilon) \neq \perp$ *(in this case, the definition of a DPDA requires that* $\delta(q, \gamma, \sigma') = \perp$ *for all* $\sigma' \in \Sigma$*), and let* $\delta(q, \gamma, \epsilon) = (q', \zeta)$*. Then the automaton pops* γ *and pushes the string of stack elements* $\zeta \in \Gamma^*$ *into the stack.*
3. *If* $\delta(q, \gamma, \sigma) = (q', \zeta)$*, then the same happens, but the automaton also irrevocably consumes the token* σ*.*
4. *If* $\delta(q, \gamma, \epsilon) = \delta(q, \gamma, \sigma) = \perp$ *the automaton rejects the input and stops.*

A *configuration* is the pair of the current state and the stack contents. Configurations represent the complete information on the state of an automaton at any given point during its computation. A *transition* of a DPDA takes it from one configuration to another. Transitions which do not consume an input character are called ϵ*-transitions*.

As mentioned above, NPDA languages are the same as CFG languages. Equivalently, *DCFG languages* (deterministic context-free grammar languages) are context-free languages that are recognizable by a DPDA. The set of DCFG languages is still a proper superset of regular languages, but a proper subset of CFG languages.

2.3 Statement of the Main Result

Let java be a function that translates a terminal $\sigma \in \Sigma$ into a call to a uniquely named function (with respect to σ). Let $\mathsf{java}(\alpha)$, be the function that translates a string $\alpha \in \Sigma^*$ into a fluent API call chain. If $\alpha = \sigma_1 \cdots \sigma_n \in \Sigma^*$, then

$$\mathsf{java}(\alpha) = \mathsf{java}(\sigma_1)\texttt{().}\cdots\texttt{.}\mathsf{java}(\sigma_n)\texttt{()}$$

For example, when $\Sigma = \{a, b, c\}$ let $\mathsf{java}(a) = \texttt{a}$, $\mathsf{java}(b) = \texttt{b}$, and, $\mathsf{java}(c) = \texttt{c}$. With these,

$$\mathsf{java}(caba) = \texttt{c().a().b().a()}$$

theorem 1. *Let* A *be a DPDA recognizing a language* $L \subseteq \Sigma^*$*. Then, there exists a* Java *type definition,* J_A *for types* `L`*,* `A` *and other types such that the* Java *command*

$$\texttt{L } \ell \texttt{ = A.build.}\mathsf{java}(\alpha)\texttt{.\$();} \tag{2.1}$$

type checks against J_A *if an only if* $\alpha \in L$*. Furthermore, program* J_A *can be effectively generated from* A*.*

Equation (2.1) reads: starting from the `static` field `build` of `class A`, apply the sequence of call chain java(α), terminate with a call to the ending character `$()` and then assign to newly declared JAVA variable ℓ of type `L`.

The proof of the theorem is by a scheme for encoding in JAVA types the pushdown automaton $A = A(L)$ that recognizes language L. Concretely, the scheme assigns a type $\tau(c)$ to each possible configuration c of A. Also, the type of `A.build` is $\tau(c_0)$, where c_0 is the initial configuration of A,

Further, in each such type the scheme places a function $\sigma()$ for every $\sigma \in \Sigma$. Suppose that A takes a transition from configuration c_i to configuration c_j in response to an input character σ_k. Then, the return type of function σ_k`()` in type $\tau(c_i)$ is type $\tau(c_j)$.

With this encoding the call chain in Eq. (2.1) mimics the computation of A, starting at c_0 and ending with rejection or acceptance. The full proof is in Section 2.6.

Since the depth of the stack is unbounded, the number of configurations of A is unbounded, and the scheme must generate an infinite number of types. Genericity makes this possible, since a generic type is actually device for creating an unbounded number of types.

There are several, mostly minor, differences between the structure of the JAVA code in Eq. (2.1) and the examples of fluent API we saw above:

Prefix, i.e., the starting `A.build` variable. All variables and functions of JAVA are defined within a class. Therefore, a call chain must start with an object (`A.build` in Eq. (2.1)) or, in case of `static` methods, with the name of a class. In fluent API frameworks this prefix is typically eliminated with appropriate `import` statements.

If so desired, the same can be done by our type encoding scheme: define all methods in type $\tau(c_0)$ as `static` and `import static` these.

Suffix, i.e., the terminal `.$()` call. In order to know whether $\alpha \in L$ the automaton recognizing L must know when α is terminated.

With a bit of engineering, this suffix can also be eliminated. One way of doing so is by defining type `L` as an `interface`, and by making all types $\tau(c)$, c is an accepting configuration, as subtype of `L`.

Parameterized methods. Fluent API frameworks support call chains with phrases such as:

- "`.when(header(foo).isEqualTo("bar")).`",
- "`.and(BOOK.PUBLISHED.gt(date("2008-01-01"))).`", and,
- "`.allowing(any(Object.class)).`".

while our encoding scheme assumes methods with no parameters.

Methods with parameters contribute to the user experience and readability of fluent APIs but their "computational expressive power"' is the same. In fact, extending Theorem 1 to support these requires these conceptually simple steps

1. Define the structure of parameters to methods with appropriate fluent API, which may or may not be, the same as the fluent API of the outer chain, or the fluent API of parameters to other methods. Apply the theorem to each of these fluent APIs.

2. If there are several overloaded versions of a method, consider each such version as a distinct character in the alphabet Σ and in the type encoding of the automaton.
3. Add code to the implementation of each method code to store the value of its argument(s) in a record placed at the end of the fluent-call-list.

2.4 Techniques of Type Encoding

This section presents techniques and idioms of type encoding in JAVA partly to serve in the proof of Theorem 1, and partly to familiarize the reader with the challenges of type encoding.

Let $g : \Gamma \nrightarrow \Gamma$ be a partial function, from the finite set Γ into itself. We argue that g can be represented using the compile-time mechanism of JAVA. Figure 2.4 encodes such a partial function for $\Gamma = \{\gamma_1, \gamma_2\}$, where $g(\gamma_1) = \gamma_2$ and $g(\gamma_2) = \bot$, i.e., $g(\gamma_2)$ is undefined.[3]

Figure 2.3 Type hierarchy of partial function $g : \Gamma \nrightarrow \Gamma$, defined by $\Gamma = \{\gamma_1, \gamma_2\}$, $g(\gamma_1) = \gamma_2$ and $g(\gamma_2) = \bot$.

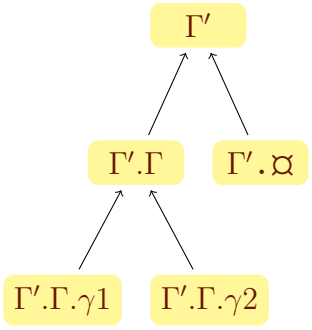


Figure 2.4 Type encoding of partial function defined in Figure 2.3

```
public static abstract class Γ′ {
  private static abstract class ¤ extends Γ′ {
    // Empty private class, cannot be used by clients.
    private ¤() { /* Private constructor hinders extension by clients */ }
  }
  public  static abstract class Γ extends Γ′ {
    public abstract Γ′ g();
    public static final class γ1 extends Γ {
      // Covariant return type in overriding:
      @Override public γ2 g() { return null; }
    }
    public static final class γ2 extends Γ {
      // Covariant return type in overriding:
      @Override  public Γ′.¤ g() { return null; }
    }
  }
}
```

[3]Unless otherwise stated, all code excerpts here represent full implementations, and automatically extracted, omitting headers and footers, from JAVA programs that compile correctly with a JAVA 8 compiler.

Figure 2.5 Use cases of partial function defined in Figure 2.3

```
public static void five_use_cases_of_function_g() {
  γ2 _1 = new γ1().g(); // ✓
  γ1 _2 = new γ2().g(); // ✗ type mismatch
  Γ′.¤ _3 = new γ2().g(); // ✗ class ¤ is private
  Γ′ _4 = new γ2().g(); // ✓
  _4.g(); // ✗ g() undefined in type Γ′
}
```

The type hierarchy depicted in Figure 2.3 shows five classes: Abstract class Γ [4] represents the set Γ, final classes **γ1**, **γ2** that extend Γ, represent the actual members of the set Γ. The remaining two classes are private final class **¤** that stands for an error value, and abstract class Γ′ that denotes the augmented set $\Gamma \cup \{¤\}$. Accordingly, both classes **¤** and Γ extend Γ′.[5]

The full implementation of these classes is provided in Figure 2.4. This actual code excerpt should be placed as a nested class of some appropriate host class. Import statements are omitted, here and henceforth for brevity.

The use cases in Figure 2.5 explain better what we mean in saying that function g is encoded in the type system: An instance of class **γ1** returns a value of type **γ2** upon method call **g()**, while an instance of class **γ2** returns a value of our **private** error type **Γ′.¤** upon the same call.

Three recurring idioms employed in Figure 2.4 are:

1. An **abstract** class encodes a set (alternatively, one can use **interfaces**). Abstract classes that extend it encode subsets, while **final** classes encode set members.

2. The interest of frugal management of name-spaces is served by the agreement that if a class X **extends** another class Y, then X is also defined as a **static** member class of Y.

3. Bodies of functions are limited to a single **return null;** command (with interfaces the method body is redundant). This is to stress that at runtime, the code does not carry out any useful or interesting computation, and the class structure is solely for providing compile-time type checks. [6]

Having seen how inheritance and overriding make possible the encoding of unary functions, we turn now to encoding higher arity functions. With the absence of multi-methods, other techniques must be used.

Consider the partial binary function $f : R \times S \nrightarrow \Gamma$, defined by

$$\begin{array}{llll} R = \{r_1, r_2\} & f(r_1, s_1) = \gamma_1 & f(r_2, s_1) = \gamma_1 \\ S = \{s_1, s_2\} & f(r_1, s_2) = \gamma_2 & f(r_2, s_2) = \bot \end{array} . \tag{2.2}$$

A JAVA type encoding of this definition of function f is in Figure 2.6; use cases are in Figure 2.7.

[4] Remember that JAVA admits Unicode characters in identifier names

[5] The use of short names, e.g., Γ instead of Γ′.Γ, is made possible by an appropriate **import** statement omitted here and henceforth.

[6] A consequence of these idioms is that the augmented class Γ′ is visible to clients. It can be made **private**. Just move class Γ to outside of Γ′, defying the second idiom.

Figure 2.6 Type encoding of partial binary function $f : R \times S \nrightarrow \Gamma$, where $R = \{r_1, r_2\}$, $S = \{s_1, s_2\}$, and f is specified by $f(r_1, s_1) = \gamma_1$, $f(r_1, s_2) = \gamma_2$, $f(r_2, s_1) = \gamma_1$, and $f(r_2, s_2) = \bot$ (except for classes Γ, Γ', γ**1**, and γ**2**, found in Figure 2.4)

```
public static abstract class f { // Starting point of fluent API
  public static r1 r1() { return null; }
  public static r2 r2() { return null; }
}
public static abstract class R {
  public abstract Γ' s1();
  public abstract Γ' s2();
  public static final class r1 extends R {
    @Override public γ1 s1() { return null; }
    @Override public γ2 s2() { return null; }
  }
  public static final class r2 extends R {
    @Override public γ2 s1() { return null; }
    @Override public Γ'.¤ s2() { return null; }
  }
}
```

Figure 2.7 Use cases of partial binary function defined in Figure 2.6

```
public static void four_use_cases_of_function_f() {
  γ1 _1 = f.r1().s1(); // ✓ f(r1, s1) = γ1
  γ2 _2 = f.r1().s2(); // ✓ f(r1, s2) = γ2
  γ2 _3 = f.r2().s1(); // ✓ f(r2, s1) = γ2
  f.r2().s2().g(); // ✗ method s2() undefined in type Γ'
}
```

As the figure shows, to compute $f(r_1, s_1)$ at compile time we write **`f.r1().s1()`**. Also, the fluent API call chain **`f.r2().s2().g()`** results in a compile time error because

$$f(r_2, s_2) = \bot.$$

Class **`f`** in the implementation sub-figure serves as the starting point of the little fluent API defined here. The return type of **`static`** member functions **`r1()`** and **`r2()`** is the respective sub-class of class **`R`**: The return type of function **`r1()`** is class **`R.r1`**; the return type of function **`r2()`** is class **`R.r2`**.

Instead of representing set S as a class, its members are realized as methods **`s1()`** and **`s2()`** in class **`R`**. These functions are defined as **`abstract`** with return type **`Γ'`** in **`R`**. Both functions are overridden in classes **`r1`** and **`r2`**, with the appropriate covariant change of their return type,

It should be clear now that the encoding scheme presented in Figure 2.6 can be generalized to functions with any number of arguments, provided that the domain and range sets are finite. The encoding of sets of unbounded size require means for creating an unbounded number of types. Genericity can be employed to serve this end.

Figure 2.8 shows a genericity based recipe for a function whose return type is the same as

Figure 2.8 Covariant return type of function `id()` with JAVA generics.

```
interface ID<T extends ID<?>> {
  default T id() { return null; }
}
class A implements ID<A> { /**/ }
abstract class B<Z extends B<?>> implements ID<Z> { /**/ }
class C extends B<C> { /**/ }
```

the receiver's type. This recipe is applied in the figure to classes `A`, `B`, and `C`. In each of these classes, the return type of `id` is, without overriding, (at least) the class itself.

It is also possible to encode with JAVA generic types unbounded data structures, as demonstrated in Figure 2.9, featuring a use case of a stack of an *unbounded* depth.

Figure 2.9 Use cases of a compile-time stack data structure.

```
1 public static void use_case_of_stack() {
2   // Create a stack a with five items in it:
3   P<γ1, P<γ1, P<γ2, P<γ1, P<γ1, E>>>>> _1 = Stack.empty.γ1().γ1().γ2().
      γ1().γ1();
4   P<γ1, P<γ2, P<γ1, P<γ1, E>>>> _2 = _1.pop(); // ✓ Pop one item
5   P<γ2, P<γ1, P<γ1, E>>> _3 = _2.pop(); // ✓ Pop another item
6   P<γ1, P<γ1, E>> _4 = _3.pop(); // ✓ Pop yet another item
7   P<γ1, E> _5 = _4.pop(); // ✓ Pop penultimate item
8   γ1 _6 = _5.top(); // ✓ Examine last item
9   E _7 = _5.pop(); // ✓ Pop last item
10  Stack.¤ _8 = _7.pop(); // ✗ Cannot pop from an empty stack
11  Γ′.¤ _9 = _7.top(); // ✗ empty stack has no top element
12 }
```

In line 3 of the figure, a stack with five elements is created: These are popped in order (ll.4–7,l.9). Just before popping the last item, its value is examined (l.8). Trying then to pop from an empty stack (l.10), or to examine its top (l.11), ends with a compile time error.

Stack elements may be drawn from some abstract set Γ. In the figure these are either class γ`1` or class γ`2` (both defined in Figure 2.4). A call to function γi pushes the type γi into the stack, for $i = 1,$. The expression

`Stack.empty.`γ`1().`γ`1().`γ`2().`γ`1().`γ`1()`

represents the sequence of pushing the value γ_1 into an empty stack, followed by γ_1, γ_2, γ_1, and, finally, γ_1. This expression's type is that of variable `_1`, i.e.,

`P<`γ`1,P<`γ`1,P<`γ`2,P<`γ`1,P<`γ`1,E>>>>>

A recurring building block occurs in this type: generic type `P`, *short for "Push"*, which takes two parameters:

1. the *top* of the stack, always a subtype of Γ,
2. the *rest* of the stack, which can be of two kinds:

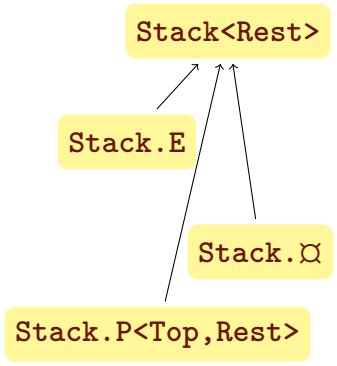
(a) another instantiation of `P` (in most cases),

(b) non-generic type `E`, *short for "Empty"*, which encodes the empty stack. Note that `E` can only occur at the deepest `P`, encoding a stack with one element, in which the rest is empty.

Incidentally, `static` field `Stack.empty` is of type `E`.

Figure 2.10 gives the type inheritance hierarchy of type `Stack` and its subtypes. Figure 2.11 gives the implementation of these types. The code in the figure shows that the "rest"

Figure 2.10 Type hierarchy of an unbounded stack data structure



parameter of `P` must extend class `Stack`, and that both types `P` and `E` extend `Stack`. Other points to notice are:

- The type at the top of the stack is precisely the return type of `top()`; it is overridden in `P` so that its return type is the first argument of `P`. The return type of `top()` in `E` is the error value $\Gamma'.\text{¤}$.
- Pushing into the stack is encoded as functions γ`1()` and γ`2()`; the two are overridden with appropriate covariant change of the return type in `P` and `E`.
- Since an empty stack cannot be popped, function `pop()` is overridden in `E` to return the *error* type `Stack.¤`. This type is indeed a kind of a stack, except that each of the four stack functions: `top()`, `push()`, γ`1()`, and, γ`2()`, return an appropriate error type.

In fact, this recursive generic type technique can used to encode S-expressions: In the spirit of Figure 2.11, the idea is to make use of a `Cons` generic type with covariant `car()` and `cdr()` methods.

A standard technique of template programming in C++ is to encode conditionals with template specialization. Since JAVA forbids specialization of generics, in lieu we use covariant overloading of function return type (e.g., the return type of `s2()` in Figure 2.6 and the return type of `top()` in Figure 2.11).

Figure 2.12 shows that a similar covariant change is possible in extending a generic type. The type of the parameter `M` to `Heap` is "`? extends Mammals`". This type is specialized as `School` extends `Heap`: parameter `W` of `School` is of type "`? extends Whales`". Covariant specialization of parameters to generics is yet another idiom for encoding conditionals.

Overloading gives rise to a third idiom for partial emulation of conditionals, as can be seen in Figure 2.13.

The figure depicts type `Peep` and overloaded versions of `peep()` which together make it possible to extract the top of the stack. The first generic parameter to `Peep` is the top of the

Figure 2.11 Type encoding of an unbounded stack data structure (except for classes Γ, Γ′, γ1, and γ2, which is in Figure 2.4)

```
public static abstract class Stack<Rest extends Stack<?>> {
  public abstract Γ′ top();
  public abstract Rest pop();
  public abstract Stack<?> γ1();  // Push type Γ′.Γ.γ1
  public abstract Stack<?> γ2();  // Push type Γ′.Γ.γ2
  public static class P<Top extends Γ, Rest extends Stack<?>>
    extends Stack<Rest> { // Type of a non−empty stack:
       @Override public Top top() { return null; }
       @Override public Rest pop() { return null; }
       @Override public P<γ1, P<Top,Rest>> γ1() { return null; }
       @Override public P<γ2, P<Top,Rest>> γ2() { return null; }
  }
  public static final class E
    extends Stack<¤> { // Type of an empty stack
      @Override public Γ′.¤ top() { return null; }
      @Override public ¤ pop() { return null; }
      @Override public P<γ1, E> γ1() { return null; }
      @Override public P<γ2, E> γ2() { return null; }
  }
  public static final E empty = null;
  private static final class ¤
    extends Stack<¤> { // Type of pop from empty stack
      @Override public Γ′.¤ top() { return null; }
      @Override public ¤ pop() { return null; }
      @Override public ¤ γ1() { return null; }
      @Override public ¤ γ2() { return null; }
  }
}
```

Figure 2.12 Covariance of parameters to generics

```
class Mammals { /* ... */ }
class Heap<M extends Mammals> { /* ... */}
class Whales extends Mammals { /* ... */}
class School<W extends Whales>
  extends Heap<W> { /* ... */}
```

stack, the second is the stack itself. Indeed, we see (l.11) that peeping into an empty stack, places a `?` in the first parameter, thanks to the first overloaded version of `peep()` (l.2).

The second overloaded version of `peep()` (ll.3–6) matches against all non-empty stacks. The return type of this version encodes in its first parameter the top of the stack, and in its second parameter, the parameter's type. A use case is in line 9.

Let τ be the type of the top of a given stack. Then, both `top()` and `peep()` can be used to extract τ. There is a subtle difference between the two though: Obtaining τ from `top()` does

Figure 2.13 Peeping into the stack

```
1 public static class Peep<γ extends Γ′, S extends Stack<? extends Stack
    <?>>> {}
2 public static Peep<?, E> peep(E _) { return null; } // First overloaded version
    of peep()
3 public static                                        // Second overloaded
    version of peep()
4 <Top extends Γ, Rest extends Stack<?>> // Two generic parameters
5 Peep<Top, P<Top, Rest>>                // Function return type
6 peep(P<Top, Rest> _) { return null; }  // Function parameters and body
7 public static void peeping_into_a_stack_use_cases() {
8   P<γ2, P<γ1, P<γ2, P<γ1, P<γ2, E>>>>> _1 = Stack.empty.γ2().γ1().γ2().
      γ1().γ2();
9   Peep<γ2, P<γ2, P<γ1, P<γ2, P<γ1, P<γ2, E>>>>>> _2 = peep(_1);
10  E _3 = Stack.empty;
11  Peep<?, E> _4 = peep(_3);
12 }
```

not make it possible to define variables, function return types, and parameters to functions and generics whose type is τ or depends on it in any way. However, since **Peep** is a type that receives τ as parameter, the body of **Peep** is free to define e.g., functions signature includes on τ, or pass τ further to other generics.

2.5 The Jump-Stack Data-Structure

A *jump-stack* is a stack data structure whose elements are drawn from a finite set Γ, except that jump-stack supports $\mathsf{jump}(\gamma)$, $\gamma \in \Gamma$ operations (which means "repetitively *pop* elements from the stack up to and including the first occurrence of γ").

Figure 2.14 shows the skeleton of type-encoding, in parameterized type **JS**, of a jump-stack whose elements are drawn from type Γ (Figure 2.4), i.e., either γ**1** or γ**2**.

Just like **Stack** (Figure 2.11), **JS** takes a **Rest** parameter encoding the type of a jump-stack after popping. In addition **JS** takes $k = |\Gamma|$ type parameters, one for each $\gamma \in \Gamma$, which is the type encoding of the jump-stack after a $\mathsf{jump}(\gamma)$ operation. In the figure, there are two such parameters: **J_**γ**1**, and **J_**γ**2**.

Functions defined in **JS** include not only the standard stack operations: **top()**, **pop()**, γ**1()** and γ**2()** (encoding a push of γ_i, $i = 1, 2$, in general, there are k), but also k functions encoding $\mathsf{jump}(\gamma)$, $\gamma \in \Gamma$. In our case, these are **jump_**γ**1** and **jump_**γ**2**, which encode $\mathsf{jump}(\gamma_i)$ thanks to their return type being **J_**γi, $i = 1, 2$.

The type hierarchy rooted at **JS** is similar to that of Figure 2.10: Two of the specializations are parameter-less and are almost identical to their **Stack** counterparts: **JS.E** encodes an empty jump-stack; **JS.¤** encodes a jump-stack in error, e.g., after popping from **JS.E**. The body of these two types is omitted here.

Type **JS.P** (lines 16–23 in Figure 2.14) makes the third specialization of **JS**, encoding a stack with one or more elements. Just like in Figure 2.8, there are no overridden functions in **JS.P**; it fulfills its duties through the type parameters it takes and the types it passes to **P'**

Figure 2.14 Skeleton of type encoding for the jump-stack data structure

```
1 public interface JS<   // 1 + k generic parameters
2   //As a convention, we use JS with its raw type when no parameters are introduced
3     Rest extends JS,
4     J_γ1 extends JS,
5     J_γ2 extends JS
6 > {
7   Γ′ top();
8   Rest pop();
9   JS γ1();
10  JS γ2();
11  J_γ1 jump_γ1();
12  J_γ2 jump_γ2();
13  interface ¤ extends JS<¤, ¤, ¤> { ... }
14  public interface E extends JS<¤, ¤, ¤> { ... }
15  public static final E empty = null;
16  public interface P<// 2 + k generic arguments:
17      Top extends Γ,
18      Rest extends JS,
19      J_γ1 extends JS,
20      J_γ2 extends JS
21  > extends P′<Top, Rest, J_γ1, J_γ2,
22        P<Top, Rest, J_γ1, J_γ2>
23  > { /**/ }
24 }
```

Figure 2.15 Auxiliary type `P'` encoding succinctly a non-empty jump-stack

```
1 private interface P′<
2 // 2 + k + 1 generic arguments:
3   Top extends Γ,
4   Rest extends JS,
5   J_γ1 extends JS,
6   J_γ2 extends JS,
7   Me extends JS
8 > extends JS<Rest, J_γ1, J_γ2> {
9    public Top top();
10   P<γ1, Me, Me, J_γ2> γ1();
11   P<γ2, Me, J_γ2, Me> γ2();
12 }
```

the generic type it extends.

Specifically, **JS.P** takes the same **Top** and **Rest** parameters (ll.17–18) as type **Stack.P**: as well as k additional parameters: **J_γ1** and **J_γ2** (ll.19–20) which are the types encoding the jump-stack after the execution $\mathsf{jump}(\gamma_i)$, $i = 1, 2$. Type **JP.P** passes these four parameters to type **P'** which it extends (l.21). The fifth parameter to **P'** (l.22) is the current incarnation

of `P`, i.e., `P<Top, Rest, J_γ1, J_γ2>`.

The auxiliary (and `private`) type `P'` itself is depicted in Figure 2.15. By extending type `JS` and passing the correct `Rest` (respectively, `J_γ1`, `J_γ2`) parameter to it, `P'` inherits correct declaration of function `pop()` (l.8 Figure 2.14) (respectively `jump_γ1` (l.11 ibid), `jump_γ2` (l.12 ibid)).

More importantly, the `Me` type parameter to `P'` represents type `JP.P` that extends `P'`. Type `Me` also captures the actual parameters *included* to `JP.P`, which makes it possible to write the return type of `γ1()` and `γ2()` more succinctly. Let, e.g., τ = `P<γ1, Me, Me, J_γ2>` be the return type of `γ1()`. The first two parameters to τ say that pushing `γ1`, results in a compound jump-stack, whose top element is `γ1`, and where the rest of the jump-stack is the current type. The third parameter to τ says that since `γ1` was pushed the result of a $\mathsf{jump}(\gamma_1)$ is the type of the receiver. The fourth parameter is `J_γ2` since a push of γ_1 does not change the result of $\mathsf{jump}(\gamma_2)$.

Figure 2.16 Use cases for the `JS` type hierarchy

```
public static void jump_stack_use_cases(){
  P<
    γ1,                                  // Top
    P<γ1,P<γ2,E,¤,E>,P<γ2,E,¤,E>,E>,// Rest
    P<γ1,P<γ2,E,¤,E>,P<γ2,E,¤,E>,E>,// jump(γ1)
    E                                    // jump(γ2)
  > _1 = JS.empty.γ2().γ1().γ1();
  E _2 = _1.jump_γ2();
  P<
    γ1,              // Top
    P<γ2,E,¤,E>,     // Rest
    P<γ2,E,¤,E>      // jump(γ1)
    ,E               // jump(γ2)
  > _3 = _1.jump_γ1();
}
```

Some use cases for the encoded jump-stack data structure are in Figure 2.16. The type of variable `_1` encodes a stack into which `γ2`, `γ1`, `γ1` were pushed (in this order). Examining the type of `_2` we see that executing `jump_γ2` on `_1`, yields the empty stack in a single step. The type of `_3` is that state of the same stack after executing `jump_γ1`; it is exactly the same as popping a single element from the stack.

2.6 Proof of the Main Theorem

On a first sight, the proof of Theorem 1 could follow the techniques sketched in Section 2.4 to type encode a DPDA (Definition 1). The partial transition function δ may be type encoded as in Figure 2.6, and the stack data structure of a DPDA can be encoded as in Figure 2.11.

The techniques however fail with ϵ-transitions, which allow the automaton to move between an unbounded number of configurations and maneuver the stack in a non-trivial manner, without making any progress on the input. The fault in the scheme lies with compile time computation being carried out by the $\mathsf{java}(\sigma)()$ functions, each converting their receiver type to the type of the receiver of the next call in the chain. We are not aware of a JAVA type

encoding which makes it possible to convert an input type into an output type, where the output is computed from the input by an unbounded number of steps. [7]

The literature speaks of finite-delay DPDAs, in which the number of consecutive ϵ-transitions is uniformly bounded and even of realtime DPDAs in which this bound is 0, i.e., no ϵ-transitions. Our proof relies on a special kind of realtime automata, described by Courcelle [13].

Definition 2 (Simple-Jump Single-State Realtime Deterministic Pushdown Automaton)**.** *A simple-jump, single-state, realtime deterministic pushdown automaton (jDPDA, for short) is a triplet* $\langle \Gamma, \gamma_1, \delta \rangle$ *where* Γ *is a set of stack elements,* $\gamma_1 \in \Gamma$ *is the initial stack element, and* δ *is the* partial transition function, $\delta : \Gamma \times \Sigma \nrightarrow \Gamma^* \cup j(\Gamma)$,

$$j(\Gamma) = \{\textit{instruction } \mathsf{jump}(\gamma) \mid \gamma \in \Gamma\}.$$

A configuration of a jDPDA is some $c \in \Gamma^*$ *representing the stack contents. Initially, the stack holds* γ_1 *only. For technical reasons, assume that the input terminates with* $\$ \notin \Sigma$*, a special end-of-file character.*

- *At each step a jDPDA examines* γ*, the element at the top of the stack, and* $\sigma \in \Sigma$*, the next input character, and executes the following:*
  1. *consume* σ
  2. *if* $\delta(\gamma, \sigma) = \zeta$, $\zeta \in \Gamma^*$*, the automaton pops* γ*, and pushes* ζ *into the stack.*
  3. *if* $\delta(\gamma, \sigma) = \mathit{jump}(\gamma')$, $\gamma' \in \Gamma$*, then the automaton repetitively pops stack elements up-to and including the first occurrence of* γ'*.*
- *If the next character is* \$*, the automaton may reject or accept (but nothing else), depending on the value of* γ*.*

In addition, the automaton rejects if $\delta(\gamma, \sigma) = \bot$ *(i.e., undefined), or if it encounters an empty stack (either at the beginning of a step or on course of a jump operation).*

	γ_1	γ_2
σ_1	$\mathsf{push}(\gamma_1, \gamma_1, \gamma_2)$	$\mathsf{push}(\gamma_2, \gamma_2)$
σ_2	$\bot$	$\mathsf{push}(\epsilon)$
σ_3	$\bot$	$\mathsf{jump}(\gamma_1)$
\$	accept	reject

Table 2.1: The transition function of a jDPDA A, $\Sigma = \{\sigma_1, \sigma_2, \sigma_3\}$, $\Gamma = \{\gamma_1, \gamma_2\}$ where γ_1 is the initial element

As it turns out, every DCFG language is recognized by some jDPDA, and conversely, every language accepted by a jDPDA is a DCFG language [13]. The proof of Theorem 1 is therefore

[7] With the presumption that the JAVA compiler halts for all inputs (a presumption that does not hold for e.g., C++, and was never proved for JAVA), the claim that there is no JAVA type encoding for all DPDAs can be proved: Employing ϵ-transitions, it is easy to construct an automaton A^∞ that never halts on any input. A type encoding of A^∞ creates programs that send the compiler in an infinite loop.

reduced to type-encoding of a given jDPDA. Towards this end, we employ the type-encoding techniques developed above, and, in particular, the jump-stack data structure (Figure 2.14).

Henceforth, let $k = |\Gamma|$, $\ell = |\Sigma|$. The simple $k = 2$, $\ell = 3$ jDPDA A defined in Table 2.1 will serve as our running example. Let L be the language recognized by A.[8]

2.6.1 Main Types

Generation of a type encoding for a jDPDA starts with two empty types for sets L, Σ^*, where L represents the languages accepted by the jDPDA and Σ^* represents all words:

```
private static class ΣΣ    // Encodes set Σ*, type of reject
  { /* empty */  }
static class L extends ΣΣ  // Encodes set L ⊆ Σ*, type of accept
  { /* empty */  }
```

(The full type encoding is in Figure 2.18 below; to streamline the reading, we bring excerpts as necessary.)

A configuration is encoded by a generic type `C`. Essentially, `C` is a representation of the stack, but $k + 1$ type parameters are required:

- `Rest`, a type encoding of the stack after a pop (or jump with the top element), and,
- k types, named `JR`γ`1`, … ,`JR`γk, encoding the type of `Rest` after jump(γ_1),… jump(γ_k).

Note that these $k + 1$ parameters are sufficient for describing a configuration, i.e., if the top is γ_j, then for all $j \neq i$

$$\mathsf{jump}(\gamma_j) = \texttt{Rest.}\mathsf{jump}(\gamma_j)$$

In the special case of jump(γ_i) the returned type is still `Rest`, this is due to the fact that before a jump operation, we do not pop an element from the stack.

All instantiations of `C` must make sure that actual parameters are properly constrained, to ensure that they are (the type version of) pointers into the actual stack, not a trivial task, as will be seen shortly.

In the running example, `C` is defined as:

[8] Incidentally,

$$L = \{w^* \mid w = (\sigma_1^n \sigma_2^m \sigma_3 | \sigma_1^n \sigma_2^n) \ , \ n > m, n > 1\}$$

which is clearly not-regular; the equivalent BNF for L is:

$$S \to WS \,|\, \epsilon \ ; \ W \to D \,|\, AD\sigma_3 \ ; \ D \to \sigma_1 D \sigma_2 \,|\, \sigma_1\sigma_2 \ ; \ A \to A\sigma_1 \,|\, \sigma_1$$

Neither the BNF nor the representation are material for the proof.

```
interface C<        // Generic parameters:
  Rest extends C,   // The rest of the stack, for pop or jump(γ) operations
  JRγ1 extends C,   // Type of  Rest.jump(γ1), may be rest, or anything in it
    .
  JRγ2 extends C    // Type of  Rest.jump(γ2), may be rest, or anything in it
    .
>
{
  ΣΣ $();           // δ transition on end of input; invalid language by
    default
  C σ1();           // δ transition on σ1; dead end by default
  C σ2();           // δ transition on σ2; dead end by default
  C σ3();           // δ transition on σ3; dead end by default
  public interface E extends C<¤,¤,¤> { /* Empty stack configuration
    */ }
  interface ¤ extends C<¤,¤,¤> { /* Error configuration. */ }
}
```

This excerpt shows also classes `E` and `¤` which encode (as in Figure 2.14) the empty and the error configurations.

Type `C` defines $\ell + 1$ functions (4 in the example), one for each possible input character, and one for the end-of-file character defined as \$. Since `C` encodes an abstract configuration, return types of functions in it are the appropriate defaults which intentionally fail to emulate the automaton's execution. The return type of `$()` is ΣΣ (rejection); the transition functions `σ1()`, … `σℓ()`, return the raw type `C`.

2.6.2 Top-of-Stack Types

Types `Cγ1`, … ,`Cγk`, specializing `C`, encode stacks whose top element is γ_1, … ,γ_k. In A there are two of these:

```
interface Cγ1< // Configuration when γ1 is at top
  Rest extends C, JRγ1 extends C, JRγ2 extends C
> extends
  C<Rest, JRγ1, JRγ2>
{
}
interface Cγ2< // Configuration when γ2 is at top
  Rest extends C, JRγ1 extends C, JRγ2 extends C
> extends
  C<Rest, JRγ1, JRγ2>
{
}
static Cγ1<E,¤,¤> build = null;
```

In A, types `Cγ1` and `Cγ2` take three parameters; in general "Top of Stack" types take the aforementioned $k + 1$ parameters.

Figure 2.17 Accepting and non-accepting call chains with the type encoding of jDPDA A (as defined in Table 2.1). All lines in `accepts()` type-check, while all lines in `rejects()` do not type-check.

```
static void isL(L l) {/**/}
static void accepts() {
  isL(A.build.$());
  isL(A.build.σ1().σ3().$());
  isL(A.build.σ1().σ2().$());
  isL(A.build.σ1().σ1().σ2().σ3().σ1().σ2().$());
}
static void rejects() {
  isL(A.build.σ1().$());
  isL(A.build.σ2().σ1().$());
  isL(A.build.σ1().σ2().σ3().$());
  isL(A.build.σ1().σ1().σ2().σ3().σ1().$());
}
```

The method signatures of these types are generated using the mentioned parameters. The generating of methods will be discussed next.

The code defines the `static` variable `build`, the starting point of all fluent API call chains, to be of type `Cγ1<E,¤,¤>`, i.e., the starting configuration of the automaton is a stack whose top is γ_1, and its `Rest` parameter is empty (`E`). Any of the two jumps possible on this rest results with, `¤`, an undefined stack. Examples of accepting and rejecting call chains starting at `A.build` can be seen in Figure 2.17.

2.6.3 Transitions

It remains to show the type encoding of δ, the transition function. Overall, there are a total of $k \cdot (\ell + 1)$ entries in a transition table such as Table 2.1. Conceptually, these are encoded by selecting the correct return type of functions `σ1()`, … ,`σk()` and `$()` in each of the k "Top of Stack" types. Thanks to inheritance, we need to do so only in the cases that this return type is different from the default.

Overall, there are six kinds of entries in a transition table:

reject The default return type of `$()` in `C` is ΣΣ, which is *not* a subtype of `L`. Normally the result of a call chain that ends with `$()` cannot be assigned to a variable of type `L`. Moreover, since ΣΣ is `private`, there is little that clients can do with this result.

accept The only case in which fluent call chain ending with `$()` can return type `L` is when the type returned of the call just prior to `.$()` covariantly changes the return type of `$()` to `L`.[9]

Recall that a jDPDA can only accept after its input is exhausted. In Table 2.1 we see that accept occurs when the top of the stack is γ_1. We therefore add to the body of type `Cγ1` the line

[9]This is not to be confused with dynamic binding; types of fluent API call chains are determined statically.

```
@Override L $();
```

⊥ When a prefix of the input is sufficient to conclude it must be rejected however it continues, the transition function returns ⊥. In A this occurs when the top of the stack is γ_1 and one of σ_2 or σ_3 is read. To type encode $\delta(\gamma_1, \sigma_2) = \bot$, one must *not* override σ`2()` in type `C`γ`1`; the inherited return type (l.15 Figure 2.18) is the raw `C`. Subsequent calls in the chain will all receive and return a raw `C` (Recall that all σi`()`, $i = 1, \ldots, \ell$, are functions in `C` that return a raw `C`). Therefore, the final `$()` will reject.

Two other situations in which a jDPDA rejects but not demonstrated in A are: a jump that encounters an empty stack, and reading a character from when the stack is empty. In our type encoding these are handled by the special types `E` and `¤` (ll.17–18 ibid), both extend `C` without overriding any of its methods. Again, remaining part of the call chain will stick to raw `C`s up until the final `$()` call rejects the input.

jump(γ_i) The design of the generic parameters makes the implementation of jump(γ_i) operations particularly simple. All that is required is to covariantly change the return type of the appropriate σj`()` function to the appropriate `JR`γi or `Rest` parameter (recall that a jump occurs after popping the current element from the stack, so we refer to `JR` type parameters rather than `J`'s).

In Table 2.1 we find that $\delta(\gamma_2, \sigma_3) = \text{jump}(\gamma_1)$. Accordingly, the type of σ`2()` in `C`γ`2` (l.34) is `JR`γ`1`.

push(ζ) Push operations are the most complex, since they involve a pop of the top stack element, and pushing any number, including zero, of new elements. The challenge is in constructing the correct $k+1$-parameter instantiation of `C`, from the current parameters of the type. Each of these $k+1$ is also an instantiation of `C` which may require more such parameters. Even though the number of ingredients is small, the resulting type expressions tend to be excessively long and unreadable.

The predicament is ameliorated a bit by the idea, demonstrated above with auxiliary type `P'` (Figure 2.15), of delegating the task of creating a complex type to an auxiliary generic type. The task of this sidekick is simplified if some of its generic parameters are sub-expressions that recur in the desired result.

Cases in point are $\delta(\gamma_1, \sigma_1) = \text{push}(\gamma_1, \gamma_1, \gamma_2)$, and $\delta(\gamma_2, \sigma_1) = \text{push}(\gamma_2, \gamma_2)$ of Table 2.1. The corresponding sidekick types, (γ`1`σ`1_Push_`γ`1`γ`1`γ`2` and γ`2`σ`1_Push_`γ`2`γ`2`) can be found in lines 36–43 of Figure 2.18. The first of these define the correct return type of σ`1()` in case γ`1` is the top element, the second of σ`2`, in case γ`2` is the top element. Examine now the definition of types `C`γ`1`,`C`γ`2` in the figure, and in particular lines 21–23 and 29–31 which define the list of types they extend. Notice that each extends one of the sidekicks, inheriting the covariant overrides of σ`1()`.

More generally, economy of expression may require that for each case of $\delta(\gamma, \sigma) = \text{push}(\zeta)$ in the transition table, one creates a sidekick type which overrides the appropriate σ`()` function. The appropriate `C`γ type then inherits the definition from the sidekick.

Figure 2.18 Type encoding of jDPDA A (as defined in Table 2.1)

```
class A { // Encode automaton A
  private static class ΣΣ    // Encodes set Σ*, type of reject
    { /* empty */  }
  static class L extends ΣΣ // Encodes set L ⊆ Σ*, type of accept
    { /* empty */  }
  // Configuration of the automaton
  interface C<       // Generic parameters:
    Rest extends C, // The rest of the stack, for pop or jump(γ) operations
    JRγ1 extends C, // Type of  Rest.jump(γ1), may be rest, or anything in it.
    JRγ2 extends C  // Type of  Rest.jump(γ2), may be rest, or anything in it.
  >
  {
    ΣΣ $();          // δ transition on end of input; invalid language by default
    C σ1();          // δ transition on σ1; dead end by default
    C σ2();          // δ transition on σ2; dead end by default
    C σ3();          // δ transition on σ3; dead end by default
    public interface E extends C<¤,¤,¤> { /* Empty stack configuration */ }
    interface ¤ extends C<¤,¤,¤> { /* Error configuration. */ }
    interface Cγ1< // Configuration when γ1 is at top
      Rest extends C, JRγ1 extends C, JRγ2 extends C
    > extends
      C<Rest, JRγ1, JRγ2>
      ,γ1σ1_Push_γ1γ1γ2<Rest,JRγ1,JRγ2,Cγ1<Rest, JRγ1, JRγ2>>
    {
      @Override L $();
    }
    interface Cγ2< // Configuration when γ2 is at top
      Rest extends C, JRγ1 extends C, JRγ2 extends C
    > extends
      C<Rest, JRγ1, JRγ2>
      ,γ2σ1_Push_γ2γ2<Rest,JRγ1,JRγ2>
    {
      @Override Rest σ2();
      @Override JRγ1 σ3();
    }
    interface γ1σ1_Push_γ1γ1γ2<Rest extends C,JRγ1 extends C,JRγ2 extends C,P
     extends Cγ1<Rest, JRγ1, JRγ2 >>{
      // Sidekick of δ(γ1, σ1) = push(γ1, γ1, γ2)
      Cγ2<Cγ1<P, Rest, JRγ2>,P,JRγ2> σ1();
    }
    interface γ2σ1_Push_γ2γ2<Rest extends C,JRγ1 extends C,JRγ2 extends C>{
      // Sidekick of δ(γ2, σ1) = push(γ2, γ2)
      Cγ2<Cγ2<Rest, JRγ1, JRγ2>, JRγ1, Rest> σ1();
    }
  }
    static Cγ1<E,¤,¤> build = null;
}
```

Conclusion The proof of Theorem 1 is an algorithm, taking as input some jDPDA, and returning as output a set of JAVA type definitions. The returned types, allow a call chain $\mathsf{java}(\alpha)$, such that the type of the returned object represents the configuration of the input automaton after reading α. If the automaton rejects after α, then the returned type is the illegal $\Sigma\Sigma$, and if the automaton accpets, the type shall be L.

2.7 The Prefix Theorem

theorem 2. *Let A be a DPDA recognizing a language $L \subseteq \Sigma^*$. Then, there exists a* JAVA *type definition, J_A for types* `L`, `A`, `C` *and other types such that the* JAVA *command*

$$\texttt{C c = A.build.}\mathbf{\mathsf{java}}(\alpha)\texttt{;} \tag{2.3}$$

type checks against J_A if an only if there exists $\beta \in \Sigma^*$ such that $\alpha\beta \in L$ and type `C` *is the configuration of A after reading α. Furthermore, for any such β, Theorem 1 applies such that the* JAVA *command*

$$\texttt{L } \ell \texttt{ = A.build.}\mathbf{\mathsf{java}}(\alpha\beta)\texttt{.\$();} \tag{2.4}$$

always type-checks. Finally, the program J_A can be effectively generated from A.

Informally, a call chain type-checks if and only if it is a prefix of some legal sequence. Alternatively, a call chain won't type-check if there is no continuation that leads to a legal string in L.

The proof resembles Theorem 1's proof. We provide a similar implementation for a jump-stack [10], that will not compile under illegal prefixes.

The main difference between the two theorems is: in Theorem 1 we allowed illegal call chains to compile, but not return the required `L` type, while in Theorem 2 the illegal chain won't compile at all.

Since the code suggested by the proof highly resembles the previously suggested code, mainly the differences will be discussed.

We will use the same running example, defined by Table 2.1.

2.7.1 Main Types

The main types here are a subset of the previously defined main types.

```
static class L // Encodes set L ⊆ Σ*, type of accept
  { /* empty */  }
public interface E { /* Empty stack configuration */ }
interface ¤ { /* Error configuration. */ }
```

First, type $\Sigma\Sigma$ is removed. A call chain that doesn't represent a valid prefix won't compile, thus, there is no need for an error return type such as $\Sigma\Sigma$. Second, `interface C` is removed. Without it, the configuration types won't have the methods σ`1()`, … ,σk`()` and `$()` from the supertype. These inherited methods, is what differentiates the previous proof from the current. Classes `¤` and `E` are defined similarly, except now they don't extend any type.

[10] recall that the two formal constructs are of the same expressiveness

2.7.2 Top-of-Stack Types

Types `C`γ`1`, … ,`C`γk, still represent stacks with γ`1`, … ,γk as their top element, this time, the methods are defined ad-hock, in each type (they are not added in this figure as they are added with the use of sidekicks). In A there are two such types:

```
interface Cγ1< // Configuration when γ1 is at top
  Rest, JRγ1, JRγ2
> extends
  γ1σ1_Push_γ1γ1γ2<Rest,JRγ1,JRγ2,Cγ1<Rest, JRγ1, JRγ2>>
{
}
interface Cγ2< // Configuration when γ2 is at top
  Rest, JRγ1, JRγ2
> extends
  γ2σ1_Push_γ2γ2<Rest,JRγ1,JRγ2>
{
}
static Cγ1<E,¤,¤> build = null;
```

Note, that the type parameters of the former types hasn't changed, since the model we are trying to implement, hasn't changed. These $k+1$ parameters still suffice for our cause.

Figure 2.19 Accepting and non-accepting call chains with the type encoding of jDPDA A (as defined in Table 2.1). All lines in **`accepts`** type-check, and all lines in **`rejects`** cause type errors

```
static void accepts() {
  A.build.$();
  A.build.σ1().σ3().$();
  A.build.σ1().σ2().$();
  A.build.σ1().σ1().σ2().σ3().σ1().σ2().$();
}
static void rejects() {
  A.build.σ1().$();
  A.build.σ2();
  A.build.σ1().σ2().σ3();
  A.build.σ1().σ1().σ2().σ3().σ1().$();
}
```

In Figure 2.19, call chains in the **`accepts()`** method correctly type-checks (i.e., in L), while the chains in **`rejects()`** do not type-check (i.e., these prefixes have no continuation that can lead to a legal word in L), where the last method invocation generates an

> "method … is undefined for the type … "

error message.

The main difference between Figure 2.19 and Figure 2.17 is that there is no need to use an auxiliary function **`isL()`** as in Figure 2.19 since now illegal prefixes do not type-check.

2.7.3 Transitions

Due to the changes we expressed, the transition table is encoded slightly different.

Encoding of the legal operations accept, jump(γ_i) and push(ζ) remains as in Theorem 1, since we want the same behavior for legal call chains. The minor differences are in the illegal operations reject and $\perp$:

reject Since we add the methods ad-hock to each type, the reject entry means that the corresponding type, *won't* have a `$()` method, i.e., type `C`γ`2` doesn't have a method `$()`.

$\perp$ We encounter $\perp$ on the transition function when some input character σ is not allowed for the top of the stack element γ. In that case, the corresponding type `C`γ *must not* have a method for σ, this way, invoking the methods will result in type error. In Table 2.1 a $\perp$ may occur when the top of the stack is γ_1 and the input character is σ_2, thus, no method σ`2` is introduced in type `C`γ`1`.

The use of sidekicks is still allowed and recommended to improve readability of code.

Conclusion In this section, a proof, similar to the one in Section 2.6 is provided. An algorithm was introduced, to not only emulate the running of some jDPDA A, but also to "halt it" in the earliest time possible, i.e., only if there is no legal call chain from this point to result in a legal word in the language of A.

2.8 Notes on Practical Applicability

Theorem 1 and its proof above provide a concrete algorithm for converting an EBNF specification of a fluent API into its realization:

1. Convert the specification into a plain BNF form [11].
2. Convert this BNF into a type of DPDA (using parsing algorithms e.g., LR(k), LALR, LL(k)). This conversion might fail [12].
3. Convert this DPDA into a jDPDA. (Conversion is guaranteed to succeed)
4. Apply the proof to generate appropriate Java type definitions, making sure to augment methods with code to maintain the fluent-call-list. Parsing the fluent-call-list can be done either in each method, or lazily, when the product of the fluent API call chain is to be used.

Although possible, a practical tool that uses the proof directly is a challenge. Part of the problem is the complexity of the algorithms used, some of which, e.g., the DPDA and jDPDA equivalence have never been implemented. Yet another issue that clients of compiler-compiler have grown to expect facilities such as means for resolving ambiguities, manipulation of attributes, etc. Also, for a fluent API to be elegant and useful, it should support method with

[11] http://lampwww.epfl.ch/teaching/archive/compilation-ssc/2000/part4/parsing/node3.html

[12] In the LR case, we know [36] there exists an equivalent grammar for which the conversion will succeed

Figure 2.20 Type encoding of jDPDA A (as defined in Table 2.1) that allow a partial call chain, if and only if, there exists a legal continuation, that leads to a word in L (the language of A)

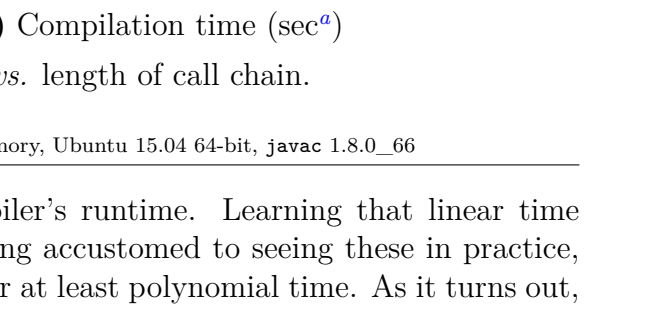
```
static class A { // Encode automaton A
  static class L // Encodes set L ⊆ Σ*, type of accept
    { /* empty */  }
  public interface E { /* Empty stack configuration */ }
  interface ¤ { /* Error configuration. */ }
  // Configuration of the automaton
  interface Cγ1< // Configuration when γ1 is at top
    Rest, JRγ1, JRγ2
  > extends
    γ1σ1_Push_γ1γ1γ2<Rest,JRγ1,JRγ2,Cγ1<Rest, JRγ1, JRγ2>>
  {
    L $();
  }
  interface Cγ2< // Configuration when γ2 is at top
    Rest, JRγ1, JRγ2
  > extends
    γ2σ1_Push_γ2γ2<Rest,JRγ1,JRγ2>
  {
    Rest σ2();
    JRγ1 σ3();
  }
  interface γ1σ1_Push_γ1γ1γ2<Rest,JRγ1,JRγ2,P extends Cγ1<Rest, JRγ1, JRγ2 >>{
    // Sidekick of δ(γ1,σ1) = push(γ1,γ1,γ2)
    Cγ2<Cγ1<P, Rest, JRγ2>,P,JRγ2> σ1();
  }
  interface γ2σ1_Push_γ2γ2<Rest,JRγ1,JRγ2>{
    // Sidekick of δ(γ2,σ1) = push(γ2,γ2)
    Cγ2<Cγ2<Rest, JRγ1, JRγ2>, JRγ1, Rest> σ1();
  }
  static Cγ1<E,¤,¤> build = null;
}
```

parameters whose parameters are also defined by a fluent API: these two APIs may mutually recursive and even the same. Support of these features through four or so algorithmic abstractions may turn out to be a decent engineering task.

Figure 2.21 Exponential compilation time for a simple JAVA program.

```
static interface Cons<Car,Cdr
   >{
  Cons<
    Cons<Car,Cdr>,
    Cons<Car,Cdr>
  > d();
}
```

(a) Encoding of a binary type tree



(b) Compilation time (sec[a]) *vs.* length of call chain.

[a] measured on an Intel i5-2520M CPU @ 2.50GHz ×4, 3.7GB memory, Ubuntu 15.04 64-bit, `javac` 1.8.0_66

Yet another challenge is controlling the compiler's runtime. Learning that linear time parsers and lexical analyzers are possible, and being accustomed to seeing these in practice, one may expect the compiler would run in linear, or at least polynomial time. As it turns out, this time is exponential in the worst case (at least for `javac`). An encoding of a S-expression in type `Cons` (Figure 2.21(a)) is a not terribly complex such worst case.

Type `Cons` takes two type parameters, `Car` and `Cdr` (denoting left and right branches). Denote the return type of `d()` by

$$\tau = \texttt{Cons< Cons<Car, Cdr>, Cons<Car, Cdr> >}.$$

Let σ denote the type of the `this` implicit parameter to `d`. Now, since $\tau = \texttt{Cons<}\sigma,\sigma\texttt{>}$, we have $|\tau| \geq 2|\sigma|$, where the size of a type is measured, e.g., in number of characters in its textual representation. Therefore, in a chain of n calls to `d()`

$$\texttt{(Cons<?,?>(null))}.\overbrace{\texttt{d()}.\cdots.\texttt{d()}}^{n \text{ times}}\texttt{;} \tag{2.5}$$

the size of the resulting type is $O(2^n)$.

Figure 2.21(b) shows, on the doubly logarithmic plane, the runtime (on a Lenovo X220) of the `javac` compiler (version 1.8.0_66) in face of a JAVA program assembled from Figure 2.21 and Eq. (2.5) placed as the single command of `main()`. Exponential growth is demonstrated by the right-hand side of the plot, in which curve converges on a straight line. (In fact, a variation of the construction may lead to even super-exponential growth rate of the size of types.)

We believe that this exponential growth is due to a design flaw in the compiler. Had the compiler used a representation of types that allows sharing of expression types, compilation time would be linear.

Still, with current compiler technology, the type encoding scheme demonstrated in Figure 2.18 might not be scalable.

Chapter 3

An Algorithm for Generating Fluent APIs for Java

Based on an article submitted to OOPSLA'16 [24]

3.1 Type States and a Fluent API Example

Fluent APIs are related to the topic of type-states. There is a large body of research on *type-states* (see e.g., these review articles [2,8]) Informally, an object that belongs to a certain type (`class` in the object oriented lingo), has type-states, if not all methods defined in this object's class are applicable to the object in all states it may be in.

File object is the classical example: It can be in one of two states: "open" or "closed". Invoking a `read()` method on the object is only permitted when the file is in an "open" state. In addition, method `open()` (respectively `close()`) can only be applied if the object is in the "closed" (respectively, "open") state.

A recent study [7] estimates that about 7.2% of JAVA classes define protocols definable in terms of type-states. This non-negligible prevalence raise two challenges:

1. ***Identification.*** Frequently, type-state receive little or no mention at all in the documentation. The challenge is in identifying the implicit type state in existing code.

   Specifically, given an implementation of a class (or more generally of a software framework), *determine* which sequences of method calls are valid and which violate the type state requirement presumed by the implementation.

2. ***Maintenance and Enforcement.*** Having identified the type-states, the challenge is in automatically flagging out illegal sequence of calls that does not conform with the type-state.

   Part of this challenge is maintenance of these automatic flagging mechanisms as the type-state specification of the API evolves.

3.1.1 A Type State Example

An object of type `Seat`[1] is created in the `down` state, but it can then be `raise`d to the `up` state, and then be `lower`ed to the `down` state. Such an object can be used by two kinds of

[1] example inspired by earlier work of Richard Harter on the topic [29].

users, **male**s and **female**s, for two distinct purposes: **urinate** and **defecate**.

Thus, there are a total of six methods that might be invoked on an instance of **Seat**:

```
male()    raise()  urinate()
female()  lower()  defecate()
```

A fluent API design specifies the order in which such calls can be made. For example, a fluent API should recognize the sequences of Figure 3.1 as being type correct.

Figure 3.1 Legal sequences of calls in the toilette seat example

```
new Seat().male().raise().urinate();
new Seat().female().urinate();
```

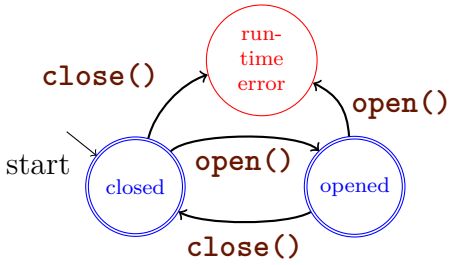
At the same time, illegal sequences as made in Figure 3.2 should be signaled as type errors.

Figure 3.2 Illegal sequences of calls in the toilette seat example

```
new Seat().female().raise();
new Seat().male().raise().defecate();
new Seat().male().male();
new Seat().male().raise().urinate().female().urinate();
```

To generate a fluent API that meets these requirements, one must observe that the protocol of a **Seat** is defined completely by the DFA depicted in Figure 3.3.

Figure 3.3 A deterministic finite automaton realizing the type states of class **Seat**.



Having constructed the automaton, generating the classes and methods is immediate by following the *encoding convention*:

1. A state q in the automaton is encoded as an **interface**.[2]

2. If there is an arc labeled ℓ leading from q_i to q_j, then **interface** q_i defines a method whose return type is **interface** q_j.[3]

3. The initial state q_0 is made special in being the only type from which a fluent API call chain can start.[4]

[2]The name of this interface is arbitrary; all that it is required is that names assigned to distinct states are distinct. For readability sake, there is often an attempt to choose a name which makes a close approximation of the name of q, within the limits of the language's syntax and Unicode vocabulary.

[3]Again, the name of this method is often selected as an approximation of the name of ℓ.

[4]This is achieved by generating a class c with **public** constructor that **implements** the **interface** encoding type q_0. If no other interfaces do not have a similar c, then clients can only start a fluent API chain by creating an instance of c, which is effectively the state q_0.

The result of employing these rules to our toilette example is depicted at Figure 3.4

Figure 3.4 A fluent API realizing the toilette seat object defined as a DFA in Figure 3.3

```
public static interface Q0 {
  Q1 female();
  Q6 male();
  void $();
}
public static interface Q1 {
  Q0 urinate();
  Q0 defecate();
}
public static interface Q2 {
  Q1 lower();
}
public static interface Q3 {
  Q2 female();
  Q4 male();
  void $();
}
public static interface Q4 {
  Q6 lower();
  Q3 urinate();
}
public static interface Q5 {
  Q3 urinate();
}
public static interface Q6 {
  Q5 raise();
  Q0 defecate();
}
public static class Seat implements Q0 {
  @Override public Q1 female() { /* ... */ }
  // ...
}
```

Using the implementation in Figure 3.4 we achieved our goals regarding the legal usage examples in Figure 3.1 and the illegal usage examples in Figure 3.2. So far.

It should be clear that the type checking engine of the compiler can be employed to distinguish between legal and illegal sequences.

3.2 Introducing Fajita

There were a total of six methods that might be invoked in the example of the previous section:

Figure 3.5 A BNF grammar for the toilette seat problem

⟨*Visitors*⟩ ::= ⟨*Down-Visitors*⟩
⟨*Down-Visitors*⟩ ::= ⟨*Down-Visitor*⟩ ⟨*Down-Visitors*⟩
| ⟨*Raising-Visitor*⟩ ⟨*Up-Visitors*⟩
| ϵ
⟨*Up-Visitors*⟩ ::= ⟨*Up-Visitor*⟩ ⟨*Up-Visitors*⟩
| ⟨*Lowering-Visitor*⟩ ⟨*Down-Visitors*⟩
| ϵ
⟨*Up-Visitor*⟩ ::= male() urinate()
⟨*Down-Visitor*⟩ ::= female() ⟨*Action*⟩
| male() defecate()
⟨*Raising-Visitor*⟩ ::= male() raise() urinate()
⟨*Lowering-Visitor*⟩ ::= female() lower() ⟨*Action*⟩
| male() lower() defecate()
⟨*Activity*⟩ ::= urinate()
| defecate()

```
male()   raise()  urinate()
female() lower()  defecate()
```

Figure 3.5 is a BNF specification of the order in which they might be invoked.

FAJITA takes this grammar specification as input, and in response generates the corresponding JAVA type hierarchy. (Our proof-of-concept implementation does not yet deal with the construction of an AST from a concrete method call chain.)

The FAJITA specification is made with a JAVA fluent API in the form of context-free grammar. The grammar for out toillete example is depicted in Figure 3.5.

To define grammars, one must first define the set of *grammar terminals*

```
enum ToiletteTerminals implements Terminal {
  male, female,
  urinate, defecate,
  lower, raise;
}
```

We are also required to define the set of *grammar variables*

```
enum ToiletteVariables implements Variable {
  Visitors, Down_Visitors, Up_Visitors,
  Up_Visitor, Down_Visitor,
  Lowering_Visitor, Raising_Visitor,
  Activity
};
```

Figure 3.6 A BNF grammar for the types state example

```
new BNF()
  .with(ToiletteTerminals.class)
  .with(ToiletteSymbols.class)
  .start(Visitors)
  .derive(Visitors).to(Down_Visitors)
  .derive(Down_Visitors)
    .to(Down_Visitor).and(Down_Visitors)
    .or(Raising_Visitor).and(Up_Visitors)
    .orNone()
  .derive(Up_Visitors)
    .to(Up_Visitor).and(Up_Visitors)
    .or(Lowering_Visitor).and(Down_Visitors)
    .orNone()
  .derive(Up_Visitor).to(male).and(urinate)
  .derive(Down_Visitor)
    .to(female).and(Action)
    .or(male).and(defecate)
  .derive(Raising_Visitor).to(male).and(raise).and(urinate)
  .derive(Lowering_Visitor)
    .to(female).and(lower).and(Action)
    .or(male).and(lower).and(defecate)
  .derive(Activity)
    .to(urinate)
    .or(defecate)
  .go();
```

Once the set of terminals and nonterminals are fixed, the grammar can be defined, in a fluent API generated by FAJITA itself as shown in Figure 3.6.

The call to function **go()** (last line in Figure 3.6) instructs FAJITA to generate the code for the fluent API specified by the subsequent part of the expression.

Nonterminals are translated to classes while terminals are translated to methods which take no parameters.

Library classes such as **String** and **Integer**, just as user-defined classes such as **Invoice** may be used as well. FAJITA generates class definitions only for classes whose name is declared in an **enum** which is passed to the **with** function in the BNF declaration. Recall also that methods that take a parameters (Figure 1.2) can be used as tokens as well.

Even though FAJITA is not at production level, we plan on submitting it to artifact evaluation. The current BNF specification of the tool is given in Figure 3.7 (which incidentally can be written in FAJITA fluent API iteself)

3.3 LL Parsing and Realtime Constraints

Formal presentation of the algorithm that compiles an LL(1) grammar into an implementation of a language recognizer with JAVA generics is delayed to Section 3.4. This section gives an

Figure 3.7 A BNF grammar for defining BNF grammars

```
⟨BNF⟩ ::= ⟨Header⟩ ⟨Body⟩ ⟨Footer⟩
⟨Header⟩ ::= ⟨Variables⟩ ⟨Terminals⟩
           | ⟨Terminals⟩ ⟨Variables⟩
⟨Variables⟩ ::= with(Class<? extends Variable>)
⟨Terminals⟩ ::= with(Class<? extends Terminal>)
⟨Body⟩ ::= ⟨Start⟩ ⟨Rule⟩ ⟨Rules⟩
⟨Start⟩ ::= start(⟨Variable⟩ )
⟨Rules⟩ ::= ⟨Rule⟩ ⟨Rules⟩
           | ϵ
⟨Rule⟩ ::= derives(⟨Variable⟩ ) ⟨Conjunctions⟩
⟨Conjunctions⟩ ::= ⟨First-Conjunction⟩ ⟨Conjunctions⟩
⟨First-Conjunction⟩ ::= to(⟨Symbol⟩ ) ⟨Symbols⟩
           | toNone()
⟨Conjunctions⟩ ::= ⟨Conjunction⟩ ⟨Conjunctions⟩
           | ϵ
⟨Conjunction⟩ ::= or(⟨Symbol⟩ ) ⟨Symbols⟩
           | orNone()
⟨Symbols⟩ ::= ϵ
           | and(⟨Symbol⟩ ) ⟨Symbols⟩
⟨Symbol⟩ ::= Variable
           | Terminal
           | Terminal ⟨Parameters⟩
⟨Parameters⟩ ::= , ⟨Existing-Class⟩ ⟨Parameters⟩
           | ϵ
⟨Footer⟩ ::= go()
```

intuitive perspective on this algorithm.

We will first recall the essentials of the classical LL(1) parsing algorithm (Section 3.3.1). Then, we will explain the limitations of the computational model offered by JAVA generics (Section 3.3.2).

The discussion proceeds to the observation that underlies our emulation of the parsing algorithm within these limitations.

Building on all these, Section 3.3.3 can make the intuitive introduction to the main algorithm. To this end, we revise here the classical algorithm [39] for converting an LL grammar (in writing LL, we, here and henceforth really mean LL(1)), and explain how it is modified to generate a recognizer executable on the computational model of JAVA generics.

3.3.1 Essentials of LL parsing

The traditional ***LL Parser*** (LL_p) is a DPDA allowed to peek at the next input symbol. Thus, δ, its transition function, takes two parameters: the current state of the automaton, and a peek into the terminal next to be read. The actual operation of the automaton is by executing in loop the step depicted in Algorithm 3.1.

Algorithm 3.1 The iterative step of an LL_p

```
1:  Let X ← pop()                                   // what's next to parse?
2:  If X ∈ Σ                                        // anticipate to match terminal X
3:    If X ≠ next()                                 // read terminal is not anticipated X
4:      Reject                                      // automaton halts in error
5:    else                                          // terminal just read was the anticipated X
6:      Continue                                    // restart, popping a new X, etc.
7:  else if X = $                                   // anticipating end-of-input
8:    If $ ≠ next()                                 // not the anticipated end-of-input
9:      Reject                                      // automaton halts in error
10:   else                                          // matched the anticipated end-of-input
11:     Accept                                      // all input successfully consumed
12: else                                            // anticipated X must be a nonterminal
13:   Let R ← δ(X, peek())                          // determine rule to parse X
14:   If R = ⊥                                      // no rule found
15:     Reject                                      // automaton halts in error
16:   else                                          // A rule was found
17:     Let (Z ::= Y_1, ..., Y_k) ← R               // break into left/right
18:     assert Z = X                                // δ constructed to return valid R only
19:     print(R)                                    // rule R has just been applied
20:     For i = k, ..., 1, push(Y_i)                // push in reverse order
21:     Continue                                    // restart, popping a new X, etc.
```

1. Input is a string of terminals drawn from alphabet Σ, ending with a special symbol $\$ \notin \Sigma$.
2. Stack symbols are drawn from $\Sigma \cup \{\$\} \cup \Xi$, where Ξ is the set of nonterminals of the grammar from which the automaton was generated.
3. Functions $\mathsf{pop}(\cdot)$ and $\mathsf{push}(\cdot)$ operate on the pushdown stack; function $\mathsf{next}()$ returns and consumes the next terminal from the input string; function $\mathsf{peek}()$ returns this terminal without consuming it.
4. The automaton starts with the stack with the start symbol $S \in \Xi$ pushed into its stack.

The DPDA maintains a stack of "anticipated" symbols, which may be of three kinds: a terminal drawn from the input alphabet Σ, an end-of-input symbol \$, or one of Ξ, the set of nonterminals of the underlying grammar.

If the top of the stack is an input symbol or the special, end-of-input symbol \$, then it must match the next terminal in the input string, the matched terminal is then consumed from the input string. If there is no match, the parser rejects. The parser accepts if the input is exhausted with no rejections (i.e., \$ was matched).

The more interesting case is that X, the popped symbol is a nonterminal: the DPDA peeks into the next terminal in the input string (without consuming it). Based on this terminal, and X, the transition function δ determines R-the derivation rule applied to derive X. The algorithm rejects if δ can offer no such rule. Otherwise, it pushes into the stack, in reverse order, the symbols found in the right hand side of R.

3.3.2 LL(1) Parsing with Java Generics?

Can Algorithm 3.1 be executed on the machinery available with JAVA generics? As it turns out, most operations conducted by the algorithm are easy. The implementation of function δ can be found in the toolbox in [25]. Similarly, any fixed sequence of push and pop operations on the stack can be conducted within a JAVA transition function: the "**For**" at the algorithm can be unrolled.

Superficially, the algorithm appears to be doing a constant amount of work for each input symbol. A little scrutiny falsifies such a conclusion.

In the case $R = X ::= Y$, Y being a nonterminal, the algorithm will conduct a $\mathsf{pop}(\cdot)$ to remove X and a $\mathsf{push}(\cdot)$ operation for Y before consuming a terminal from the input. Further, δ may return next the rule $Y ::= Z$, and then the rule $Z ::= Z'$, etc. Let k' be the number of such substitutions occurring while reading a single input token.

Definition 3 (k' - subtitution factor)**.** *Let A be a nonterminal at the top of the stack and t be the next input symbol. If $t \in \mathsf{First}(A)$ then k' is the number of consecutive substitutions the parser will perform (replacing the nonterminal at the top the stack with one of it's rules) until t will be consumed from the input.*

Also, in the case $R = X ::= \epsilon$ the DPDA does not push anything into the stack. Further, in its next iteration, the DPDA conducts another pop,

$$X' \leftarrow \mathsf{pop}(),$$

instruction and proceeds to consider this new X'. If it so happens, it could also be the case that the right hand side of rule R'

$$R' = \delta(X', \mathsf{peek}())$$

is once again empty, and then another $\mathsf{pop}()$ instruction occurs

$$X'' \leftarrow \mathsf{pop}(),$$

etc. Let k^* be the number of such instructions in a certain such mishap.

Definition 4 (k^* - pop factor)**.** *Let X be a nonterminal at the top of the stack and t be the next input symbol. Then k^* is the number of consecutive substitutions the parser will perform* of only ϵ rules *until t will be consumed from the input.*

The cases $k' > 0$ and $k^* > 0$ are not rarities. For example, let us concentrate on the grammar depicted in Figure 3.8. This grammar, inspired by the prototypical specification of PASCAL [51][5], shall serve as running example.

Figure 3.8 An LL(1) grammar over the alphabet

$$\Sigma = \left\{ \begin{array}{l} \texttt{program}, \texttt{begin}, \texttt{end}, \\ \texttt{label}, \texttt{const}, \texttt{id}, \\ \texttt{procedure}, \texttt{;}, \texttt{()} \end{array} \right\}.$$

(inspired by the original PASCAL grammar; to serve as our running example)

$$\begin{array}{rcl}
\langle Program \rangle & ::= & \text{program id } \langle Parameters \rangle \text{ ; } \langle Definitions \rangle \ \langle Body \rangle \\
\langle Body \rangle & ::= & \text{begin end} \\
\langle Definitions \rangle & ::= & \langle Labels \rangle \ \langle Constants \rangle \ \langle Nested \rangle \\
\langle Labels \rangle & ::= & \epsilon \mid \text{label } \langle Label \rangle \ \langle MoreLabels \rangle \\
\langle Constants \rangle & ::= & \epsilon \mid \text{const } \langle Constant \rangle \ \langle MoreConstants \rangle \\
\langle Label \rangle & ::= & \text{;} \\
\langle Constant \rangle & ::= & \text{;} \\
\langle MoreLabels \rangle & ::= & \epsilon \mid \langle Label \rangle \ \langle MoreLabels \rangle \\
\langle MoreConstants \rangle & ::= & \epsilon \mid \langle Constant \rangle \ \langle MoreConstants \rangle \\
\langle Nested \rangle & ::= & \epsilon \mid \langle Procedure \rangle \ \langle Nested \rangle \\
\langle Procedure \rangle & ::= & \text{procedure id } \langle Parameters \rangle \text{ ; } \langle Definitions \rangle \\
 & & \langle Body \rangle \\
\langle Parameters \rangle & ::= & \epsilon \mid \text{()}
\end{array}$$

`program id ; begin end`
`program id () ; label ; begin end`
`program id () ; label ; ; ; ; const ; begin end`
`program id ; procedure id ; procedure id ;` `begin end begin end begin end`

Table 3.1: Legal words in the language defined by Figure 3.8

The grammar preserves the "theme" of nested definitions of PASCAL, while trimming these down as much as possible. Table 3.1 presents some legal sequences derived by this grammar.

In order to demonstrate the problems with k' and k^* we first supply the $\mathsf{First}(\cdot)$, and $\mathsf{Follow}(\cdot)$ sets of our example grammar.

The $\mathsf{First}(\cdot)$ set of symbol X is the set of all terminals that might appear at the beginning of a string derived from X (algorithm for computing $\mathsf{First}(\cdot)$ is presented in Algorithm 3.2), thus, for example, if X is a terminal then its first set is the set containing only X. The $\mathsf{First}(\cdot)$ set is extended for strings too, $\mathsf{First}(\alpha)$ contains all terminals that can begin a string derived from α (Appropriate algorithm can be found at Algorithm 3.3).

[5] http://www.fit.vutbr.cz/study/courses/APR/public/ebnf.html

The $\mathsf{Follow}(\cdot)$ set of nonterminal A is the set of all terminals that might appear immediately after a string that was derived from A (Appropriate algorithm can be found at Algorithm 3.4).

Algorithm 3.2 An algorithm for computing $\mathsf{First}(X)$ for each grammar symbol X in the input grammar $G = \langle \Sigma, \Xi, P \rangle$

```
For A ∈ Ξ                                          // initialize First(·) for all nonterminals
  First(A) = ∅
For t ∈ Σ                                          // initialize First(·) for all terminals
  First(t) = {t}
While No changes in any First(·) set
  For X ::= Y_0 ... Y_k ∈ P                        // for every production
    First(X) ∪= First(Y_0)                         // add First(Y_0) to First(X)
    For i ∈ {0 ... k}                              // for each symbol in the RHS
      If Nullable(Y_0 ... Y_{i-1})                 // if the prefix is nullable
        First(X) ∪= First(Y_i)                     // add First(Y_i) to First(X)
```

Algorithm 3.3 An algorithm for computing $\mathsf{First}(\alpha)$ for some string of symbols α. This algorithm relies on results from Algorithm 3.2

```
Let Y_0 ... Y_k ← α                                // break α into its symbols
First(α) ∪= First(Y_0)                             // initialize First(α) with First(Y_0)
For i ∈ {1 ... k}                                  // for each symbol in the string
  If Nullable(Y_0 ... Y_{i-1})                     // if the prefix is nullable
    First(α) ∪= First(Y_i)                         // add First(Y_i) to First(α)
```

Algorithm 3.4 An algorithm for computing $\mathsf{Follow}(A)$ for each nonterminal X in the input grammar $G = \langle \Sigma, \Xi, P \rangle$ when $\$ \notin \Sigma$ and $S \in \Sigma$ is the start symbol of G

```
Follow(S) = {$}                                    // initialize the start symbol
While No changes in any Follow(·) set
  For A ::= Y_0 ... Y_k ∈ P                        // for each grammar rule
    For i ∈ {0 ... k}                              // for each symbol in the RHS
      If Y_i ∉ Ξ                                   // compute only for nonterminals
        continue
      Follow(Y_i) ∪= First(Y_{i+1} ... Y_k)
      If Nullable(Y_{i+1} ... Y_k)                 // if the suffix is nullable
        Follow(Y_i) ∪= First(A)                    // add First(A)
```

For example, the nonterminal ⟨*Definitions*⟩ can be derived to a string beginning with **`label`** (if there are defined labels), or **`const`** (if there are no labels defined), or **`procedure`** (if there are no labels or constants defined). $\mathsf{Follow}(\langle \textit{Definitions} \rangle)$ on the other hand, contains only **`begin`** since ⟨*Definitions*⟩ is always followed by ⟨*Body*⟩, and $\mathsf{First}(\langle \textit{Body} \rangle)$ contains only **`begin`**.

The $\mathsf{First}(\cdot)$ and $\mathsf{Follow}(\cdot)$ sets for our running example (Figure 3.8) are listed in Table 3.2.

In our example, nonterminal ⟨*Procedure*⟩ has only one derivation rule, which begins with the terminal **`procedure`**. Thus, all derivations of ⟨*Procedure*⟩ must begin with terminal **`procedure`** and the $\mathsf{First}(\cdot)$ set of ⟨*Procedure*⟩ is contains only **`procedure`**, as can be seen

Nonterminal	First(·)	Follow(·)
⟨*Program*⟩	`program`	∅
⟨*Labels*⟩	`label`	`const`, `procedure`, `begin`
⟨*Constants*⟩	`constant`	`procedure`, `begin`
⟨*Label*⟩	`;`	`;`, `const`, `procedure`, `begin`
⟨*MoreLabels*⟩	`;`	`const`, `procedure`, `begin`
⟨*Constant*⟩	`;`	`;`, `procedure`, `begin`
⟨*MoreConstants*⟩	`;`	`procedure`, `begin`
⟨*Nested*⟩	`procedure`	`begin`
⟨*Procedure*⟩	`procedure`	`procedure`, `begin`
⟨*Definitions*⟩	`label`, `const`, `procedure`	`begin`
⟨*Body*⟩	`begin`	`procedure`, `begin`
⟨*Parameters*⟩	`()`	`;`

Table 3.2: Sets First(·) and Follow(·) for nonterminals of the grammar in Figure 3.8

in Table 3.2. The Follow(·) set of ⟨*Procedure*⟩ contains two terminals. The first is `procedure`, because the nonterminal ⟨*Procedure*⟩ is followed by ⟨*Nested*⟩ in ⟨*Nested*⟩'s rule, and ⟨*Nested*⟩ can begin only with `procedure`. The second is `begin`, because ⟨*Nested*⟩ is nullable, and `begin` appears in the Follow(·) set of ⟨*Nested*⟩.

Examples for k' for the Pascal grammar fragment in Figure 3.8

Consider a state of the parser, in which ⟨*Definitions*⟩ is on the top of the top of the stack, and the next token on the input string is `label`. Until that token is consumed, the parser will:

1. pop ⟨*Definitions*⟩ from the stack, and push the nonterminals ⟨*Labels*⟩, ⟨*Constants*⟩, and ⟨*Nested*⟩ in reversed order.

2. pop ⟨*Labels*⟩ and push the symbols `label`, ⟨*Label*⟩, and ⟨*MoreLabels*⟩ in reversed order.

3. match the top of the stack `label` with the input symbol `label` and consume it.

Since we first substituted ⟨*Definitions*⟩, then ⟨*Labels*⟩ and only then consumed the input symbol, $k' = 2$.

The problem is that every such operation is an ϵ-move of the parser's DPDA, and as was shown in [25] it causes problems when we want to employ Java's compiler to "run" our automaton.

Another case is when encountering ⟨*Nested*⟩ on the top of the stack and `procedure` in the input string. $k' = 2$ again, where at first the non-ϵ rule of ⟨*Nested*⟩ will be pushed, then the non-ϵ rule of ⟨*Procedure*⟩ will be pushed, and only then will the terminal `procedure` will match, and be consumed from the input string.

Examples of k^* for the Pascal grammar fragment in Figure 3.8

Consider a state in which the parser is in, where the only rule of $\langle Program \rangle$ is being processed, $\langle Definitions \rangle$ was just replaced on the stack by it's only rule, and $\langle Body \rangle$ is below. Assume also, that the first terminal in the input string is `begin`.

Until the next input token will be consumed, the following will happen:

1. consulting with the transition function δ, $\langle Labels \rangle$'s ϵ-rule will be chosen, thus, $\langle Labels \rangle$ will be popped from the stack.

2. the same will happen with $\langle Constants \rangle$ and $\langle Nested \rangle$.

3. only after seeing nonterminal $\langle Body \rangle$ shall the right rule will be pushed, and the token matched.

In this case, $k^* = 3$ because we had to pop three nonterminals and "replace" them with their matching rules, that are all ϵ-rules before we got to a nonterminal that will derive to something other then ϵ.

In the last example k^* was determined by the size of $\langle Definitions \rangle$, but that doesn't have to be the case all the time. For example, if in our grammar, in $\langle Program \rangle$'s rule, nonterminal $\langle Parameters \rangle$ would follow $\langle Definitions \rangle$, then in the last example, after popping $\langle Labels \rangle$, $\langle Constants \rangle$ and $\langle Nested \rangle$, nonterminal $\langle Parameters \rangle$ would also be popped for similar reasons, and k^* would have been 4 in this case.

Again, the problem relies with the multiple pops that occur without reading the input.

3.3.3 LL(1) parser generator

The LL(1) parser is based on a prediction table. For a given nonterminal as the top of the stack, and an input symbol, the prediction table provides the next rule to be parsed. The parser generator fills prediction table, leaving the rest to the parsing algorithm in Algorithm 3.1 The prediction table construction is depicted in Algorithm 3.5

Algorithm 3.5 An algorithm for filling the prediction table $\mathsf{predict}(A, b)$ for some grammar $G = \langle \Sigma, \Xi, P \rangle$ and an end-of-input symbol $\$ \notin \Sigma$, and where $A \in \Xi$ and $b \in \Sigma \cup \{\$\}$.

```
For r ∈ P                                          // for each grammar rule
  Let A ::= α ← r                                  // break r into its RHS & LHS
  For t ∈ First(α)                                 // for each terminal in First(α)
    If predict(A, t) ≠ ⊥                           // is there a conflict?
      Error                                        // grammar is not LL(1)
    predict(A, t) = r                              // set prediction to rule r
  If Nullable(α)                                   // α might derive to ϵ
    For t ∈ Follow(A)                              // for each terminal in Follow(A)
      If predict(A, t) ≠ ⊥                         // is there a conflict?
        Error                                      // grammar is not LL(1)
      predict(A, t) = r                            // set prediction to rule r
```

3.4 Generating a Realtime Parser from an LL Grammar

The main effort of this section is in explaining the algorithm for converting an LL grammar into RLL_p, a parsing automaton equipped with a stack that has the unique quality of spending constant time on each read input. This realtime property is crucial to the implementation language recognizers with JAVA types implementation.

When this is achieved, all that remains is the "compilation"' of the RLL_p into JAVA types. The challenging part of this translation is dealing with the "JSM", a data structure on which the RLL_p relies.

This data structure has been used before, e.g., for efficient management of the runtime environment in programming languages with dynamic scoping, i.e., languages such as LISP [27], in which name resolution is with respect to previous bindings made on the run time stack (disregarding the usual scoping and nesting rules in languages such as PASCAL).

Interestingly, the JSM data structure is just another name for the "*Jump Deterministic PushDown Automata*" (JDPDA), a theoretical model used in the study of automata and formal languages (see e.g., [13, 40]). And, Incidentally, this JDPS is the one used by Gil and Levy [25], in their proof that the JAVA type checker can recognize DCFG languages. Using their construction, the translation to JAVA is rather mechanical.

3.4.1 The Realtime LL Parser

The ***R**ealtime **L**eft-to-right **L**eftmost-derivation Parser* (RLL_p), is a variant of the famous LL(1) parser [39]. The adjective realtime is to claim that:

- an RLL_p examines its input only after consuming it, and,
- an RLL_p conducts at most one (potentially extended) stack operation in each step.

An extended stack operation is either a $\mathsf{push}(\alpha)$ of a string of symbols, or a long $\mathsf{jump}(\cdot)$ into the stack position denoted by its argument, involving an unbounded number of $\mathsf{pop}()$ operation.

The stack symbols of an RLL_p are *items*, where an item is pair of a grammar rule and a "*dot*", written as

$$A ::= Y_0 \dots Y_{i-1} \cdot Y_i \dots Y_k$$

Formally, the dot is an integer, ranging from 0 to the length of the right-hand side of the rule. But it is better to think of it as a notation for the prefix of this right-hand side.

An item represents these precise moments in the analysis process of the input in which:

- all symbols included in this prefix have been successfully parsed, and,
- all symbols that lie after this prefix, are awaiting their turn to be parsed.

Items can be thought of as a generalization of the stack symbols of an LL_p (recall that these are the grammar's terminals and nonterminals). LL_p stack symbols are markers of the next symbol to be read or parsed. RLL_p stack symbols store this information as well: The next symbol to be read or parsed, is simply the symbol that follows that dot. The generalization is in adding to this symbol the context of containing rule and the point of derivation within it.

Just like the LL_p, the RLL_p is a stack automaton equipped with a prediction table, whose next action is a function (realized in the prediction table) of the next input symbol and the item present at the stack's top.

The RLL_p is initialized with the stack containing an item denoting the degenerate prefix of a rule for deriving the start symbol. In case there are more than one such rule, the automaton selects the rule dictated by the first input token. (This rule is uniquely determined since the grammar is LL.)

After this initialization, the RLL_p proceeds following the instructions in Algorithm 3.6.

Algorithm 3.6 A high level sketch of the iterative step of an RLL_p

```
1:  Let [X ::= α · Yβ] ← pop()                    // retrieve parsing state
2:  Let t ← next()                                // examine input once per iteration
3:  If |Yβ| = 0                                   // was rule fully parsed?
4:    If X is not the start symbol
5:      jump(t)                                   // pop this, potentially other items
6:      continue                                  // restart, popping a new item
7:    If t = $                                    // X must be the start symbol
8:      Accept                                    // start symbol fully parsed
9:    else                                        // start symbol parsed, but not all input consumed
10:     Reject                                    // RLL_p halts in error
11: If Y ∈ Σ                                      // Y is a terminal
12:   If Y ≠ t                                    // read terminal is not anticipated Y
13:     Reject                                    // RLL_p halts in error
14:   else                                        // anticipated terminal found on input, proceed
15:     push(X ::= αY · β)                        // push item with dot advanced
16:     continue                                  // restart, popping this item, and parsing β
17: assert Y ∈ Ξ                                  // Y must be a nonterminal
18: Let a ← Δ(Y, t)                               // Δ(Y, t) says what to do with Y
19: If a = ⊥                                      // input t was unanticipated
20:   Reject                                      // RLL_p halts in error
21: If a = I_1, ..., I_ℓ                          // a string of ℓ items to push
22:   push(I_1, ..., I_ℓ)                         // note, this is a single push operation
23:   continue                                    // restart with a somewhat deeper stack
24: assert a represents a jump                    // t indicates that Yβ ⇒* ϵ
25: jump(t)                                       // pop this, potentially other items
```

Comparing Algorithm 3.6 with the LL-parsing algorithm (Algorithm 3.1), we see that they both begin with popping a stack symbol. More similarities are apparent, after observing that the next symbol to read or parse, denoted by X in Algorithm 3.1 is obtained by extracting the symbol Y that follows the dot in the item X in Algorithm 3.6.

In some ways, our RLL_p emulates an LL_p except that it attaches to each symbol the rule in which it was found and the location within this rule. (Thus, a push of single item is equivalent to the loop of push operations in line 20 in Algorithm 3.1).

Function $\Delta(\cdot, \cdot)$, the transition function of an RLL_p is also a bit more general, and may command the algorithm to push a sequence of stack symbols (items), or carry out a jump into the stack.

3.4.2 The Jump Stack Map Structure

We still need to explain how the jump operations of RLL_p (lines 5 and 25 in Algorithm 3.6) are implemented. For this purpose, we construct here the ***J**ump **S**tack **M**ap* (JSM).

Let S_0 be the implicit stack of items used by the RLL_p, and suppose that this stack is implemented as a singly linked list of nodes of an appropriate type.

The top item of a stack (and in particular S_0) is represented by a pointer, called the "*top pointer*". This pointer can be used for pushing or popping from the list. It can also be used for pushing into the stack a pre-made string of items as done in line 22 of the RLL_p algorithm.

Storing pointers into designated items that lie deeper in the stack makes it possible to make a direct jump into these items. We call these pointers "*jump pointers*" and use the letter J to denote the type of these.

Let us now build on top of S_0 an abstract data type D, which can thought of as a stack of dictionaries, supporting ordinary push and pop operations:

$$D = d_n \cdots d_1\$$$

where d_i, $i = 1 \ldots n$ is a (partial) map of the type $\Sigma \nrightarrow J$ (Σ being our alphabet, d_n being the top of the stack and d_1 the deepest item in it).

Sending query $\mathsf{get}(t)$ to D, $t \in \Sigma$ returns the J value associated with t in the top most (maximal i) dictionary d_i, which satisfies $t \in d_i$. The query returns $\perp$ if t is not found in any of the d_is.

There is an efficient implementation of D in which $\mathsf{get}(t)$ is $O(1)$ time, regardless of the depth at which t is found in D, while keeping updates of the form $\mathsf{push}(d)$ or $d \leftarrow \mathsf{pop}()$ in $O(|d|)$ time ($|d|$ being the size of the partial function).

In fact, since D has the same semantics as that of the stack of binding in dynamically scoped language[6] definitions (such as TEX [37]), we can rely on the classical method [46] for implementing these

1. Maintain a hash table H mapping each $t \in \Sigma$ to a *stack* $s(t)$ of values of type J. A search for key $t \in \Sigma$, then returns in $O(1)$ time the value at the top of $s(t)$ or $\perp$ if stack $s(t)$ is empty.

2. The call $\mathsf{push}(d)$, $|d| = m$ is implemented in $O(m)$ time. Let

   $$d = \{(t_1, j_1), \ldots, (t_m, j_m)\}.$$

   Then iterate over the pairs (t_i, p_i), $i = 1, \ldots, m$, pushing p_i to stack $s(t_i)$.

   The call $\mathsf{push}(d)$ terminates by pushing d itself (represented, say, as linked list) as a single item into an auxiliary stack, to be denoted S_1.

3. Thus, abstract data type D is implemented as the pair $\langle S_1, H \rangle$.

   When D pops a map d, it uses S_1 to locate d, and just before returning it, the implementation of D uses this d to pop an element from the set $s(d_i)$ of stacks in H,

   $$s(d_i) = \{s(t) \mid t \in d_i\}.$$

[6] a dictionary d_i is a collection of names and their binding to nameable entities defined in the scope, which hide the definitions made in containing scopes

Let us now generalize D so that it also supports jumps into it. To do so, let clients of D store pointers to designated dictionaries in the stack S_1.

Upon jumping to a deep dictionary d_i, our generalized D must be able to do a constant time jumps in all stacks in $s(d_i)$.

For this to happen we equip each d_i in the stack S_1 with the correct jump pointers into these stacks. The value of these jump pointers is set as the top pointers of stacks $s(t)$ at the time d_i was pushed into D.

A jump into D, yielding a dictionary of size m can now be implemented in $O(m)$ time: The jump pointer into stack S_1 yields a dictionary d stored in it. In d we find m jump pointers into the m stacks $s(d)$ in H, carrying out the these m jumps, concludes the jump operation on D.

The final step in constructing the JSM is by coordinating stacks S_0 and S_1:

- A push operation on stack S_0 is required to push, a (potentially empty) dictionary into the stack S_1.
- A pop operation (the first command at every step of Algorithm 3.6) on stack S_0 forces a pop operation of stack S_1.
- Augment items in stack S_0 to include also the jump pointer to the dictionary it pushed into S_1.
- Let a jump into stack S_0 also force a jump into the stack S_1. (As expected, a jump into an item i means also a jump in stack S_1 to dictionary d_i.)

Thus, the linearly sized JSM data structure keeps its two duties: supporting jumps into the items stack, and maintaining a current map of the jumps to carry out at each item. This map is updated with every push, which might override (or add) entries to it. Most importantly, this map is correctly restored in the process of long jumps into the items stack.

In a way, the JSM is a data structure generalizing the symbol table of a programming languages with lexical binding. The jump operation in the JSM generalizes a "goto" operation to label on the stack.

3.4.3 Push After Jump

Before the algorithm for coordinating stacks S_0 and S_1 is shown, another problem regarding the jump operation needs to be discussed. This problem combines both k' and k^* problems.

Consider the legal string

$$s = \texttt{program id ; const ; begin end}$$

derived from the Pascal BNF fragment (Figure 3.8), representing a program with no parameters, a single definition of a constant, and a body.

During the RLL$_p$'s parse of s, just after parsing the prefix **program id ; const ;** and before reading **begin**, the state of the items stack S_0 is

$$\begin{aligned}
&\langle \textit{Constant} \rangle && ::= \texttt{;} \cdot \\
&\langle \textit{Constants} \rangle && ::= \texttt{const} \cdot \langle \textit{Constant} \rangle \langle \textit{MoreConstants} \rangle \\
&\langle \textit{Definitions} \rangle && ::= \langle \textit{Labels} \rangle \cdot \langle \textit{Constants} \rangle \langle \textit{Nested} \rangle \\
&\langle \textit{Program} \rangle && ::= \texttt{program id} \langle \textit{Parameters} \rangle \texttt{;} \cdot \langle \textit{Definitions} \rangle \langle \textit{Body} \rangle
\end{aligned}$$

when ⟨*Constant*⟩'s item is the top of the stack.

After reading the next input symbol **begin**, the state of the stack ought to be

$$\begin{aligned} \langle \mathit{Body} \rangle &::= \texttt{begin} \cdot \texttt{end} \\ \langle \mathit{Program} \rangle &::= \texttt{program id} \langle \mathit{Parameters} \rangle \texttt{;} \langle \mathit{Definitions} \rangle \cdot \langle \mathit{Body} \rangle \end{aligned}.$$

Breaking down the steps, a non-realtime LL_p would perform three steps before reading the input symbol **begin**:

Popping since the item at the top of the stack was fully parsed, and since **begin** is in $\mathsf{Follow}(\langle \mathit{Constant} \rangle)$, the LL_p would pop all fully parsed rules, without consuming the input symbol.

Advancing upon finishing the parse of ⟨*Definitions*⟩ in the derivation of ⟨*Program*⟩, the LL_p would advance to parse ⟨*Body*⟩.

Consuming since ⟨*Body*⟩'s only rule begins with the terminal **begin**, the input symbol will finally be matched.

The first step encapsulates the k^* factor, while the other two encapsulate the k' factor.

The RLL_p needs to perform all three steps *together*, in a constant amount of work, meaning it needs to jump (completing the first step), push the advanced item of ⟨*Definitions*⟩ (second step) and lastly push ⟨*Body*⟩'s item.

For that cause the RLL_p relies on the JSM, supporting constant time jump operations, and solving the k^* factor problems. The RLL_p also relies on the $\mathsf{Consolidate}(\cdot,\cdot)$ function (introduced later at Section 3.4.5) that solves the problems that rise with k' factor.

But as the current example demonstrates, the two problems combine as the RLL_p is required to *Push after Jump* in realtime.

The problem is solved by elaborating the type of previously mentioned J (the jump pointer type) to hold also information regarding the push that will occur after each jump.

3.4.4 The Jumps Dictionary

How should the JSM be used by the algorithm that generates a specific RLL_p?

Jump operations are all at the responsibility of the JSM, which maintains the information required to support these. But, in order to be able to do this, the generator must provide the RLL_p with the contents dictionary D that should be pushed to S_1, the d_is, together with the push to S_0 (lines 15 and 22 in Algorithm 3.6).

In particular, the RLL_p generator must compute for each item $i \in I$, the dictionary $\mathsf{Jumps}(i)$ which map each token t that triggers a jump with respect to i, to t's jump value (type J). Algorithm 3.7 is the algorithm for doing so.

Let $Y_1\beta = Y_1 \dots Y_n$ be the suffix denoted by i's dot. Then, for each terminal $t \in \Sigma$, function $\mathsf{Jumps}(i)$ determines the shortest prefix $Y_2 \dots Y_{j-1}$ of β which is nullable and such that $t \in \mathsf{First}(Y_j)$. The following line of thought explains the rationale.

Assume that $Y_2 \dots Y_{j-1}$ is nullable and that $t \in \mathsf{First}(Y_j)$, for some $t \in \Sigma$. Then, when the RLL_p finished parsing Y_1 and t is seen, the RLL_p should conclude that $Y_2 \dots Y_{j-1} \overset{*}{\Rightarrow} \epsilon$ and jump forward to the point of the parsing process just before seeing Y_j.

This point in parsing is exactly the item obtained from i by moving the dot to just before Y_j.

One subtlety applies though. In the case that the assumption holds for both j and j', $j < j'$ for the same token t, there are two alternatives to choose from:

Algorithm 3.7 Function $\mathsf{Jumps}(i)$ returning, for an item $i \in I$, the dictionary d mapping each token t that triggers a jump with respect to i, to t's jump value.

Let $[A ::= \alpha \cdot Y_1 \dots Y_n] \leftarrow i$ *// break i into components*
Let $d \leftarrow \emptyset$ *// initialize return variable*
For $j = 2, \dots, n$ *// for all symbols in suffix of i*
  If not $\mathsf{Nullable}(Y_2 \dots Y_{j-1})$ *// continue to build d?*
    break
  For $t \in \mathsf{First}(Y_j)$ *// symbols that might cause jump in* Y_1
    If $d(t) = \bot$ *// 1st update of key t in dictionary d?*
      $i_{addr} = [A ::= \alpha Y_1 \dots \cdot Y_j \dots Y_n]$ *// the jump address*
      // handle the Push after Jump phenomena
      $d(t) = Consolidate(i_{addr}, t)$ *// update d(t)*
If not $\mathsf{Nullable}(Y_2 \dots Y_n)$ *// Is it possible to jump beyond d?*
  For $t \in \Sigma$ *// Don't allow a fallback after this point*
    If $d(t) = \bot$ *// Not a legal jump*
      $d(t) =$ ERROR
Return d

- ERROR is a special jump value that causes the RLL_p to reject instantly.

Item i	Terminal t	$\mathsf{Jumps}(i)[t]$
$[\langle Program \rangle ::= \texttt{program id} \langle Parameters \rangle \texttt{;} \cdot \langle Definitions \rangle \langle Body \rangle]$	`begin`	$[\langle Body \rangle ::= \texttt{begin} \cdot \texttt{end}]$ $[\langle Program \rangle ::= \texttt{program id} \langle Parameters \rangle \texttt{;} \langle Definitions \rangle \cdot \langle Body \rangle]$
$[\langle Constants \rangle ::= \texttt{const} \cdot \langle Constant \rangle \langle MoreConstants \rangle]$	`;`	$[\langle Constant \rangle ::= \texttt{;} \cdot]$ $[\langle MoreConstants \rangle ::= \cdot \langle Constant \rangle \langle MoreConstants \rangle]$ $[\langle Constants \rangle ::= \texttt{const} \langle Constant \rangle \cdot \langle MoreConstants \rangle]$
$[\langle Definitions \rangle ::= \cdot \langle Labels \rangle \langle Constants \rangle \langle Nested \rangle]$	`begin`	$\bot$

Table 3.3: Example values of an entry t from the map d_i returned from $\mathsf{Jumps}(\cdot)$. The grammar in use is our running example defined in Figure 3.8

- $Y_2 \dots Y_{j-1} \overset{*}{\Rightarrow} \epsilon$ and t is the first token in the derivation of Y_j i.e.,

  $$Y_j \overset{*}{\Rightarrow} t\gamma.$$

- $Y_2 \dots Y_{j'-1} \overset{*}{\Rightarrow} \epsilon$ and t is the first token in the derivation $Y_{j'}$ i.e.,

  $$Y_{j'} \overset{*}{\Rightarrow} t\gamma'.$$

Recalling that an LL_p pushes stack symbols in reverse order and that RLL_p emulates its behavior, we can see that $Y_{j'}$ is never given the opportunity to derive t.

After computing Y_j, the algorithm uses $\mathsf{Consolidate}(\cdot, \cdot)$ (defined in Algorithm 3.8) to overcome the push after jump phenomena mentioned in Section 3.4.3.

Table 3.3 presents some of the values computed by Algorithm 3.7 on our running example defined in Figure 3.8

Item i	Terminal t	Consolidate(i,t)
[⟨*Body*⟩ ::= **begin** · **end**]	**end**	[⟨*Body*⟩ ::= **begin end** ·]
[⟨*Constants*⟩ ::= **const** ⟨*Constant*⟩ · ⟨*MoreConstants*⟩]	**;**	[⟨*Constant*⟩ ::= **;** ·] [⟨*MoreConstants*⟩ ::= · ⟨*Constant*⟩ ⟨*MoreConstants*⟩] [⟨*Constants*⟩ ::= **const** ⟨*Constant*⟩ · ⟨*MoreConstants*⟩]
[⟨*Definitions*⟩ ::= · ⟨*Labels*⟩ ⟨*Constants*⟩ ⟨*Nested*⟩]	**const**	[⟨*Constants*⟩ ::= **const** · ⟨*Constant*⟩ ⟨*Constants*⟩] [⟨*Definitions*⟩ ::= ⟨*Labels*⟩ · ⟨*Constants*⟩ ⟨*Nested*⟩]

Table 3.4: Example values for the Consolidate$(\cdot,\cdot)$ functions on the grammar defined in Figure 3.8

The first row of Table 3.3 shows that in case ⟨*Program*⟩'s item will be pushed to S_0, the matching push to S_1 will hold a jump option on terminal **begin**, because **begin** is in First(⟨*Body*⟩), this jump operation will conclude by pushing the two items in Consolidate(i,t) to the stack.

The second row demonstrates a similar idea, where if the parse of ⟨*Constant*⟩ concludes by seeing the terminal **;**, the state of the stack should be changed to the state in which **;** was matched ; Consolidate$(\cdot,\cdot)$ concludes this state.

The last example in Table 3.3 shows that in case a terminal t is not in

$$\mathsf{First}(\langle Labels\rangle\langle Constants\rangle\langle Nested\rangle)$$

(the string following the input item's dot), then the dictionary d_i that Jumps(i) returns, has no mapping for t.

3.4.5 Consolidating Push Operations

Recalling Definition 3, capturing the notion of repeated substitution, we realize that the RLL$_p$-generator must consolidate k' push operations into one.

Function Consolidate(i,t) in Algorithm 3.8 returns the consolidated list of consecutive push operations conducted by the RLL$_p$ parser in state i and encounters terminal t. This is achieved by figuring out, ahead-of-time, the operations of the LL$_p$.

Function Consolidate$(\cdot,\cdot)$ is invoked by the RLL$_p$ generator to pre-compute the consolidated list of push operations for all relevant item-token pairs. At runtime, the RLL$_p$ will push the consolidated lists in constant time, irrespective of the length of the consolidated list.

Some examples of the consolidation table for our running example (defined in Figure 3.8) are presented in Table 3.4.

The first example (in the first line) presents the most "deformed" case of function Consolidate$(\cdot,\cdot)$, in which the input item has a dot before a terminal. In this case the function returns the same item with the dot advanced to after the terminal.

In the second example, the function returns three items. The last item in the result (the third in the table) is the input item itself, the reason is that the rule is currently being parsed when the dot precedes ⟨*MoreConstants*⟩ because it is yet to be fully parsed.

Upon seeing nonterminal ⟨*MoreConstants*⟩ and terminal "**;**" the LL$_p$ would choose rule

$$\langle MoreConstants\rangle ::= \langle Constant\rangle\ \langle MoreConstants\rangle,$$

and thus it is added to the result with a dot at the beginning of the rule's right-hand side(again, because at this time nothing of the rule was parsed). Now a similar decision based on the

Algorithm 3.8 Function Consolidate(i, t) pre-computing L, the list of push operations that happen when an item i at the top of an RLL$_p$'s stack encounters terminal $t \in \Sigma \cup \{\$\}$ on the input.

```
[X ::= α · Yβ] ← i                                  // break i into its components
Let L ← ∅                                           // initialize return value
While Y ∉ Σ                                         // loop while Y is a nonterminal
  push(L, [X ::= α · Yβ])                           // currently parsing Y
  r ← predict(Y, t)                                 // next rule to apply
  [Y ::= X_1 ... X_m] ← r                           // break r into components
  If X_1 ... X_m = ϵ                                // predict(·,·) returned an ϵ-rule
    While exhausted(peek(L))                        // pop all exhausted rules
      pop(L)
    [X ::= α · Yβ] ← pop(L)                         // Y was just fully parsed
    push(L, [X ::= αY · β])                         // advance the rule
  else                                              // predict(·,·) returned a non ϵ-rule
    push(L, [Y ::= ·X_1 ... X_m])                   // we now turn to parse r
  [X ::= α · Yβ] ← pop(L)                           // break i into its components
push(L, [X ::= αY · β])                             // Y must be t
Return L                                            // return the items to push
```

- List L is used in the main loop of the code to emulate the runtime stack of the generated RLL$_p$, and thus, pre-compute the net effect of stack operations that take place in configuration $\langle i, t\rangle$ until t is consumed.

  Accordingly, the emulation applies stack functions pop$(\cdot)$ and peek$(\cdot)$ as well as operation push$(\cdot,\cdot)$ on the list L, as if it were a stack.

  Calling $(\cdot)$ on the emulation stack L at the time an RLL$_p$ is generated, obviates the need for the RLL$_p$ to peek(), at runtime, into its own stack.

- The algorithm relies on function exhausted(i) that returns true for an item i when its dot is in its penultimate position, i.e., the item represents times during parsing in which all right hand side symbols of the item's rule have been parsed except for the last. Since the item was just revealed at the top of the stack, the last symbol was parsed as well, and the rule is fully parsed (and thus, exhausted).

  Intuitively, an item i is exhausted right after the rule's body have been seen in full, and the item "waits" for the RLL$_p$ to take the stack action appropriate when the rule reduces, and items representing it are no longer need.

LL$_p$'s prediction table rule

$\langle Constant\rangle$ ::= **;**

is chosen. Since this rule revealed the terminal **;** that will match the input symbol, the dot in the result item follows the terminal.

The last example (third row of the table) presents another "ability" of the consolidation, which is to ignore nonterminals that derive to ϵ. In the example, since the input token is **const**, $\langle Labels\rangle$ is derived to ϵ. Thus, the last item in the output, is the input item with the dot following nonterminal $\langle Labels\rangle$ instead of preceding it, $\langle Constants\rangle$ rule is then chosen as we seen in the second example.

3.4.6 Putting the Pieces Together

Generating an RLL$_p$ for a given LL grammar requires providing to the built-in algorithm of the RLL$_p$, the specific information it needs to realize the given grammar.

Item i	Terminal t	$\Delta[i,t]$
$[\langle Labels\rangle ::= \epsilon\cdot]$	`procedure`	jump(`procedure`)
$[\langle Nested\rangle ::= \langle Procedure\rangle\ \langle Nested\rangle\cdot]$	`begin`	jump(`begin`)
$[\langle Labels\rangle ::= \epsilon\cdot]$	`end`	**Error**
$[\langle Labels\rangle ::= \texttt{label}\ \langle Label\rangle\ \cdot\ \langle MoreLabels\rangle]$	`;`	push(Consolidate($[\langle Labels\rangle ::= \texttt{label}\ \langle Label\rangle\ \cdot\ \langle MoreLabels\rangle], ;$))
$[\langle Definitions\rangle ::= \cdot\ \langle Labels\rangle\ \langle Constants\rangle\ \langle Nested\rangle]$	`begin`	jump(`begin`)

Table 3.5: Example values for the prediction table Δ on the grammar defined in Figure 3.8

Reexamining the code of the RLL$_p$ (Algorithm 3.6) we see that all such information is contained in function Δ. Therefore, the core of the RLL$_p$ generator is Algorithm 3.9 that computes Δ.

Algorithm 3.9 Compute contents of prediction table (transition function) entry $\Delta[i,t]$ for all item $i \in I$, token $t \in \Sigma$ pairs for which this entry is defined.

```
For i ∈ I                                                    // for each item
  Let [A ::= α · Yβ] ← i                                     // break i into components
  If Y ∈ Σ                                                   // Δ doesn't handle terminals
    continue                                                 // handle next item.
  For t ∈ Σ                                                  // calculate entry Δ[i,t]
    If t ∈ First(Yβ)                                         // t is consumed while parsing i
      Let L ← Consolidate(i,t)
      Let Δ[i,t] ← [push(L)]                                 // a push operation
    else if t ∈ Follow(A) and Nullable(Yβ)                   // t is consumed after parsing i
      // Obtain jumps dictionary and store in Δ:
      Let Δ[i,t] ← [jump(t)]                                 // a jump operation
```

Table 3.5 provides some of Δ's values for our running example defined in Figure 3.8.

In the first two rows of Table 3.5, there are examples for rules that have been fully parsed. In these cases, the input symbols (**`procedure`** and **`begin`** correspondingly) are in the rules left-hand side's Follow$(\cdot)$ set ($\langle Labels\rangle$ and $\langle Nested\rangle$ correspondingly), causing Δ returns a jump operation.

The third row presents a case in which the rule $\langle Labels\rangle ::= \epsilon$ was fully parsed, but the input symbol is not in the set Follow$(\langle Labels\rangle)$, and thus, Δ returns no operation (interpreted as error).

The fourth row of Table 3.5 presents a case in which the input terminal is in the First$(\cdot)$ set of the symbol following the dot, thus, a push operation is returned, with the values of the corresponding Consolidate$(\cdot,\cdot)$ function.

The last row of the table presents a special case, in which the input item's dot is not at the end of $\langle Definition\rangle$'s rule, and the operation is a jump(**`begin`**) operation. This happens because **`begin`** is not in First$(\langle Definition\rangle)$ and is in Follow$(\langle Definition\rangle)$ (it is also required the $\langle Definition\rangle$ will be nullable).

Recall that we manage the "k*" phenomena using jumps, and that the jumps are realized by the sophisticated JSM data structure that drives the RLL$_p$.

The contract between the RLL$_p$ and its generator is that function Δ supplies the dictionaries that need to be pushed (and later jumped to) by the JSM.

This information is retrieved by the RLL$_p$ from the Δ table, and used and push$(\cdot)$ed

accordingly into the JSM. Only with the aide of this information, the underlying JSM can support the constant-time jumps (lines 5 and 25).

For this reason, Algorithm 3.9 implicitly consults the function Jumps$(\cdot)$ (recall that Jumps$(\cdot)$ is the coordinator between stacks S_0 and S_1) to compute this dictionary storing it in the Δ table.

Another part of the contract between the parser and its generator deals with the "k'" phenomena. The generator invokes function Consolidate$(\cdot)$ (Algorithm 3.8) to compute L, the consolidated list of push operations. List L will be later read by RLL_p (line 18 in Algorithm 3.6).

Chapter 4

A Prototype Implementation

In previous chapters, we've seen the RLL_p, a jump automata based recognizer that emulates the classic LL_p. The RLL_p uses the JSM, a special data structure supporting jump and push operations in constant amount of work, it also uses some pre-computed tables ($\mathsf{Consolidate}(\cdot,\cdot)$, $\mathsf{Jumps}(\cdot)$, $\Delta(\cdot,\cdot)$...).

In this chapter we will show can such data structure be encoded using the JAVA type checker, and the generics mechanism in particular.

4.1 Encoding of the Stack Items

As we seen in Section 2.4, the stack symbols are encoded to types. The main stack of the RLL_p (S_0) uses items as stack symbols, thus, each grammar item is encoded into a JAVA type.

For example, of the grammar rule

$$\langle \mathit{Definitions} \rangle ::= \langle \mathit{Labels} \rangle \langle \mathit{Constants} \rangle \langle \mathit{Nested} \rangle$$

taken from our running example in Figure 3.8, four derived RLL_p items will be encoded

$$\begin{aligned}
&[\langle \mathit{Definitions} \rangle ::= \cdot \langle \mathit{Labels} \rangle \langle \mathit{Constants} \rangle \langle \mathit{Nested} \rangle] \\
&[\langle \mathit{Definitions} \rangle ::= \langle \mathit{Labels} \rangle \cdot \langle \mathit{Constants} \rangle \langle \mathit{Nested} \rangle] \\
&[\langle \mathit{Definitions} \rangle ::= \langle \mathit{Labels} \rangle \langle \mathit{Constants} \rangle \cdot \langle \mathit{Nested} \rangle] \\
&[\langle \mathit{Definitions} \rangle ::= \langle \mathit{Labels} \rangle \langle \mathit{Constants} \rangle \langle \mathit{Nested} \rangle \cdot]
\end{aligned} .$$

The notation we use from now on, is that the name of the class is the nonterminal at the left-hand side of the item concatenated with the dot index. The types in our example are depicted in Figure 4.1.

4.2 Encoding the JSM

The RLL_p uses the JSM to support jump operations. During it's run, the RLL_p uses the JSM's information if $\Delta(\cdot,\cdot)$ determines a jump operation. Observe that this might happen only when the input symbol is in $\mathsf{Follow}(\langle A \rangle)$ set, $\langle A \rangle$ being the left-hand side of the top of the stack item.

Figure 4.1 Type encoding of items derived from rule

$$\langle Definitions \rangle ::= \langle Labels \rangle \langle Constants \rangle \langle Nested \rangle$$

(taken from the PASCAL fragment BNF defined in Figure 3.8). The index of the dot is denoted by the number in the class name.

```
class Definition0<…> { … }
class Definition1<…> { … }
class Definition2<…> { … }
class Definition3<…> { … }
```

For example, when recognizing the PASCAL fragment grammar, if the top of the stack is e.g.,

$$i = [\langle Definitions \rangle ::= \langle Labels \rangle \langle Constants \rangle \langle Nested \rangle \cdot]$$

then for each input symbol from $\mathsf{Follow}(\langle Definitions \rangle)$ (that contains **begin** and **procedure**), the RLL_p will perform a jump operation (the only exception is when \$ is in the follow set, which in this case the RLL_p accepts).

Since the JSM's state can only be known at "runtime" of the RLL_p (being compile time of the fluent API), the JSM must be encoded with JAVA generics. Using out observation regarding the $\mathsf{Follow}(\cdot)$ set, each item

$$[\langle A \rangle ::= \alpha \cdot \beta]$$

will have k type parameters, k being the size of the corresponding $\mathsf{Follow}(\langle A \rangle)$ set - for item i, $k = 1$.

The JSM's hash table is encoded with the type parameters of each type, when "querying" this hash table is done by using the name of this parameter. The second operation of the JSM is push operation. It will be discussed in the next sections.

4.3 Encoding Δ

Recall that Δ is the main prediction table of the RLL_p. With the main loop step in Algorithm 3.6, it determines whether the automaton "accept"s, "reject"s, "jump"s or "push"es.

"Reject" as we seen in the jDPDA encoding (Chapter 2), encoding of "reject" operation in case the next input symbol is t summarizes in *not* adding any method for t, this way, trying to invoke t's method will cause a type-check error.

"Accept" operation can only occur if the end-of-input symbol, \$, was seen. Encoding "accept" operation then, is by adding method \$() to an item that can be followed by \$ (there is more to this issue than meets the eye, and we will revise it later in Section 5.2.2).

"Jump" encoding a "jump" operation upon seeing input symbol t means popping multiple times from the stack and reverting to a former state.

This operation is enabled by the JSM. Querying it is done by using the correct type parameter. Performing a $\mathsf{jump}(t)$ operation is simply returning t's type parameter. The actual parameter will be the state of the JSM after a jump operation.

Figure 4.2 shows an example of jump operations with our PASCAL fragment grammar example.

Figure 4.2 Type encoding example of jump operations where

$$i = [\langle Constant \rangle ::= \texttt{;} \cdot]$$

is the item at top of the stack (taken from the PASCAL fragment BNF defined in Figure 3.8). Methods `semi_t()`, `begin_t()` and `procedure_t()` represent the terminals "`;`", "`begin`" and "`procedure`" respectively.

```
class Constant1<jump_semi, jump_procedure, jump_begin>{
  public jump_semi semi_t() { … }
  public jump_begin begin_t() { … }
  public jump_procedure procedure_t() { … }
}
```

Item i in Figure 4.2 is ready to reduce, thus, $\Delta(i,t) = \mathrm{jump}(t)$ for every terminal t from

$$\mathsf{Follow}(\langle Constant \rangle) = \{\texttt{;}, \texttt{procedure}, \texttt{begin}\}.$$

In all of these cases, the implementation of the "jump" operation with JAVA generics is expressed by the return type of the methods, specifically, the corresponding type argument will be returned.

"Push" lastly, the "push" operation is the most complicated one as it needs to express the full complexity of the JSM : S_0, S_1 and the coordination between them. The coordination between S_0 and S_1 is expressed in JAVA with the actual parameters of items.

Figure 4.3 shows how the coordination between S_0 and S_1 is encoded using JAVA generics.

Figure 4.3 Type encoding example of $\mathsf{push}(\mathsf{Consolidate}(i,t))$ where

$$i = [\langle Procedure \rangle ::= \texttt{procedure id} \cdot \langle Parameters \rangle \texttt{ ;} \langle Definitions \rangle \langle Body \rangle]$$

and $t =$ `()`. Method `pair_t()` represents the terminal "`()`". Type `Procedure4` represents the item i with dot after `;` and type `Parameters1` represents item $[\langle Parameter \rangle ::= \texttt{()}\cdot]$.

```
class Procedure2<jump_procedure, jump_begin> {
  public Parameters1<Procedure4<jump_procedure, jump_begin>> pair_t() { … }
  ⋮
}
```

Figure 4.3 presents the encoding of item

$$i = [\langle Procedure \rangle ::= \texttt{procedure id} \cdot \langle Parameters \rangle \texttt{ ;} \langle Definitions \rangle \langle Body \rangle].$$

Upon seeing terminal $t =$ `()`, the RLL$_p$ decides to push the value

$$\mathsf{Consolidate}(i, t) = \{\ \begin{matrix}[\langle \textit{Parameters} \rangle ::= \texttt{()} \cdot\ ] \\ [\langle \textit{Procedure} \rangle ::= \texttt{procedure id} \cdot\ \langle \textit{Parameters} \rangle\ \texttt{;}\ \langle \textit{Definitions} \rangle\ \langle \textit{Body} \rangle\ ]\end{matrix}\ \},$$

when $\langle \textit{Parameter} \rangle$'s item is the new item at the top of the stack. Sure enough, the return type of method `pair_t()` in Figure 4.3 is `Parameter1`. Note that the actual parameters (i.e., the instantiation of generic types) encode the JSM's "runtime" state. Consider now what should be pushed to S_1 (for brevity, we only show `()`'s entry in the partial maps). Since

$$\mathsf{Jumps}([\langle \textit{Parameters} \rangle ::= \texttt{()} \cdot\ ]) = \{\},$$

it does not effect on S_1, and accordingly, on the actual parameters, while

$$\mathsf{Jumps}([\langle \textit{Procedure} \rangle ::= \texttt{procedure id} \cdot\ \langle \textit{Parameters} \rangle\ \texttt{;}\ \langle \textit{Definitions} \rangle\ \langle \textit{Body} \rangle\ ])[\texttt{()}] =$$

$$[\langle \textit{Procedure} \rangle ::= \texttt{procedure id} \langle \textit{Parameters} \rangle\ \texttt{;}\ \cdot\ \ \langle \textit{Definitions} \rangle\ \langle \textit{Body} \rangle\ ]$$

means that if the RLL$_p$ will perform $\mathsf{jump}(\texttt{;})$, the operation will return the type $\textit{Procedure}4$ (defined in Figure 4.3).

Returning to our example (Figure 4.3), `Parameters1`'s actual parameter is `Procedure4` as it should be the type return upon jumping on "`;`", and that since we have no additional information regarding the state of the JSM, `Procedure4`'s actual parameters are given from `Procedure2` formal parameters.

4.4 Small Implementation Details

It is worth mentioning that for technical convenience, we did a small transformation on the input grammar, as being done by LR parsers. A new start symbol $\langle S' \rangle$ is added to the grammar while for each old start symbol $\langle S \rangle$, a rule $\langle S' \rangle ::= \langle S \rangle$ is introduced.

The reason is to prevent cases where the start symbol appears on the right-hand side of derivations, without constraining the use of Fajita. In these cases, the $\mathsf{Follow}(\cdot)$ set of the start symbol might not contain \$ exclusively, which means that the derived items will have type parameters. This situation is awkward as the correct instantiation of these items will always have `Error` types as arguments.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The main contribution of this thesis is the first *practical* algorithm for the automatic generation of a fluent API for Java from LL grammars. Fajita, a fluent API software system by itself, is a prototypical implementation of this algorithm.

More work is required to mature Fajita into a production tool. A stumbling block is that experimenting with Fajita we discovered limitations of the Java compiler (e.g., `javac 1.8.0_66_`), sometimes crashing when trying to compile Fajita generated code and often applying unnecessary replication in the representation of types, leading to memory bloat problems.

These limitations are likely to receive more attention by compiler developer as the use of fluent APIs increases.

We also contributed on a *theoretical* front: we provide a proof that most useful grammars have a fluent API. This brings good news to library designers laboring at making their API slick, accessible, and more robust to mistakes of clients: If your API can be phrased in terms of a "decent" BNF, do not lose hope; the task may be Herculean, but it is (most likely) possible.

Other practitioners may appreciate the toolbox of type encodings offered here gaining better understanding of the computational expressiveness of Java generics and type hierarchy, and, a better tool for designing, experimenting with and perfecting fluent APIs.

On a *philosophical* perspective, several modern programming languages acquire high-level constructs at a staggering rate (C++ and Scala [44] being prominent examples). The main yardstick for evaluation these is "programmer's convenience". This work suggests an orthogonal perspective, namely computational expressiveness, or, stated differently, ranking of a new construct by its ability to recognize languages in the Chomsky hierarchy [11].

5.2 Future Work

5.2.1 Parsing

A problem related to that of recognizing a formal language, is that of parsing, i.e., creating, for input which is within the language, a parse tree according to the language's grammar. In the domain of fluent APIs, the distinction between recognition and parsing is in fact the distinction between compile time and runtime. Before a program is run, the compiler checks whether

the fluent API call is legal, and code completion tools will only suggest legal extensions of a current call chain.

In contrast, a parse tree can only be created at runtime, as the fluent API can either or compile, or not compile, but cannot do much more.

There are two possible ways to parse at runtime with fluent APIs. The first is by creating the parse-tree iteratively, where each method invocations in the call chain adding more components to this tree (again, it's computed during runtime).

The second way is by generating this tree in "batch" mode: this is done by maintaining a *fluent-call-list* which starts empty and grows at runtime by having each method invoked add to it a record storing the method's name and values of its parameters. The list is completed at the end of the fluent-call-list, at which point it is fed to an appropriate parser that converts it into a parse tree (or even an AST).

An interesting hypothesis we have, is that the parse tree can be created at runtime without running any additional parsing algorithm. We believe it can be done because in LL parsing once we see a new rule, we know it is the next derivation in the leftmost derivation of the string.

The main question we wish to leave the reader with is:

> Can we ALWAYS parse without running ANY parsing method on runtime?

5.2.2 The Endable Symbols Pitfall

Consider the nonterminal $\langle Body \rangle$ from out running example (Figure 3.8), since $\langle Body \rangle$ appears as the last symbol in a rule of a start symbol, \$ is in $\mathsf{Follow}(\langle Body \rangle)$.

In that case, the type encoding item

$$i = [\langle Body \rangle ::= \texttt{begin end} \cdot]$$

should have a method `$()` concluding the API chain.

However, $\langle Body \rangle$ appears in a non-endable context in $\langle Procedure \rangle$'s rule (this context is not endable as \$ is not in $\mathsf{Follow}(\langle Procedure \rangle)$) and since we have only one type encoding for item i, a user of the fluent API could legally conclude the API chain even when the call chain is not legal.

We believe this problem can be solved with some grammar tweaking.

5.2.3 The Recursive JSM Encoding Pitfall

In the way the JSMs are encoded in Java, the S_1 stack of partial mapping is encoded as if the JSM have for each terminal a JSM snapshot that represents the state of the JSM in case of jumping.

In the case of right recursion, the JSM is requested to keep a snapshot of itself (which includes another snapshot of itself, etc.). It can be thought of as the popping of the jump and the "push after jump" cancel each other perfectly, to remain in the same state. Thus, the matching encoding of the JSM is a type, that tries to have itself as one of its type arguments.

We think this problem can be solved with two possible solutions:

1. Upon recognizing this state, remove the recursive formal parameter, and the method that performs the jump will simply return the type of "`this`"

2. Java allows recursive types definitions as in the following example

```
interface Test<T extends Enum<T>>{ }
```

5.2.4 Generating a Realtime Parser from an LR Grammar

LR grammar are much more expressive than LL grammar, as they can express the syntax of most modern programming languages.

The LR parser is much more complex as every state of the parser holds all the possibilities for parsing from this point (this is due to the parser's rightmost derivation property).

We know from Chapter 2 that it can theoretically be done, but we would like a practical algorithm.

5.2.5 Elaboration for different languages

Having opened a debate over the expressiveness of the Java type checker, and Java generics. This research can be extended to other languages, such as C# or Scala.

Bibliography

[1] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. C++ in Depth Series. Addison-Wesley, 2004.

[2] J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-oriented programming. In G. Leavens, editor, *Proc. of the 24th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'09)*, pages 1015–1022, Orlando, FL, USA, Oct. 2009. ACM Press.

[3] K. Arnold and J. Gosling. *The* Java *Programming Language*. The Java Series. Addison-Wesley, Reading, MA, 1996.

[4] M. H. Austern. *Generic programming and the STL: using and extending the C++ Standard Template Library*. Addison-Wesley, 1998.

[5] J.-M. Autebert, J. Berstel, and L. Boasson. *Context-Free Languages and Pushdown Automata*. Springer, 1997.

[6] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming. In *Advanced Functional Programming*, pages 28–115. Springer, 1999.

[7] N. E. Beckman, D. Kim, and J. E. Aldrich. An empirical study of object protocols in the wild. In M. Mezini, editor, *Proc. of the 25th Euro. Conf. on OO Prog. (ECOOP'11)*, volume 6813 of *LNCS*, pages 2–26, Lancaster, UK, June25-29 2011. Springer.

[8] K. Bierhoff and J. E. Aldrich. Lightweight object specification with typestates. In M. Wermelinger and H. C. Gall, editors, *Proc. of the 10th European Soft. Eng. Conf. and 13th ACM SIGSOFT Symp. on the Foundations of Soft. Eng. (ESEC/FSE'05)*, pages 217–226, Lisbon, Portugal, Sept. 2005. ACM Press.

[9] E. Bodden. 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*, pages 1–6, 2014.

[10] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In B. N. B. Freeman and C. Chambers, editors, *Proc. of the 13th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'98)*, pages 183–200, Vancouver, BC, Canada, Oct.18-22 1998. ACM SIGPLAN Notices 33(10).

[11] N. Chomsky. *Formal properties of grammars*. Addison-Wesley, 1963.

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

[13] B. Courcelle. On jump-deterministic pushdown automata. *Mathematical Systems Theory*, 11:87–109, 1977.

[14] J. C. Dehnert and A. Stepanov. Fundamentals of generic programming. In *Generic Programming*, pages 1–11. Springer, 2000.

[15] A. V. Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[16] C. Donnelly and R. Stallman. *Bison*, 2015.

[17] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[18] S. Erdweg, L. C. Kats, T. Rendel, C. Kästner, K. Ostermann, and E. Visser. Sugarj: Library-based language extensibility. In K. Fisher, editor, *Proc. of the 26th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'10)*, pages 187–188, Portland OR, USA, Oct.22-27 2011. ACM Press.

[19] M. Fowler. *Domain-specific languages*. Pearson Education, 2010.

[20] S. Freeman and N. Pryce. Evolving an embedded domain-specific language in Java. In P. L. Tarr and W. R. Cook, editors, *Proc. of the OOPSLA'06 Companion*. ACM Press, Oct.22-26 2006.

[21] R. Garcia, J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock. A comparative study of language support for generic programming. In R. Crocker and G. L. S. Jr., editors, *Proc. of the 18th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'03)*, pages 115–134, Anaheim, CA, USA, Oct. 2003. ACM SIGPLAN Notices 38 (11).

[22] J. Gil and Z. Gutterman. Compile time symbolic derivation with C++ templates. In *Proc. of the USENIX C++ Conf.*, pages 249–264, Santa Fe, NM, Apr. 1998. USENIX Association.

[23] J. Y. Gil and K. Lenz. Simple and safe SQL queries with templates. In C. Consel, editor, *Proc. of the 6th Conf. on Generative Prog. & Component Eng.*, LNCS, pages 13–24, Salzburg, Austria, Oct. 2007. ACM Press.

[24] J. Y. Gil and T. Levy. An algorithm for generating fluent apis for java. http://stlevy.cswp.cs.technion.ac.il/wp-content/uploads/sites/50/2016/04/00.pdf. Have been submitted to OOPSLA 2016.

[25] J. Y. Gil and T. Levy. Formal language recognition with the java type checker. http://stlevy.cswp.cs.technion.ac.il/wp-content/uploads/sites/50/2015/12/00.pdf. To appear in the proceedings of ECOOP 2016.

[26] A. Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, MA, 1984.

[27] P. Graham. *ANSI Common LISP*. Prentice Hall, 1995.

[28] Z. Gutterman. Turing templates—on compile time power. Master's thesis, Technion—Israel Institute of Technology, 2003.

[29] R. Harter. A game theoretic approach to the toilette seat problem. *The Science Creative Quarterly*, 1(1), May 2005.

[30] A. Hejlsberg, S. Wiltamuth, and P. Golde. *The C# Programming Language*. Addison-Wesley, Reading, MA, 2^{nd} edition, Oct. 2003.

[31] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 2^{nd} edition, 2001.

[32] P. Hudak. Domain-specific languages. *Handbook of Programming Languages*, 3:39–60, 1997.

[33] C. Ibsen and J. Anstey. *Camel in action*. Manning Publications Co., Shelter Island, NY, 2010.

[34] JBoss Group. Hibernate product homepage. http://www.hibernate.org/, 2006.

[35] J. Kabanov and R. Raudjärv. Embedded typesafe domain specific languages for JAVA. *Proceedings of the 6^{th} international symposium on Principles and practice of programming in* JAVA- *PPPJ '08*, 2008.

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

[37] D. E. Knuth. *TEX and METAFONT: New Directions in Typesetting*. American Mathematical Society, Boston, MA, USA, 1979.

[38] R. Larsen. Fluenty: A type safe query API. Master's thesis, University of Oslo, 2012.

[39] P. M. Lewis, R. E. Stearns, et al. Syntax directed transduction. In *Proc. of the 7^{th} Annual Symposium on Switching and Automata Theory*, pages 21–35. IEEE, 1966. Original introduction of LL parsing.

[40] M. Linna and M. Penttonen. New proofs for jump dpda's. In *Mathematical Foundations of Computer Science*, pages 354–362. Springer, 1979.

[41] P. Linz. *An Introduction to Formal Languages and Automata*. Jones & Bartlett Learning, 2011.

[42] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling objects, relations and XML in the .NET framework. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (ICMD'2006)*, Chicago, Illinois, 2006.

[43] D. R. Musser and A. A. Stepanov. Generic programming. In *Symbolic and Algebraic Computation*, pages 13–25. Springer, 1989.

[44] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[45] M. M. Papi. Practical pluggable types for Java. Master's thesis, Massachusetts Institute of Technology, 2008.

[46] J. Schoenbrunner. A LiFo dynamic dictionary. From ArXiv Mathematics e-prints, Mar. 1995.

[47] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 3^{rd} edition, 1997.

[48] A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages. *Centrum voor Wiskunde en Informatika*, 5:12, 2000.

[49] D. Vandevoorde and N. M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, 2002.

[50] T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

[51] N. Wirth. The programming language Pascal. *Acta Informatica*, 1:35–63, 1971.

[52] D. H. Younger. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967.