An Algorithm For Generating Fluent APIs for Java

Mr. U. N. Owen & Co.

Abstract

A recent result shows that the JAVA type system and its type-checker are mighty enough to emulate deterministic pushdown automata (DPDAs), and hence, capable (disregarding cost) to recognize deterministic context free languages (DCFGs). The problem is of concrete practical value, since, as it turns out, this recognition is essential for automatic generation of fluent APIs from their specification.

This work advances the state of knowledge, in presenting, 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 (and parses) 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.

We implement the algorithm in FAJITA, a prototypical tool to convert a (fluent API) specification of a certain fluent API design, into an actual implementation.

1. Introduction

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?

One concrete reason is fluent APIs [11, 20, 29]. There is joy, and in many cases boost to productivity in elegant, 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 [21], open-source integration framework), and,

```
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[1], a framework for writing SQL like code in JAVA [3], much like LINQ project [25] in the context of C$^{\#}$ [19]).

We argue that the actual implementation of the many fluent API systems jMock [12], Hamcrest[2], EasyMock[3], jOOR[4], jRTF[5], etc., 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] [17]), the type system is not Turing complete.

It was shown recently that for any reasonable fluent API language definition (specifically, one which can be recognized by an LR(k) grammar for some $k \geq 0$), there exists some JAVA implementation that realizes this language [15].

The theoretical result 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 (Alg. 4.5)

[1] http://www.jooq.org

[2] http://hamcrest.org/JavaHamcrest/

[3] http://easymock.org/

[4] https://github.com/jOOQ/jOOR

[5] https://github.com/ullenboom/jrtf

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 Sect. 5.1) for the emulation, and more efficient "compilation" of the RLL$_p$ into the JAVA type system.

We also present FAJITA[6] a JAVA tool that implements this algorithm.

Fig. 1.1 is a BNF specification of a fluent API for a certain fragment of SQL.

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

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

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

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

```
new BNFBuilder(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 non-terminals. These 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;
}
```

[6] ***F**luent **API** for **J**AVA (**I**nspired by the **T**heory of **A**utomata)*

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,

Fig. 1.3 Legal sequences of calls in the sql fragment example

```
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.

Fig. 1.4 Illegal sequences of calls in the sql fragment example

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

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. Some fluent API definitions create the parse-tree iteratively, where each method invocations in the call chain adding more components to this tree. However, it is always possible to generate 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).

1.1 Related Work

The challenges of JAVA generic programming were highlighted by Garcia et al. [13] research on the expressive power of generics in half a dozen major programming languages, Indeed, unlike C++ [**?**,1,5,10,14,26], the literature on meta-programming with JAVA generics is minimal.

Suggestions for semi-automatic generation of fluent APIs in JAVA can be found e.g., in the work of Bodden [8] and on numerous locations in the web. However, none of these matured into an actual general purpose implementation.

Fluflu[7] is a software artifact similar in principle to FAJITA, which uses JAVA annotations to define deterministic finite automaton (DFA), to be compiled into a fluent API based on this automaton.

Compare FAJITA use case (Figure 1.2) with Figure 1.5 in which fluflu is used to define a DFA with a single, both initial and accepting state, with a single self transition, labeled with terminal a, whereby realizing the regular language $L = a^*$.

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

Fig. 1.5 A fluflu specification of the DFA recognizing the regular language a^*

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

Outline. Sect. 2 familiarizes the reader with techniques of converting a formal language specification into JAVA types that realizes the fluent API defined by the language. The example in this section is used in Sect. 3 to demonstrate FAJITA. The reminder of LL parsing theory in Sect. 4 is to explain the challenges to be met by main algorithm for the compilation of an LL language into into the JAVA type-checking model. The main algorithm is described in Sect. 5. Sect. 6 concludes.

2. 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 [2,7]) 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 [6] 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.

2.1 A Type State Example

An object of type `Seat`[8] 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 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()`

[8] example inspired by earlier work of Richard Harter on the topic [18].

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 2.1 as being type correct.

Fig. 2.1 Legal sequences of calls in the toilette seat example

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

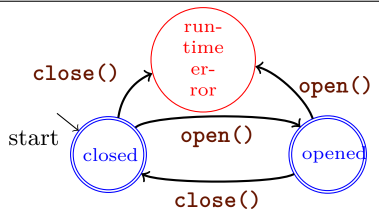
At the same time, illegal sequences as made in Figure 2.2 should be signaled as type errors.

Fig. 2.2 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 2.3.

Fig. 2.3 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`.[9]
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.[10]
3. The initial state q_0 is made special in being the only type from which a fluent API call chain can start.[11]

The result of employing these rules to our toilette example is depicted at Figure 2.4

[9] 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.

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

[11] 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.

Fig. 2.4 A fluent API realizing the toilette seat object defined as a DFA in Figure 2.3

```
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 2.4 we achieved our goals regarding the legal usage examples in Figure 2.1 and the illegal usage examples in Figure 2.2. 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. Introducing Fajita

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

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

Figure 3.1 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 3.1.

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*

Fig. 3.1 A BNF grammar for the toilette seat problem

$$
\begin{array}{rcl}
\langle \textit{Visitors} \rangle & ::= & \langle \textit{Down-Visitors} \rangle \\
\langle \textit{Down-Visitors} \rangle & ::= & \langle \textit{Down-Visitor} \rangle \ \langle \textit{Down-Visitors} \rangle \\
 & | & \langle \textit{Raising-Visitor} \rangle \ \langle \textit{Up-Visitors} \rangle \\
 & | & \epsilon \\
\langle \textit{Up-Visitors} \rangle & ::= & \langle \textit{Up-Visitor} \rangle \ \langle \textit{Up-Visitors} \rangle \\
 & | & \langle \textit{Lowering-Visitor} \rangle \ \langle \textit{Down-Visitors} \rangle \\
 & | & \epsilon \\
\langle \textit{Up-Visitor} \rangle & ::= & \texttt{male()} \ \texttt{urinate()} \\
\langle \textit{Down-Visitor} \rangle & ::= & \texttt{female()} \ \langle \textit{Action} \rangle \\
 & | & \texttt{male()} \ \texttt{defecate()} \\
\langle \textit{Raising-Visitor} \rangle & ::= & \texttt{male()} \ \texttt{raise()} \ \texttt{urinate()} \\
\langle \textit{Lowering-Visitor} \rangle & ::= & \texttt{female()} \ \texttt{lower()} \ \langle \textit{Action} \rangle \\
 & | & \texttt{male()} \ \texttt{lower()} \ \texttt{defecate()} \\
\langle \textit{Activity} \rangle & ::= & \texttt{urinate()} \\
 & | & \texttt{defecate()}
\end{array}
$$

Fig. 3.2 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();
```

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

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 3.2.

The call to function `go()` (last line in Figure 3.2) 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 3.3 (which incidentally can be written in FAJITA fluent API iteself)

Fig. 3.3 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()
```

Algorithm 4.1 The iterative step of an LL_p

```
Let $X \leftarrow$ pop()                        // what's next to parse?
If $X \in \Sigma$                               // anticipate to match terminal $X$
    If $X \neq$ next()                          // read terminal is not anticipated $X$
        REJECT                                  // automaton halts in error
    else                                        // terminal just read was the anticipated $X$
        CONTINUE                                // restart, popping a new $X$, etc.
else if $X = \$$                                // anticipating end-of-input
    If $\$ \neq$ next()                         // not the anticipated end-of-input
        REJECT                                  // automaton halts in error
    else                                        // matched the anticipated end-of-input
        ACCEPT                                  // all input successfully consumed
else                                            // anticipated $X$ must be a nonterminal
    Let $R \leftarrow \delta(X,$ peek()$)$      // determine rule to parse $X$
    If $R = \bot$                               // no rule found
        REJECT                                  // automaton halts in error
    else                                        // A rule was found
        Let $(Z ::= Y_1, \ldots, Y_k) \leftarrow$ R   // break into left/right
        assert $Z = X$                          // $\delta$ constructed to return valid $R$ only
        print($R$)                              // rule $R$ has just been applied
        For $i = k, \ldots, 1$, push($Y_i$)     // push in reverse order
        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 pop($\cdot$) and push($\cdot$) operate on the pushdown stack; function next() returns and consumes the next terminal from the input string; function 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.

4. 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 Sect. 5. This section gives an intuitive perspective on this algorithm.

We will first recall the essentials of the classical LL(1) parsing algorithm (Sect. 4.1). Then, we will explain the limitations of the computational model offered by JAVA generics (Sect. 4.2).

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

Building on all these, Sect. 4.3 can make the intuitive introduction to the main algorithm. To this end, we revise here the classical algorithm [23] 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.

4.1 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 Alg. 4.1.

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.

4.2 LL(1) Parsing with Java Generics?

Can Alg. 4.1 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 [15]. 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 1 (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 2 (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 4.1. This grammar, inspired by the prototypical specification of Pascal [30][12], shall serve as running example.

The grammar preserves the "theme" of nested definitions of Pascal, while trimming these down as much as possible. Table 1 presents some legal sequences derived by this grammar.

`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 1: Legal words in the language defined by Figure 4.1

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 Alg. 4.2), 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 Alg. 4.3).

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

Fig. 4.1 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)

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

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 Alg. 4.4).

Algorithm 4.2 An algorithm for computing $\mathsf{First}(X)$ for each grammar symbol X in the input grammar $G = \langle \Sigma, \Xi, P \rangle$

```
For $A \in \Xi$                               // initialize $\mathsf{First}(\cdot)$ for all nonterminals
  $\mathsf{First}(A) = \emptyset$
For $t \in \Sigma$                            // initialize $\mathsf{First}(\cdot)$ for all terminals
  $\mathsf{First}(t) = \{t\}$
While No changes in any $\mathsf{First}(\cdot)$ set
  For $X ::= Y_0 \ldots Y_k \in P$            // for every production
    $\mathsf{First}(X) \cup= \mathsf{First}(Y_0)$     // add $\mathsf{First}(Y_0)$ to $\mathsf{First}(X)$
    For $i \in \{0 \ldots k\}$                // for each symbol in the RHS
      If $\mathsf{Nullable}(Y_0 \ldots Y_{i-1})$    // if the prefix is nullable
        $\mathsf{First}(X) \cup= \mathsf{First}(Y_i)$ // add $\mathsf{First}(Y_i)$ to $\mathsf{First}(X)$
```

Algorithm 4.3 An algorithm for computing $\mathsf{First}(\alpha)$ for some string of symbols α. This algorithm relies on results from Alg. 4.2

```
Let $Y_0 \ldots Y_k \leftarrow \alpha$               // break $\alpha$ into its symbols
$\mathsf{First}(\alpha) \cup= \mathsf{First}(Y_0)$    // initialize $\mathsf{First}(\alpha)$ with $\mathsf{First}(Y_0)$
For $i \in \{1 \ldots k\}$                  // for each symbol in the string
  If $\mathsf{Nullable}(Y_0 \ldots Y_{i-1})$          // if the prefix is nullable
    $\mathsf{First}(\alpha) \cup= \mathsf{First}(Y_i)$    // add $\mathsf{First}(Y_i)$ to $\mathsf{First}(\alpha)$
```

For example, the nonterminal $\langle \textit{Definitions} \rangle$ 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 $\langle \textit{Definitions} \rangle$ is always followed by $\langle \textit{Body} \rangle$, and $\mathsf{First}(\langle \textit{Body} \rangle)$ contains only `begin`.

The $\mathsf{First}(\cdot)$ and $\mathsf{Follow}(\cdot)$ sets for our running example (Figure 4.1) are listed in Table 2.

Algorithm 4.4 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

```
$\mathsf{Follow}(S) = \{\$\}$                          // initialize the start symbol
While No changes in any $\mathsf{Follow}(\cdot)$ set
  For $A ::= Y_0 \dots Y_k \in P$                     // for each grammar rule
    For $i \in \{0 \dots k\}$                        // for each symbol in the RHS
      If $Y_i \notin \Xi$                            // compute only for nonterminals
        continue
      $\mathsf{Follow}(Y_i) \cup= \mathsf{First}(Y_{i+1} \dots Y_k)$
      If $\mathsf{Nullable}(Y_{i+1} \dots Y_k)$          // if the suffix is nullable
        $\mathsf{Follow}(Y_i) \cup= \mathsf{First}(A)$            // add First(A)
```

Nonterminal	$\mathsf{First}(\cdot)$	$\mathsf{Follow}(\cdot)$
⟨*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 2: Sets $\mathsf{First}(\cdot)$ and $\mathsf{Follow}(\cdot)$ for nonterminals of the grammar in Figure 4.1

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 in Table 2. The $\mathsf{Follow}(\cdot)$ 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 $\mathsf{Follow}(\cdot)$ set of ⟨*Nested*⟩.

4.2.1 Examples for k' for the Pascal grammar fragment in Figure 4.1

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 [15] 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.

4.2.2 Examples of k^* for the Pascal grammar fragment in Figure 4.1

Consider a state in which the parser is in, where the only rule of ⟨*Program*⟩ is being processed, ⟨*Definitions*⟩ was just replaced on the stack by it's only rule, and ⟨*Body*⟩ 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, ⟨*Labels*⟩'s ϵ-rule will be chosen, thus, ⟨*Labels*⟩ will be popped from the stack.
2. the same will happen with ⟨*Constants*⟩ and ⟨*Nested*⟩.
3. only after seeing nonterminal ⟨*Body*⟩ 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 ⟨*Definitions*⟩, but that doesn't have to be the case all the time. For example, if in our grammar, in ⟨*Program*⟩'s rule, nonterminal ⟨*Parameters*⟩ would follow ⟨*Definitions*⟩, then in the last example, after popping ⟨*Labels*⟩, ⟨*Constants*⟩ and ⟨*Nested*⟩, nonterminal ⟨*Parameters*⟩ 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.

4.3 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 Alg. 4.1 The prediction table construction is depicted in Alg. 4.5

5. 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 re-

Algorithm 4.5 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 $r \in P$                                   // for each grammar rule
  Let $A ::= \alpha \leftarrow r$               // break r into its RHS & LHS
  For $t \in \mathsf{First}(\alpha)$            // for each terminal in First(α)
    If $\mathsf{predict}(A,t) \neq \bot$        // is there a conflict?
      Error                                     // grammar is not LL(1)
    $\mathsf{predict}(A,t) = r$                 // set prediction to rule r
  If $\mathsf{Nullable}(\alpha)$                // α might derive to ϵ
    For $t \in \mathsf{Follow}(A)$              // for each terminal in Follow(A)
      If $\mathsf{predict}(A,t) \neq \bot$      // is there a conflict?
        Error                                   // grammar is not LL(1)
      $\mathsf{predict}(A,t) = r$               // set prediction to rule r
```

altime 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 [16], 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., [9, 24]). And, Incidentally, this JDPS is the one used by Gil and Levy [15], in their proof that the Java type checker can recognize DCFG languages. Using their construction, the translation to Java is rather mechanical.

5.1 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 [23]. 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

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

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 Alg. 5.1.

Algorithm 5.1 A high level sketch of the iterative step of an RLL$_p$

```
1:  Let $[X ::= \alpha \cdot Y\beta] \leftarrow \mathsf{pop}()$   // retrieve parsing state
2:  Let $t \leftarrow \mathsf{next}()$          // examine input once per iteration
3:  If $|Y\beta| = 0$                          // was rule fully parsed?
4:    If $X$ is not the start symbol
5:      $\mathsf{jump}(t)$                      // 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 $Y \in \Sigma$                           // Y is a terminal
12:   If $Y \neq t$                             // read terminal is not anticipated Y
13:     Reject                                  // RLL_p halts in error
14:   else   // anticipated terminal found on input, proceed
15:     $\mathsf{push}(X ::= \alpha Y \cdot \beta)$   // push item with dot advanced
16:     continue // restart, popping this item, and parsing β
17: assert $Y \in \Xi$                          // Y must be a nonterminal
18: Let $a \leftarrow \Delta(Y, t)$             // Δ(Y,t) says what to do with Y
19: If $a = \bot$                               // input t was unanticipated
20:   Reject                                    // RLL_p halts in error
21: If $a = I_1, \ldots, I_\ell$                // a string of ℓ items to push
22:   $\mathsf{push}(I_1, \ldots, I_\ell)$      // note, this is a single push operation
23:   continue                                  // restart with a somewhat deeper stack
24: assert $a$ represents a jump      // t indicates that $Y \overset{*}{\Rightarrow} \epsilon$
25: $\mathsf{jump}(t)$                          // pop this, potentially other items
```

Comparing Alg. 5.1 with the LL-parsing algorithm (Alg. 4.1), 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 Alg. 4.1 is obtained by extracting the symbol Y that follows the dot in the item X in Alg. 5.1.

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 Alg. 4.1).

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.

5.2 The Jump Stack Map Structure

We still need to explain how the jump operations of RLL_p (lines 5 and 25 in Alg. 5.1) are implemented. For this purpose, we construct here the ***Jump Stack Map*** (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 $\bot$ 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[13] definitions (such as TeX [22]), we can rely on the classical method [27] 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 $\bot$ 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,
   $$s(d_i) = \{s(t) \mid t \in d_i\}.$$

[13] 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 the 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 Alg. 5.1) 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.

5.3 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

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

derived from the Pascal BNF fragment (Figure 4.1), 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 ; label ;`** and before reading **`begin`**, the state of the items stack S_0 is

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

when $\langle\mathit{Constant}\rangle$'s item is the top of the stack.

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

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

Breaking down the steps a non-realtime LL$_p$ would perform while reading the input symbol `begin` we get three steps:

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

Advancing upon finishing the parse of $\langle$*Definitions*$\rangle$ in the derivation of $\langle$*Program*$\rangle$, the LL$_p$ would advance to parse $\langle$*Body*$\rangle$.

Consuming since $\langle$*Body*$\rangle$'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 $\langle$*Definitions*$\rangle$ (second step) and lastly push $\langle$*Body*$\rangle$'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 Consolidate$(\cdot,\cdot)$ function (introduced later at Sect. 5.5) 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.

5.4 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 Alg. 5.1).

In particular, the RLL$_p$ generator must compute for each item $i \in I$, the dictionary Jumps(i) which map each token t that triggers a jump with respect to i, to t's jump value (type J). Alg. 5.2 is the algorithm for doing so.

Algorithm 5.2 Function 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 $[A ::= \alpha \cdot Y_1 \ldots Y_n] \leftarrow i$        // break i into components
Let $d \leftarrow \emptyset$                              // initialize return variable
For $j = 2, \ldots, n$                          // for all symbols in suffix of i
    If not Nullable$(Y_2 \ldots Y_{j-1})$              // continue to build d?
        break
    For $t \in$ First$(Y_j)$   // symbols that might cause jump in $Y_1$
        If $d(t) = \bot$         // $1^{st}$ update of key t in dictionary d?
            $j = [A ::= \alpha Y_1 \ldots \cdot Y_j \ldots Y_n]$    // the jump address
            // handle the Push after Jump phenomena
            $d(t) = Consolidate(j, t)$                    // update d(t)
Return d
```

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

Assume that $Y_2 \ldots Y_{j-1}$ is nullable and that $t \in$ 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 $Y_2 \ldots Y_{j-1} \overset{*}{\Rightarrow} \epsilon$ 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:

- $Y_2 \ldots Y_{j-1} \overset{*}{\Rightarrow} \epsilon$ and t is the first token in the derivation of Y_j i.e.,
$$Y_j \overset{*}{\Rightarrow} t\gamma.$$
- $Y_2 \ldots Y_{j'-1} \overset{*}{\Rightarrow} \epsilon$ 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 Consolidate$(\cdot,\cdot)$ (defined in Alg. 5.3) to overcome the push after jump phenomena mentioned in Sect. 5.3.

Table 3 presents some of the values computed by Alg. 5.2 on our running example defined in Figure 4.1

The first row of Table 3 shows that in case $\langle$*Program*$\rangle$'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($\langle$*Body*$\rangle$), 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 $\langle$*Constant*$\rangle$ 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 shows that in case a terminal t is not in

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.

5.5 Consolidating Push Operations

Recalling Def. 1, 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 Alg. 5.3 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 4.1) are presented in Table 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

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

Table 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 4.1

Item i	Terminal t	$\mathsf{Consolidate}(i, t)$
$[\langle\mathit{Body}\rangle ::=$ **begin** $\cdot$ **end**$]$	**end**	$[\langle\mathit{Body}\rangle ::=$ **begin end** $\cdot]$
$[\langle\mathit{Constants}\rangle ::=$ **const** $\langle\mathit{Constant}\rangle \cdot \langle\mathit{MoreConstants}\rangle]$	**;**	$[\langle\mathit{Constant}\rangle ::=$ **;** $\cdot]$ $[\langle\mathit{MoreConstants}\rangle ::= \cdot \langle\mathit{Constant}\rangle \langle\mathit{MoreConstants}\rangle]$ $[\langle\mathit{Constants}\rangle ::=$ **const** $\langle\mathit{Constant}\rangle \cdot \langle\mathit{MoreConstants}\rangle]$
$[\langle\mathit{Definitions}\rangle ::= \cdot \langle\mathit{Labels}\rangle \langle\mathit{Constants}\rangle \langle\mathit{Nested}\rangle]$	**const**	$[\langle\mathit{Constants}\rangle ::=$ **const** $\cdot \langle\mathit{Constant}\rangle \langle\mathit{Constants}\rangle]$ $[\langle\mathit{Definitions}\rangle ::= \langle\mathit{Labels}\rangle \cdot \langle\mathit{Constants}\rangle \langle\mathit{Nested}\rangle]$

Table 4: Example values for the $\mathsf{Consolidate}(\cdot, \cdot)$ functions on the grammar defined in Figure 4.1

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 $\langle\mathit{MoreConstants}\rangle$ because it is yet to be fully parsed.

Upon seeing nonterminal $\langle\mathit{MoreConstants}\rangle$ and terminal "**;**" the LL_p would choose rule

$$\langle\mathit{MoreConstants}\rangle ::= \langle\mathit{Constant}\rangle\, \langle\mathit{MoreConstants}\rangle,$$

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 LL_p's prediction table rule

$$\langle\mathit{Constant}\rangle ::= \textbf{;}$$

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**, $\langle\mathit{Labels}\rangle$ is derived to ϵ. Thus, the last item in the output, is the input item with the dot following nonterminal $\langle\mathit{Labels}\rangle$ instead of preceding it, $\langle\mathit{Constants}\rangle$ rule is then chosen as we seen in the second example.

5.6 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.

Reexamining the code of the RLL_p (Alg. 5.1) we see that all such information is contained in function Δ. Therefore, the core of the RLL_p generator is Alg. 5.4 that computes Δ. Table 5 provides some of Δ's values for our running example defined in Figure 4.1.

In the first two rows of Table 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 $\mathsf{Follow}(\cdot)$ set ($\langle\mathit{Labels}\rangle$ and $\langle\mathit{Nested}\rangle$ correspondingly), causing Δ returns a jump operation.

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

The fourth row of Table 5 presents a case in which the input terminal is in the $\mathsf{First}(\cdot)$ set of the symbol following the dot, thus, a push operation is returned, with the values of the corresponding $\mathsf{Consolidate}(\cdot, \cdot)$ function.

The last row of the table presents a special case, in which the input item's dot is not at the end of $\langle\mathit{Definition}\rangle$'s rule, and the operation is a jump(**begin**) operation. This happens because **begin** is not in $\mathsf{First}(\langle\mathit{Definition}\rangle)$ and is in $\mathsf{Follow}(\langle\mathit{Definition}\rangle)$ (it is also required the $\langle\mathit{Definition}\rangle$ 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 $\mathsf{push}(\cdot)$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, Alg. 5.4 implicitly consults the function $\mathsf{Jumps}(\cdot)$ (recall that $\mathsf{Jumps}(\cdot)$ 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 $\mathsf{Consolidate}(\cdot)$ (Alg. 5.3) to compute L, the consolidated list of push operations. List L will be later read by RLL_p (line 18 in Alg. 5.1).

6. Conclusions

We presented the first practical algorithm for the automatic generation of a fluent API for JAVA from LL grammars. Future research is extending the achievement to LR grammars.

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**
$[\langle Labels\rangle ::=$ `label` $\langle Label\rangle \cdot \langle MoreLabels\rangle]$	`;`	push(Consolidate($[\langle Labels\rangle ::=$ `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 5: Example values for the prediction table Δ on the grammar defined in Figure 4.1

Algorithm 5.3 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.

```
$[X ::= \alpha \cdot Y\beta] \leftarrow i$                  // break i into its components
Let $L \leftarrow \emptyset$                                  // initialize return value
While $Y \notin \Sigma$                               // loop while Y is a nonterminal
  push$(L, [X ::= \alpha \cdot Y\beta])$                // currently parsing Y
  $r \leftarrow \delta(Y, t)$                                // next rule to apply
  $[B ::= X_1 \ldots X_m] \leftarrow r$                 // break r into components
  If $X_1 \ldots X_m = \epsilon$                         // δ returned an ε-rule
    While exhausted(peek$(L)$)               // pop all exhausted rules
      pop$(L)$
    $[X ::= \alpha \cdot Y\beta] \leftarrow$ pop$(L)$         // Y was just fully parsed
    push$(L, [X ::= \alpha Y \cdot \beta])$                  // advance the rule
  else                                          // δ returned a non ε-rule
    push$(L, [B ::= \cdot X_1 \ldots X_m])$          // we now turn to parse r
  $[X ::= \alpha \cdot Y\beta] \leftarrow$ pop$(L)$         // break i into its components
push$(L, [X ::= \alpha Y \cdot \beta])$                          // 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.

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.

Algorithm 5.4 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 $i \in I$                                        // for each item
  Let $[A ::= \alpha \cdot Y\beta] \leftarrow i$         // break i into components
  If $Y \in \Sigma$                            // Δ doesn't handle terminals
    continue                                     // handle next item.
  For $t \in \Sigma$                                 // calculate entry Δ[i, t]
    If $t \in$ First$(Y\beta)$             // t is consumed while parsing i
      Let $L \leftarrow$ Consolidate$(i, t)$
      Let $\Delta[i, t] \leftarrow$ [push$(L)$]          // a push operation
    else if $t \in$ Follow$(A)$        // t is consumed after parsing i
      // Obtain jumps dictionary and store in Δ:
      Let $\Delta[i, t] \leftarrow$ [jump$(t)$]           // a jump operation
```

As the two phenomena happen, advanced features such as automatic production of ASTs (which does not expected to pose a theoretical hurdle), and EBNF support should become even more useful.

References

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

[6] 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.

[7] 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.

[8] 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.

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

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

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

[12] 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.

[13] 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).

[14] 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.

[15] J. Y. Gil and T. Levy. Formal language recognition witw 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.

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

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

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

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

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

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

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

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

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

[25] 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.

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

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

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

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

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