Grammar Type for String

Citation
Xu, Licheng. 2023. Grammar Type for String. Master's thesis, Harvard University Division of Continuing Education.

Permanent link
https://nrs.harvard.edu/URN-3:HUL.INSTREPOS:37377219

Terms of Use
This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

Share Your Story

The Harvard community has made this article openly available.
Please share how this access benefits you. Submit a story .

Accessibility

Grammar Type for String

Licheng Xu

A Thesis in the Field of Software Engineering

for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

November 2023

Copyright 2023 Licheng Xu

Abstract

*String*s are ubiquitous in computer programs. Both the correctness and the security of programs that use *String*s often rely on them not being arbitrary *String*s, but belonging to specific sets of *String*s. However, this restriction is often not enforced, let alone clearly specified. To remediate this issue, this thesis creates a language extension on top of the standard Java language. The language extension introduces a Grammar Type that is a subtype of *String* but conforming to the regex expression specified for each Grammar Type. For example, *String*[[“*a*∗*b*”]] is a Grammar Type that represents all *String*s that are any number of *a*’s followed by a *b*. When we declare or cast a variable as *String*[[*some_regex*]], the variable has to be a *String* conforming to that *some_regex*, otherwise it would be either a compile time error or runtime error depending on the situation. This makes the Java type system more powerful, as we can now inherently validate *String*s with any pattern we define using a regex. At the same time, since Grammar Types compile down to *String*s, they inherit all the functionality of *String*s. With these advantages, Grammar Types can be used in various types of applications that need input validation, like validating email addresses, validating URLs, or mitigating SQL injection attacks.

Dedication

This effort is dedicated to my wife Yahui Zhu, my parents Chengdong Xu and Jing Wei, and my grandfather Xirong Xu for all of your love and support along the way.

Acknowledgements

I would like to thank Professor Stephen Chong who graciously agreed to be my thesis director. Thank you for the tremendous help and support throughout the thesis journey, and your infinite knowledge that provided me guidance whenever I needed it.

I would like to thank my research advisor Professor Hongming Wang, my thesis coordinator and academic advisor, and all other Harvard faculty and staff for making this process as smooth as possible.

I would like to thank the friends I made during my time at Harvard: Narek Asadorian, Alyssa Bédard, Rubén Martínez, Ben Tan, Forum Sanjanwala, Anusha Datta, and Anthony Jesmok. You guys brought joy to my life.

Table of Contents

Table of Contents

List of Figures

List of Tables

List of Code

Chapter I: Introduction

1.1 Motivation 1

1.2 Goal 6

1.3 Thesis Outline 7

Chapter II: Related Work

Chapter III: Design Overview

Chapter IV: Implementation

4.1 Lexing 16

4.2 Parsing 17

4.3 AST 19

4.4 Types and Type System 22

4.5 Disambiguating 25

4.6 Subtyping 27

4.7 Type Casting 31

Chapter V: Regex Containment Problem

5.1 Regex to Syntax Tree 38

5.2 Syntax Tree to NFA 40

5.3 NFA to DFA 43

5.4 DFA Containment Problem 45

Chapter VI: Example Use Cases

6.1 Valid Email Addresses 52

6.2 Valid User Inputs 53

6.3 Valid URLs 55

Chapter VII: Conclusion

References

List of Figures

1 Compilation process overview 14

2 Polyglot architecture overview 15

3 Dependency between goals 16

4 Types of AST *Node*s in Polyglot 20

5 Extension chaining (Java 1.4 → Java 5 → Java 7) 20

6 Syntax tree for regex $a{*}b|c$ 39

7 NFA for regex $a{*}b|c$ 42

8 Fundamental NFA structures for different *SyntaxTreeNode* types . . 43

9 DFA for regex $a{*}b|c$ 44

10 Total DFA for regex $a{*}b|c$ 46

11 Complement of total DFA for regex $a{*}b|c$ 47

12 $DFA3$ and $DFA2$ 48

13 Intersection of $DFA3$ and $DFA2$ 50

List of Tables

1 Sample data in *users* table 2

2 Example programs to showcase ambiguity removing 26

3 Example programs to showcase subtyping 29

4 Example programs to showcase type casting 33

5 Test programs to validate email addresses 53

6 Test programs to validate redirect URLs 56

List of Code

1 Application code snippet with SQL injection vulnerability 3

2 Example implementation of the *Username* custom class 5

3 Added lexing rules in grammar.flex 17

4 Definitions of DBLBRACK and DBRBRACK in grammar.ppg 17

5 Definition of *class_or_interface_type* in grammar.ppg 18

6 Definition of *grammar* method in grammar.ppg 18

7 New rule for *class_or_interface_type* in grammar.ppg 18

8 New rule for *cast_expression* in grammar.ppg 19

9 GrammarNodeFactory_c.java 21

10 GrammarType.java 22

11 Excerpt from GrammarType_c.java 23

12 GrammarTypeSystem.java 24

13 Excerpt from GrammarTypeSystem_c.java 25

14 GrammarTypeNodeExt.java 26

15 Implementation of *isSubtypeImpl* in GrammarType_c.java 29

16 Implementation of *typeCheckOverride* in StringLitExt.java 31

17 Implementation of $isCastValid$ in GrammarTypeSystem_c.java . . . 33

18 Original code in Grammar language 34

19 Output code in Java language 34

20 RuntimeCastChecker.java 35

21 Implementation of $translate$ in GrammarCastExt_c.java 36

22 Excerpt from SyntaxTree.java 38

23 Definitions of FA, $Node$, $Edge$, and $NFAEdge$ 41

24 Structure of $DFANode$ 43

25 Structure of $IntersectionDFANode$ 48

26 Original Java code with no input validation 53

27 Same code rewritten in the Grammar language 55

Chapter I.

Introduction

Types are a fundamental part of programming languages because they both convey the meaning of programs, and also provide validation on programs through type checking. When we define a variable to be a *String* for example, we can have a peace of mind that the variable will always be a *String* throughout the program, and avoid an entire class of errors. Because types are so useful, sometimes we find ourselves wanting to define more types to capture the specific classes of data we want to represent. This would provide us with both more clear and concise meaning of the program, and also offer protection to certain unexpected harm. For this thesis, we would focus on the Java programming language. But the same concept and techniques are applicable for most other languages.

1.1. Motivation

One of the most commonly used types in Java is *String*. However, in most cases, when a program annotates a variable as *String*, it really only makes sense for the variable to take on values that are a specific subset of all possible *String*s. For

example, it would be great if all the variables that would store email addresses are of a type that can only accept *String* values of valid email addresses. Similarly, an example use case from work is that there might be certain restrictions that a redirect URL needs to follow. It would be great if all the variables storing redirect URLs are of a type that can only take valid redirect URLs according to specific rules.

In addition, as programmers we often need more granular *String* types not just for convenience, but for security of the program. Consider for example an application that stores user information (username, password, age, social security number) in a *users* table. Table 1 shows information stored for 2 users ben and jerry.

username	password	age	social_security_number
ben	ben_password	3	123-45-6789
jerry	jerry_password	5	987-65-4321

Table 1: Sample data in *users* table

The application allows users to query their information by first prompting user to enter their username, and then prompting user to enter their password. With the entered username and password, the application then constructs a SQL query to return all the information of the user. For example, when user Ben wants to get his information, he would type *ben* at the first prompt for username, and *ben_password* at the second prompt for password. The application would then construct and execute the following SQL query to fetch user information: *SELECT * FROM users WHERE username = "ben" AND password = "ben_password"*. If username or password was wrong, the application would return no data, otherwise matching user data would be returned. For illustration purpose a code snippet of this

example application is provided below.

```
// Establishing a database connection
final Connection connection = DriverManager.getConnection(url, mysqlUsername,
    mysqlPassword);

// Creating a Scanner object to read user input
final Scanner scanner = new Scanner(System.in);

// Prompting the user for an SQL command
System.out.print("Enter username: ");
final String username = scanner.nextLine();
System.out.print("Enter password: ");
final String password = scanner.nextLine();
final String sql = "SELECT * FROM users WHERE username = \"" + username + "\"
    AND password = \"" + password + "\"";
System.out.println(sql);

// Creating a SQL statement
final Statement statement = connection.createStatement();

// Executing the user's SQL command
final boolean hasResultSet = statement.execute(sql);
```

Listing 1: Application code snippet with SQL injection vulnerability

The idea of course is that a user would only be able to query their own information due to the username and password requirements. However, the reality is that this application is susceptible to SQL injection attacks. Specifically, consider the following input: *anything*” *OR* “1” = “1 for username field, and *anything*” *OR* “1” = “1 for password field. The constructed SQL query would be *SELECT* ∗ *FROM users WHERE username* = “*anything*” *OR* “1” = “1” *AND password* = “*anything*” *OR* “1” = “1”. Due to the *OR* “1” = “1” conditions, the query effectively becomes *SELECT* ∗ *FROM users WHERE true AND true*, which unfortunately would return all users’ information. There are a couple of ways one might think of to mitigate this:

We could perform additional checks after reading users' input. Specifically, we can define a method $private\ boolean\ isUsernameValid(final\ String\ username)$ and another method $private\ boolean\ isPasswordValid(final\ String\ password)$ and call these methods after reading user's supplied username and password. There are a couple of drawbacks with this approach however, the first being we need to add extra code to check the validity of user input for each field we get from the user. The second drawback is that we have to rely on application developers to remember to always add the input validation code for each input, which becomes error prone. The third drawback is that we would have to write one validation method for each type of user input.

Another thought to mitigate this vulnerability is to define a custom class representing user input. For example, we can create a custom class $Username$ to represent usernames (shown in Listing 2), and another custom class $Password$ to represent passwords.

We would construct $Username$ and $Password$ objects when we read user input by doing something like $final\ Username\ u = new\ UserName(scanner.nextLine());$. If the username entered was invalid, we would throw $IllegalArgumentException$. Otherwise, we call $getUsername()$ on the $Username$ object to retrieve the actual $String$ input, and concatenate it with the SQL query template. But with this approach, there are again some shortcomings. Firstly, with a custom class, the $String$ validity check can only happen at runtime. For example, if we had a new variable

```
public class Username {
    private final String username;

    public Username(final String username) {
        this. username = username;
        if (!isUsernameValid()) {
            throw new IllegalArgumentException("invalid username");
        }
    };

    private boolean isUsernameValid() {
        // perform validity check on username and return whether if it's
    valid
    }

    public String getUsername() {
        return username;
    }
}
```

Listing 2: Example implementation of the *Username* custom class

declaration like $final\ Username\ foo = new\ Username(``anything"\ OR\ ``1" = ``1")$, we would only catch the problem during runtime, instead of earlier during compile time. Another shortcoming with this approach is that every time we need to use the *Username*, we need to call an extra method to retrieve the *String* content inside the wrapper object. Last but not least, similar to the first approach, with custom classes we would need one custom class for each type of *String* we need to represent, since the validation logic is different for different *String* types.

All of these leave something to be desired: another approach to address the vulnerability (and more generally, the need for more precise *String* types), but at the same time overcome the drawbacks of the aforementioned approaches.

1.2. Goal

The goal is to be able to define different *String* types based on different "grammars". For example, we can have a *String* type to represent passwords, a *String* type to represent email addresses, a *String* type to represent valid SQL queries, etc. And actually, that's still not enough. For example, since different applications have different password requirements (ex. different password length requirements, capitalization requirements), we need one *String* type to represent each of these different password requirements. But we also don't want to create a custom validation checker method or a custom class for each "grammar".

This is where Grammar Type for String comes into play. We would introduce a new type called Grammar Type, in a new language called the Grammar language. Grammar Type would compile down to vanilla Java *String*, so no custom classes or methods are needed and we can use them just as we use *String*s. But at the same time, Grammar Types take custom regular expressions to represent different "grammars". To achieve this, we introduce the syntax *String*[[*some_regex*]] to represent a Grammar Type taking the regular expression *some_regex*. For example, a Grammar Type that takes a regex *email_regex* would represent and only accept *String*s that are valid email addresses. To ensure this, there will be several typing related functionalities supported in this project. (i) subtyping validity will be checked during compile time: Firstly, Grammar Type is a subtype of *String*. Secondly, a *String* literal is a subtype of a Grammar Type if and only if the *String* literal matches the regex spec-

ified in the Grammar Type. Thirdly, between 2 Grammar Types, $GrammarType1$ is a subtype of $GrammarType2$, if and only if the regex specified for $GrammarType1$ is a sublanguage of the regex specified for $GrammarType2$. (ii) casting validity will be checked during compile time and also possibly runtime: For a cast of variable from type T to a Grammar Type, during compile time, we first check if the Grammar Type is a subtype or superset of T. If neither, we would throw a compile-time error. Otherwise, if the cast passed compile-time check, we then rewrite the output Java code to perform a runtime check on the cast.

1.3. Thesis Outline

In Chapter II, we discuss prior work related to analyzing *String* variables in programs, approximating *String* values, or building pluggable type systems.

In Chapter III, we give a high level overview of the design, including the syntax we choose for Grammar Types, how we plan to implement the language extension, and how we plan to test our implementation.

In Chapter IV, we first provide a quick recap of a typical compilation process. We then briefly introduce the Polyglot compiler framework (Nystrom et al., 2003), on top of which we build our Grammar Type and the underlying Grammar language. Finally we dive into the implementation details by explaining how we handle different aspects of the compilation process for the Grammar language. These include lexing, parsing, AST, types and type system, disambiguating, subtyping, and type casting.

In Chapter V, we zoom into the regex containment problem, which is a prerequisite for determining subtyping relationships between Grammar Types. We do this in 4 phases: converting the regex to syntax tree, converting the syntax tree to NFA, converting the NFA to DFA, and finally deciding whether one DFA "contains" another DFA.

In Chapter VI, we validate our work by testing our implementation against several different scenarios: validating email addresses, validating user inputs for SQL queries, and validating whether a redirect URL is allowed for a real life application.

In Chapter VII, we conclude by summarizing the learnings and achievements of this work, and discuss any potential future work.

Chapter II.

Related Work

Due to the omnipresence of *String*s (and *String*-related bugs) in programs, a multitude of work has been devoted into finding or preventing such problems.

One stream of work was to run empirical study to categorize these bugs (Barlas et al., 2022; Eghbali & Pradel, 2021). Specifically, Eghbali and Pradel (Eghbali & Pradel, 2021) performed an empirical study of *String*-related software bugs. They investigated 204 *String*-related bugs and found that the majority of them were due to incorrect *String* literals or incorrect regular expressions. Although most of them only required minor fixes, they found that a popular static checker missed 203 of the total 204 bugs. This work is useful for highlighting the vulnerability of *String*-related programs and the difficulty of enforcing correctness on such programs. However it is simply an empirical study and does not systematically modify or improve the type system.

Another stream of work performs static analysis to evaluate the correctness of programs (Costantini et al., 2015; Kim et al., 2013; Costantini et al., 2011; Christensen et al., 2003; Wassermann & Su, 2007). Specifically, Christensen et al. (Christensen

et al., 2003) devised a way to approximate the possible values of each *String* expression in a Java program. They first converted a Java program to a flow graph, then converted the flow graph to a context free grammar, then constructed a multi-level automaton from the CFG, and finally deduced regexes to approximate possible values of *String* expressions in the original Java program. This is useful, however it differs from the current thesis in that it is doing static analysis on a given program, instead of modifying the type system of the language directly and preventing invalid *String* expressions from the source.

In similar veins, Wassermann and Su (Wassermann & Su, 2007) also used static analysis to find SQL injection attacks in the PHP language. The authors used CFGs to approximate the set of possible SQL queries, and tracked information flow from the source (user input) to the end query.

Another approach was to approximate or limit the values *String* variables can take (Kim et al., 2014; Minamide, 2005; Kiezun et al., 2009; Tabuchi et al., 2003; Thiemann, 2005; Cook & Rai, 2005). For example, Tabuchi et al. (Tabuchi et al., 2003), presented a new lambda calculus language λ^{re} that typed *String*s by regular expressions. Even though the authors provided some type inferencing rules, there was no algorithm provided. In addition, using lambda calculus isn't very useful in real world applications. In contrast, the proposal in this thesis uses the Java programming language, which is one of the most commonly used programming languages.

In the same spirit, Thiemann (Thiemann, 2005) created a polymorphic type

system such that *String* constraints are modeled as context free grammar containment. The type system guarantees that the value of a *String* expression belongs to the given context-free language. However, even though the solutions are sound, they are incomplete due to the inherent undecidability of the broader problem of context-free language inclusion.

In a somewhat novel approach, Cook and Rai (Cook & Rai, 2005) proposed to write database queries using object-oriented classes and methods instead of *String*s. These statically typed objects enable compile-time type checking, but only works for typed languages, and are not very intuitive for developers who are used to writing *String* queries.

Last but not least, one other branch of research was to enhance or modify the programming language's type system (Andreae et al., 2006; Greenfieldboyce & Foster, 2007; Ali et al., 2008; Santino, 2016; Papi et al., 2008). The most notable one of them is the Checker Framework (Papi et al., 2008). It was developped as a pluggable type system for Java. Users of the Checker Framework can use predefined or define custom checkers using annotations (ex. @*NonNull*, @*Nullable*, etc), and the framework would enhance the Java type system by running respective compile-time checks. This is very useful and like this paper, it also modifies the Java type system. The difference is that the Checker Framework can only perform compile-time checks, so it won't be able to catch runtime exceptions, ex. if a *String* that's supposed to be a phone number got assigned to an email address during runtime.

Chapter III.

Design Overview

We will define a new language extension in Polyglot called Grammar in order to support Grammar Type for String. The added syntax would be $String[[some_regex]]$ to represent a Grammar Type that represents $String$s conforming to the specified regular expression $some_regex$. Grammar Type would still be a subtype of $String$, but with added functionality and type checks. Below is the typing rule for Grammar Type:

$$\frac{x : String \epsilon E \quad matches(regex_foo,\ x)}{E \vdash x : String[[regex_foo]]}$$

The rule says: if the variable x is a $String$ in the typing environment E, and the $String$ value of the variable x matches $regex_foo$, then, the variable x is of Grammar Type $String[[regex_foo]]$.

In order to type check Grammar Types when assigning or casting variables, we also need to define subtyping relationships for Grammar Types. Below are the subtyping rules:

$$\frac{}{String[[any_regex]] <: String}$$

$$\frac{regex1 <: regex2}{String[[regex1]] <: String[[regex2]]}$$

The first rule says that Grammar Types are always subtypes of *String*. The second rule says that a Grammar Type is a subtype of another Grammar Type, if and only if the regex specified for the first Grammar Type is a sublanguage of the regex specified for the second Grammar Type.

With these typing and subtyping guidelines, we will extend the Polyglot compiler to build our Grammar language by implementing lexing, parsing, AST, types and type system, disambiguating, subtyping and type casting.

Finally, we will test our Grammar language by writing programs utilizing Grammar Types, and see if they are able to realize the benefits we were hoping for, such as addressing the vulnerability concerns from SQL injection attacks, or generalizing input "grammar" validation.

Chapter IV.

Implementation

Compilation is the process of translating a target language to a source language. Figure 1 below depicts a typical compilation process (Chong, 2019). The source code first goes through lexing to become tokens, then we parse the tokens into abstract syntax trees. The next step is elaboration which performs type checking, and the last step is code generation, that's when we write out the code in the target language.

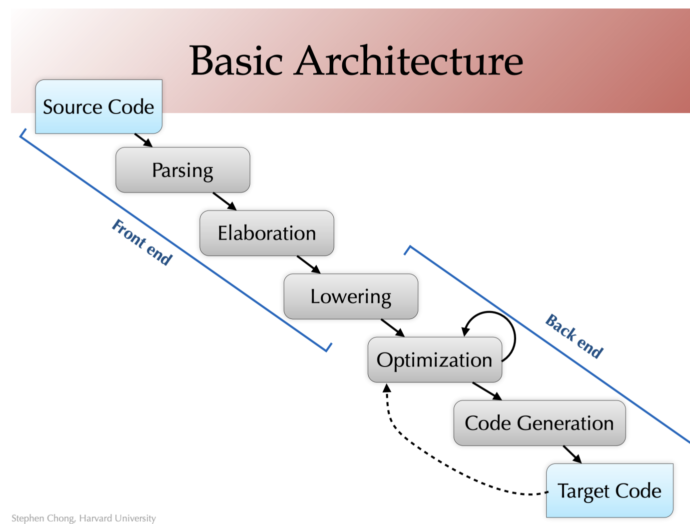

Figure 1: Compilation process overview

Specifically for this project, we want to introduce a new Grammar language

and compile it down to plain Java code by extending the Polyglot compiler. Polyglot is a highly extensible compiler frontend for the Java programming language. The architecture of Polyglot can be simplified as Figure 2.

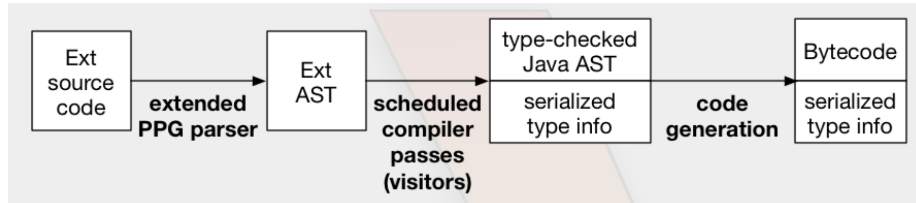

Figure 2: Polyglot architecture overview

https://www.cs.cornell.edu/Projects/polyglot/pldi14/tutorial/architecture/

First, the input to Polyglot is the source code in the extension language. Then, Polyglot uses an extended version of the CUP parser called PPG to parse the source code into ASTs. Each node in the AST also might have a doubly linked list of Exts (extension objects) attached to it. These Exts contain the states and operations for each corresponding extension layer. After building the Ext AST, Polyglot runs a series of compiler passes. Each compiler passes is run on the AST to transform it into a new AST. The compiler passes are run to satisfy "goal"s. Figure 3 shows the list of standard goals in Polyglot for Java 1.4. Written in parentheses are the passes to satisfy each goal.

We see that goals have dependencies on each other. The end goal is CodeGenerated, which means producing vanilla Java code. And to achieve that goal, we need to first achieve a series of other goals starting from Parsed which parses the source code into AST, and including TypeChecked, which performs static type checking on the AST.

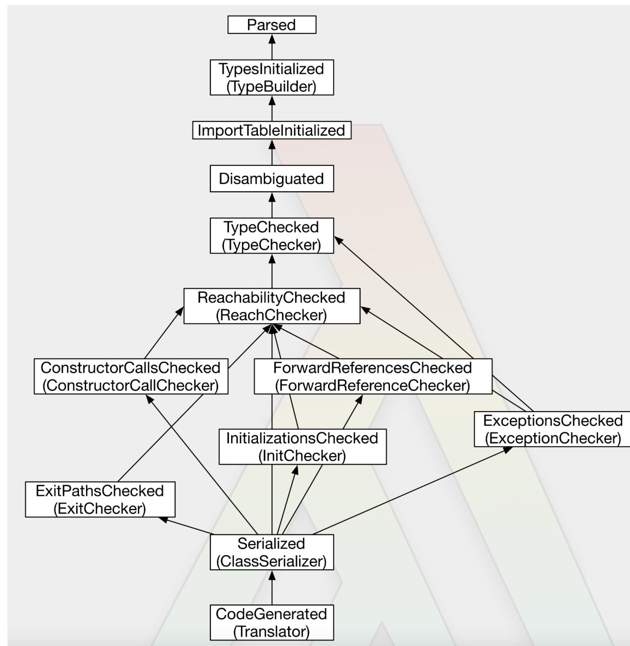

Figure 3: Dependency between goals

https://www.cs.cornell.edu/Projects/polyglot/pldi14/tutorial/architecture/

4.1. Lexing

Lexing is the first step in the compilation process. In this step, we need to convert the sequence of characters in the Grammar language into a sequence of tokens for the parser to parse. For example, the characters i and f would be converted to the token IF. Polyglot uses the JFlex lexical analyzer generator to generate the lexer. The default grammar.flex file already lexes the regular Java language, so we just need to add support for lexing our extension. To avoid any ambiguity with the existing Java grammar, we decided to choose [[and]] characters as delimiters for Grammar Types. We will call [[DBLBRACK (for double left bracket) and call]] DBRBRACK (for double right bracket).

```
1  "[["   { return op(sym.DBLBRACK);  }
2  "]]"   { return op(sym.DBRBRACK);  }
```

Listing 3: Added lexing rules in grammar.flex

To add it to the lexer, we simply add the lines in Listing 3 to the $<YYINITIAL>$ section of grammar.flex. Now the lexer will translate [[to the token DBLBRACK and translate]] to the token DBRBRACK.

4.2. Parsing

After the lexer converts the sequence of characters in the Grammar language into a sequence of tokens, we are ready to parse the sequence of tokens into abstract syntax trees (ASTs) using PPG (Polyglot Parser Generator). PPG is extensible in the sense that it contains the parser for regular Java (java12.cup), so for our extension we only need to add the additional rules for parsing Grammar Types. We decided to use the syntax $String[[regex]]$ to highlight the fact that Grammar Type is an extension of $String$, and to allow Grammar Type to take in a regex. We first add definitions for the DBLBRACK and DBRBRACK tokens (shown in Listing 4).

```
1  terminal Token DBLBRACK;
2  terminal Token DBRBRACK;
```

Listing 4: Definitions of DBLBRACK and DBRBRACK in grammar.ppg

Then, because we are modifying the syntax for class types, we first look at the regular Java parser for rules regarding class types. Listing 5 shows that the lexer defines $class_or_interface_type$ to be a name, where name is just an identifier.

We need to extend the syntax for $class_or_interface_type$ to include $String[[regex]]$.

```
class_or_interface_type ::=
                    // TypeNode
        name:a
            {: RESULT = a.toType(); :}
```

Listing 5: Definition of *class_or_interface_type* in grammar.ppg

To do this, we first add a new method *grammar* (shown in Listing 6).

```
public TypeNode grammar(TypeNode n, String s) throws Exception {
    return nf.GrammarTypeNode(n.position(), (AmbTypeNode) n, s);
}
```

Listing 6: Definition of *grammar* method in grammar.ppg

The *grammar* method takes in a *TypeNode* and a *String* (will be the regex), and returns a *GrammarTypeNode*. With the *grammar* method in place, we are now ready to extend the syntax for the parser by adding the following rule (shown in Listing 7).

```
extend class_or_interface_type ::=
                    // TypeNode
        class_or_interface_type:a DBLBRACK STRING_LITERAL:s DBRBRACK
            {: RESULT = parser.grammar(a, s.getValue()); :}
    ;
```

Listing 7: New rule for *class_or_interface_type* in grammar.ppg

This extends the existing definition of *class_or_interface_type* to include a new syntax. Namely, if the sequence of tokens are: *class_or_interface_type*, DBLBRACK, STRING_LITERAL (the regex), DBRBRACK, then it's also a *class_or_interface_type*, and the result is a *TypeNode* obtained by calling the *grammar* method defined earlier. However, as a counter example, if the input program was $int[[“a*b”]]\ i;$, this would fail the parsing step with the error "unexpected operator [[". This is because *int* is not a *class_or_interface_type*, and so the sequence of tokens doesn't match any

of the rules defined.

class_or_interface_type is not the only thing we are modifying however. This is because the Grammar language also supports casting, namely, expressions like (*String*[[*regex*]]) *var*. If we dig through the existing definition of *cast_expression* in java12.cup, this case is not covered. So similar to *class_or_interface_type*, we extend *cast_expression* by adding the following rule (shown in Listing 8).

```
1 extend cast_expression ::=
2          LPAREN:p class_or_interface_type:a DBLBRACK STRING_LITERAL:s
      DBRBRACK RPAREN unary_expression_not_plus_minus:b
3             {: RESULT = parser.nf.Cast(parser.pos(p, b, a),
4                            parser.grammar(a, s.getValue()), b); :}
5     ;
```

Listing 8: New rule for *cast_expression* in grammar.ppg

The *Cast* method in the node factory takes in a position, a *TypeNode*, and an expression. The *TypeNode* is the type we are casting to, so in this case we call our *grammar* method to get a *TypeNode* representing Grammar Type. The expression is the expression being casted, so we pass along *b*, the *unary_expression_not_plus_minus*.

4.3. AST

As alluded to in the previous section, the output of the parsing phase is an AST. The AST in Polyglot is composed of AST *Node*s, which are created by the *NodeFactory* object associated with the current language extension. The *Node* interface is extended by several interfaces to represent different constructs in an AST. Figure 4 shows the types of AST Nodes represented in Polyglot.

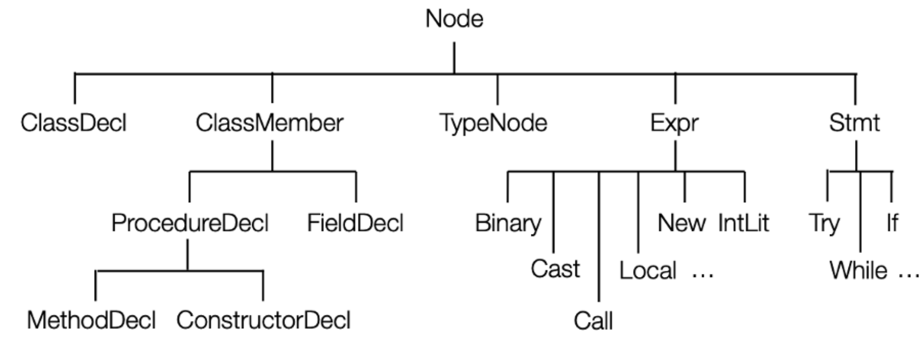

Figure 4: Types of AST *Node*s in Polyglot

https://www.cs.cornell.edu/Projects/polyglot/pldi14/tutorial/slides/Polyglot_Tutorial.pdf

Each *Node* can also have a chain of extension objects (*Ext*s) that contain states and operations associated with each particular extension layer. Each *Ext* has a pointer to the root *Node* object, a pointer to the previous *Ext*, and a pointer to the next *Ext*. Figure 5 illustrates the extension chaining for Java 7 extension on top of Java 5 extension on top of Java 1.4 base compiler.

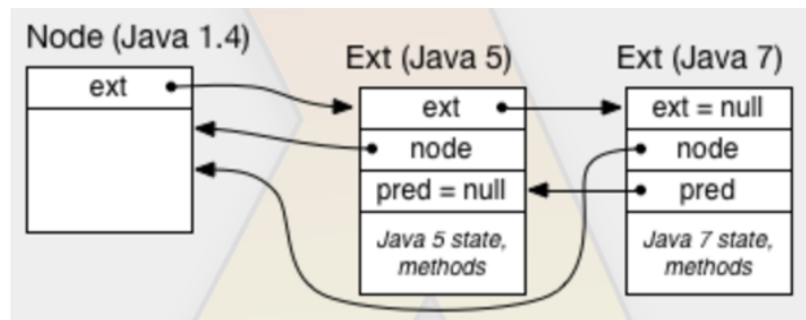

Figure 5: Extension chaining (Java 1.4 → Java 5 → Java 7)

https://www.cs.cornell.edu/Projects/polyglot/pldi14/tutorial/architecture/

For our Grammar Type extension, we will define a *GrammarExt* class as the *Ext* for Grammar Types. *GrammarExt* is extended by 3 classes to represent 3 different cases of *Ext*s in the Grammar language. The first is *StringLitExt* to represent

String literals. The second is *GrammarCastExt* to represent cast operations. The third is *GrammarTypeNodeExt* to represent Grammar *TypeNode*s. In order to do that, *GrammarTypeNodeExt* stores an additional state, namely the regex of the Grammar Type. We create these *GrammarExt*s in a *GrammarExtFactory* (that extends from *polyglot.ast.ExtFactory*). We also create a *GrammarNodeFactory* (that extends from *polyglot.ast.NodeFactory*) to create AST Nodes corresponding to Grammar Types. So analogous to the Java 1.4 → Java 5 → Java 7 example above, for Grammar Types the root node would be an *AmbTypeNode*, and it will have a *GrammarExt* chained to it. Listing 9 is a code snippet of *GrammarNodeFactory* to give an overview of the process.

```
1  /**
2   * NodeFactory for grammar extension.
3   */
4  public class GrammarNodeFactory_c extends NodeFactory_c implements
5          GrammarNodeFactory {
6      public GrammarNodeFactory_c(GrammarLang lang, GrammarExtFactory
       extFactory) {
7          super(lang, extFactory);
8      }
9
10     @Override
11     public GrammarExtFactory extFactory() {
12         return (GrammarExtFactory) super.extFactory();
13     }
14
15     @Override
16     public AmbTypeNode GrammarTypeNode(Position pos, AmbTypeNode base, String
        regex_str) {
17         final Ext ext = extFactory().extGrammarTypeNode(regex_str);
18         return new AmbTypeNode_c(pos, base.qual(), base.id(), ext);
19     }
20 }
```

Listing 9: GrammarNodeFactory_c.java

We can see that the *GrammarNodeFactory* can create a *GrammarExtFactory*

to create *GrammarExt*s. And in the *GrammarTypeNode* method (which was the method called from the parser in the previous section), we create an *AmbTypeNode*, and compose it with a *GrammarTypeNodeExt*.

4.4. Types and Type System

In order to perform passes such as type checking and disambiguation, we need to establish the types and type system of the Grammar language. In particular, we will create a new *GrammarType* interface (shown in Listing 10) to represent Grammar Type.

```
1 public interface GrammarType extends ClassType {
2     String getRegexString();
3 }
```

Listing 10: GrammarType.java

We can see from the definition of *GrammarType* above that we make it inherit from *ClassType* so it behaves like a *ClassType*, and inherits all the attributes and methods of *ClassType*. The only additional method we need to define that's specific to *GrammarType* is *getRegexString*(). This allows us to retrieve the (only) additional state of a *GrammarType*, namely the regex specified when declaring the *GrammarType*. We then create a concrete *GrammarType_c* class to implement *GrammarType*. Listing 11 shows the important parts of the *GrammarType_c* class.

Each *GrammarType_c* has an instance variable *regexStr*, which stores the regex specified for the *GrammarType*. It also has another instance variable *isStringLiteral*, which indicates whether this GrammarType is a *String* literal. This is because

```
public class GrammarType_c extends ClassType_c implements GrammarType {
    private final String regexStr;
    private final boolean isStringLiteral;

    public GrammarType_c(TypeSystem ts, String regexStr, boolean
    isStringLiteral) {
        super(ts);
        this.regexStr = regexStr;
        this.isStringLiteral = isStringLiteral;
    }

    @Override
    public List<? extends MethodInstance> methods() {
        return this.ts.String().methods();
    }

    @Override
    public List<? extends FieldInstance> fields() {
        return this.ts.String().fields();
    }

    @Override
    public List<? extends ReferenceType> interfaces() {
        return this.ts.String().interfaces();
    }

    @Override
    public Type superType() {
        return this.ts.String().superType();
    }

    @Override
    public String getRegexString() {
        return regexStr;
    }

    @Override
    public boolean isSubtypeImpl(Type ancestor) {
        ...to be covered in Subtyping section
    }

    @Override
    public String toString() {
        return super.toString() + "[[" + this.regexStr + "]]";
    }
}
```

Listing 11: Excerpt from GrammarType_c.java

we create *GrammarType_c* instances for both *GrammarType*s and *String* literals (to enable subtyping/type checking). Because *GrammarType* is a *ClassType*, *GrammarType_c* needs to extend *ClassType_c* and implement all the abstract methods defined in *ClassType_c*. However, we can see from Listing 11 that most methods we override we simply borrow from *String*, since we want *GrammarType* to behave just like *String*. The only methods of note are: *getRegexString*() which returns back the regex specified for the *GrammarType*, *toString*() which ensures when printing *GrammarType*s look like *String*[[*regex*]], and *isSubtypeImpl*(*Type ancestor*) which will be covered in the next section.

Using types, type system implements methods for semantic checking, such as *isSubtype*, *isCastValid*, *methodCallValid*, etc. In addition, type system also serves as the factory for creating type objects. For the Grammar language, we define a *GrammarTypeSystem* interface (shown in Listing 12) that inherits from *polyglot.types.TypeSystem*.

```
1 public interface GrammarTypeSystem extends TypeSystem {
2     GrammarType grammarOf(Position pos, String regex, boolean isStringLiteral
      );
3 }
```

Listing 12: GrammarTypeSystem.java

The only additional method we define is *grammarOf*, which constructs a *GrammarType*. We then create a concrete *GrammarTypeSystem_c* class to implement *GrammarTypeSystem* (shown in Listing 13).

Because *GrammarTypeSystem* is a *TypeSystem*, *GrammarTypeSystem_c*

```
public class GrammarTypeSystem_c extends TypeSystem_c implements
    GrammarTypeSystem {
    @Override
    public boolean isCastValid(Type fromType, Type toType) {
        ...to be covered in Type Casting section
    }

    @Override
    public grammar.types.GrammarType grammarOf(Position pos, String regex,
    boolean isStringLiteral) {=
        return new grammar.types.GrammarType(this, regex, isStringLiteral);
    }
}
```

Listing 13: Excerpt from GrammarTypeSystem_c.java

needs to extend *TypeSystem_c*. However, the only method *GrammarTypeSystem_c* overrides from *TypeSystem_c* is *isCastValid*, which will be covered in the Type Casting section. *GrammarTypeSystem_c* also overrides the abstract method *grammarOf* defined in *GrammarTypeSystem*. It takes in a position, the regex *String*, and whether if this is a *String* literal, and constructs a *GrammarType* object.

4.5. Disambiguating

With the types and type system in place, we are almost ready to perform type checking. As mentioned in the Polyglot chapter, the compilation process of Polyglot composes of several passes. And before the *TypeChecker* pass we first need to go through the *AmbiguityRemover* pass. This is to remove ambiguity of any *AmbTypeNode*, and validate if the resolved type is actually valid. For example, program 1 and 2 in Table 2 are invalid, whereas program 3 is valid.

Recall during the parsing step we built an AST, in particular we created *GrammarTypeNodeExt*s for Grammar Types. Now is a good time to show the code

#	Program
1	$Integer[[``a{*}b"]]\ i = Integer.valueOf(42);$ // compile-time error
2	$Integer[[``a{*}b"]]\ i;$ // compile-time error
3	$String[[``a{*}b"]]\ s = ``aaab";$

Table 2: Example programs to showcase ambiguity removing

of *GrammarTypeNodeExt* (in Listing 14)

```
1  public class GrammarTypeNodeExt extends GrammarExt {
2      private String regex_str;
3      public GrammarTypeNodeExt(String regex_str) {
4          super();
5          this.regex_str = regex_str;
6      }
7
8      @Override
9      public Node disambiguate(AmbiguityRemover ar) throws SemanticException {
10         AmbTypeNode n = (AmbTypeNode) this.node();
11         GrammarTypeSystem ts = (GrammarTypeSystem) ar.typeSystem();
12         NodeFactory nf = ar.nodeFactory();
13
14         TypeNode n2 = (TypeNode)n.disambiguate(ar);
15         if (!n2.isDisambiguated()) {
16             return n2;
17         }
18         Type t = n2.type();
19
20         if (!t.isCanonical()) {
21             return n2;
22         }
23
24         // t is canonical. Check that it is String.
25         if (!ts.typeEquals(ts.String(), t)) {
26             throw new SemanticException("Grammar types must be String", n.
   position());
27         }
28
29         return nf.CanonicalTypeNode(
30                         n.position(),
31                         ts.grammarOf(n.position(), regex_str, false)
32                     );
33     }
34
35 }
```

Listing 14: GrammarTypeNodeExt.java

During the *AmbiguityRemover* pass, the compiler calls the *disambiguate*

method of the *GrammarTypeNodeExt* object. We overwrite the method so that each time it is called, it first disambiguates the *TypeNode* once, and then checks if it is now disambiguated or canonical. If not, then we will leave the *TypeNode* as is for future invocations of *disambiguate* to handle. If the *TypeNode* is now indeed disambiguated and canonical, we check to make sure it is a *String*, since Grammar Types must be *String*. After this check, example programs 1 & 2 in Table 2 will expectedly fail with a *SemanticException* "*Grammar types must be String*". If the *TypeNode* is now indeed a *String*, we call the *grammarOf* method in the *GrammarTypeSystem* to create a *GrammarType* object. When constructing it, we pass in the regex and also the boolean indicating it is not a *String* literal. Finally, we wrap it inside a *CanonicalTypeNode* whose type is the *GrammarType* we just created.

4.6. Subtyping

After *AmbiguityRemover* pass is finished, the next pass is *TypeChecker* pass. As the name suggests, in this pass we perform compile-time type checking. There are 2 main parts of type checking: subtyping and type casting. This section focuses on subtyping.

On a high level, we need to check subtyping relationship when we try to assign a variable of type A to type B. This assignment is valid if and only if type A is a subtype of type B. As it pertains to Grammar Type, there are 3 scenarios of

assignment:

(i) Assigning a variable of type *String* to *GrammarType*

(ii) Assigning a variable of *GrammarType* to *String*

(iii) Assigning a variable of *GrammarType* to *GrammarType*

It's worth noting (and will be explained later in this section) that we will override the Grammar type system so that before type checking, a *String* literal will be converted from *String* type to *GrammarType* with a regex that's equal to the value of that *String* literal (ex. the *String* literal "*abc*" would have type *String*[["*abc*"]]). With that clarification out of the way, we can discuss the validity of each of the 3 scenarios. Table 3 shows some example programs, each of which demonstrates one of the 3 scenarios. The first scenario (example program 6) is clearly invalid because *String* is not a subtype of *GrammarType*. The second scenario (example program 3) is always valid because *GrammarType* is a subtype of *String*. The third scenario (example programs 1, 2, 4, 5) is valid iff the *GrammarType* on the right side of the assignment is a subtype of the *GrammarType* on the left side of the assignment.

To implement the validity check for subtyping, we override the *isSubtypeImpl*(*Type ancestor*) method on the *GrammarType_c* object during the *TypeChecker* pass. If the *GrammarType_c* object is a subtype of *ancestor*, then this is a valid assignment. Otherwise it is a compile-time error. In the Types and Type Sys-

#	Program
1	$String[[“a{*}b”]]\ s = “aaab”;$
2	$String[[“a{*}b”]]\ s = “c”$ // compile-time exception
3	$String[[“a{*}b”]]\ s = “aaab”;$ $String\ foo\ =\ s;$
4	$String[[“aa{*}b”]]\ sub = “aab”;$ $String[[“a{*}b”]]\ sup = sub;$
5	$String[[“a{*}b”]]\ sup = “ab”;$ $String[[“aa{*}b”]]\ sub = sup;$ // compile-time exception
6	$String\ foo;$ $String[[“abc”]]\ bar\ =\ foo;$ // compile-time exception

Table 3: Example programs to showcase subtyping

tem section we omitted the implementation of *isSubtypeImpl*(*Type ancestor*) of *GrammarType_c*, now is the time to look at it (in Listing 15).

```
1  @Override
2  public boolean isSubtypeImpl(Type ancestor) {
3      if (ancestor instanceof GrammarType) {
4          GrammarType gt = (GrammarType) ancestor;
5          // check that the regexp of this type is a superset of the regexp of
   gt
6          String thisRegexString = this.getRegexString();
7          if (this.isStringLiteral) {
8              thisRegexString = escape(thisRegexString);
9          }
10         final boolean isSubType = DFAOperations.contains(gt.getRegexString(),
    thisRegexString);
11         return isSubType;
12     }
13     return ts.String().isSubtype(ancestor);
14 }
```

Listing 15: Implementation of *isSubtypeImpl* in GrammarType_c.java

In *isSubtypeImpl*, we first check if the *ancestor* type is *GrammarType* (remember the current type is always *GrammarType* because this is a method of *GrammarType_c*). If the *ancestor* type is not *GrammarType*, then we say the current type (*GrammarType*) is subtype of *ancestor* type if *ancestor* type is a su-

pertype of *String* (last line in the method). The intuition for this decision is that we want *GrammarType* to be a subtype of *String* (and by definition any supertype of *String*). This covers the second scenario described above (example program 3).

If on the other hand, the *ancestor* type is also a *GrammarType*, then we fetch the regex *String* of the current type and the regex *String* of the *ancestor* type, and utilize the regex helper method (detailed in Regex Containment Problem chapter later) to check if the first regex is a sublanguage of the second regex. If so then the current type is a subtype of the *ancestor* type, otherwise it is a compile-time error. For example, *GrammarType String*[[“$aa*b$”]] is a subtype of *GrammarType String*[[“$a*b$”]] because regex $aa*b$ is a sublanguage of regex $a*b$. This covers the third scenario described above (example programs 1, 2, 4, 5).

In terms of the first scenario (example program 6) where we try to assign a *String* to a *GrammarType*, the built-in *isSubtypeImpl*(*Type ancestor*) method on the *String* type would be called. When we plug in *GrammarType* for the *ancestor* param, it would obviously return false since we never modified the *isSubtypeImpl*(*Type ancestor*) method on the *String* type to say that *String* is a subtype of *GrammarType*.

But there is one detail we have deferred to elaborate: at the beginning of the section we mentioned that in order to check if a *String* literal is a subtype of a *GrammarType*, we need to convert the type of the *String* literal to a *GrammarType*. Recall in the AST section, we mentioned that during the parsing stage, we attach

a *StringLitExt* to each *String Node*. Now, during the *TypeChecker* pass, we can override the *typeCheckOverride* method of the *StringLitExt* to convert it into a *GrammarType* (shown in Listing 16).

```
1      @Override
2      public Node typeCheckOverride(Node parent, TypeChecker tc) throws
       SemanticException {
3          if (this.node() instanceof StringLit_c) {
4              StringLit_c n = (StringLit_c)this.node();
5              GrammarTypeSystem ts = (GrammarTypeSystem) tc.typeSystem();
6              return n.type(ts.grammarOf(n.position(), n.value(), true));
7          }
8          return this.superLang().typeCheckOverride(this.node(), parent, tc);
9      }
10 }
```

Listing 16: Implementation of *typeCheckOverride* in StringLitExt.java

The essence is that in *typeCheckOverride* we call the *grammarOf* method of the *GrammarTypeSystem* to override the type of the *String* to be a *GrammarType*. When calling, we pass in the value of the *String* literal, and an indication that this *GrammarType* is actually a wrapper around a *String* literal. As an example, the *String* "aaab" would be converted to a *GrammarType* *String*[["*aaab*"]] after *typeCheckOverride*. With this conversion, we are now able to check if a *String* literal is a subtype of a *GrammarType* by reusing the *isSubtypeImpl* method, just as if we were checking the subtyping relationship between two *GrammarType*s.

4.7. Type Casting

Another important aspect of type checking is type casting. Because we support type casting in the Grammar language, we want to perform both compile-time and

run-time checks for type casting. Similar to subtyping, there are three scenarios:

(i) Casting a *String* to a *GrammarType*

(ii) Casting a *GrammarType* to a *String*

(iii) Casting a *GrammarType* to a *GrammarType*

And like before, Table 4 shows some example programs, each of which demonstrates one of the three scenarios above. The second scenario (example program 7 in table 4) is the most trivial one: In this case, we are casting a subtype (*GrammarType*) to a supertype (*String*). Therefore the cast is actually a no-op and allowed. In the first scenario (example programs 5 & 6), since *GrammarType* is a subtype of *String*, and since we don't know for sure the value inside the *String* at compile time, we have to defer the task of determining the validity of the cast to runtime, where we check if the actual value inside the *String* variable matches the regex of the *GrammarType*. The third scenario (example programs 1, 2, 3, 4) is actually either analogous to the first scenario or to the second scenario: if the $fromType$ *GrammarType* is a subtype of the *toType GrammarType*, then this cast is a no-op and allowed similar to the second scenario; if the $fromType$ *GrammarType* is a supertype of the *toType GrammarType*, then we need to perform runtime cast check similar to the first scenario.

To implement the validity check for type casting, we override the $isCastValid($ *Type* $fromType$, *Type toType*) method on the *GrammarTypeSystem*, which is

#	Program
1	$String[[``a{*}b"]]\ foo = ``aab"$; $String[[``aa{*}b"]]\ bar\ =\ (String[[``aa{*}b"]])\ foo$;
2	$String[[``a{*}b"]]\ foo = ``aab"$; $String[[``c \vert d"]]\ bar\ =\ (String[[``c \vert d"]])\ foo$; // compile-time exception
3	$String[[``a{*}b"]]\ foo = ``ab"$; $String[[``aaa{*}b"]]\ bar\ =\ (String[[``aaa{*}b"]])\ foo$; // runtime exception
4	$String[[``aa{*}b"]]\ foo = ``aab"$; $String[[``a{*}b"]]\ bar\ =\ (String[[``a{*}b"]])\ foo$;
5	$String\ foo = ``aab"$; $String[[``a{*}b"]]\ bar\ =\ (String[[``a{*}b"]])\ foo$;
6	$String\ foo = ``xyz"$; $String[[``a{*}b"]]\ bar\ =\ (String[[``a{*}b"]])\ foo$; // runtime exception
7	$String[[``a{*}b"]]\ sub = ``aab"$; $String\ sup\ =\ (String)\ sub$;

Table 4: Example programs to showcase type casting

called during the $TypeChecker$ pass. The implementation of $isCastValid$ was omitted in the Types and Type System section. It is now shown in Listing 17.

```
1  @Override
2  public boolean isCastValid(Type fromType, Type toType) {
3      if (this.isSubtype(fromType, this.String())
4              && toType instanceof GrammarType
5              && (this.isSubtype(toType, fromType) || this.isSubtype(fromType,
       toType))) {
6          return true;
7      }
8
9      return super.isCastValid(fromType, toType);
10 }
```

Listing 17: Implementation of $isCastValid$ in GrammarTypeSystem_c.java

The first scenario where we cast a subtype to a supertype is allowed by the last line in the $isCastValid$ method implementation. The gist of the remaining function body is that it simply checks if the $fromType$ is a subtype or supertype of the $toType$. If it is neither, that means the $fromType$ and the $toType$ simply aren't compatible,

and would return false. Otherwise, either the $fromType$ is a subtype of the $toType$ in which case it is trivially true, or the $fromType$ is a supertype of the $toType$ in which case we cannot determine the validity of the cast at compile time. In either case, $isCastValid$ would return true.

We now focus on the cases where we couldn't determine the cast validity during compile-time. In these cases, we run a custom library method at runtime to determine the validity of the cast (i.e. whether the actual *String* value of the $fromType$ at runtime matches the regex defined in the *toType*). To invoke the runtime library method, we would rewrite the output Java code during compile time. Specifically, if the original code was like in Listing 18, then the output Java code would become like in Listing 19.

```
1 String foo = "xyz";
2 String[["a*b"]] bar = (String[["a*b"]]) foo;
```

Listing 18: Original code in Grammar language

```
1 String foo = "xyz";
2 String bar = grammar.runtime.RuntimeCastChecker.check("a*b", foo);
```

Listing 19: Output code in Java language

This way, we would still catch the invalid cast during runtime. To implement the runtime check, we first create a runtime library *grammar.runtime*, and in the runtime library we create a class $RuntimeCastChecker$ (shown in Listing 20). $RuntimeCastChecker$ only has one static method *check* which takes in the regex specified of the *toType*, and the actual *String* value of the $fromType$ at runtime. Then, *check* simply utilizes Java's regex library to determine if the *String* matches

the regex. If so, this is a valid runtime cast; if not, the program would get a *ClassCastException* during execution. Now that we have this util class to check for validity of runtime cast, we need to actually find a way to invoke it during runtime.

```
public class RuntimeCastChecker {
    public static String check(final String regex, final String var) {
        if (Pattern.matches(regex, var)) {
            return var;
        } else {
            throw new ClassCastException("Invalid Runtime Cast");
        }
    }
}
```

Listing 20: RuntimeCastChecker.java

So the task now is to rewrite a cast such as ($String$[[“$a*b$”]]) foo to $grammar.runtime.RuntimeCastChecker.check$(“$a*b$”, foo), if and only if foo is a supertype of $String$[[“$a*b$”]]) (if not then there is no runtime cast check needed). This kind of rewrite happens in the *CodeGeneration* step of Polyglot. Recall in the AST section, we mentioned that during parsing we create a *GrammarCastExt_c* object for each cast operation. What we need to do is to override the *translate* method of *GrammarCastExt_c* to perform the rewrite, as shown in Listing 21.

In *translate*, we check if the type being casted to (*ltype*) is a *GrammarType*, and also if *ltype* is a subtype of *rtype* (the type being casted). If both conditions are true, then that's the condition we are looking for to perform the rewrite. When rewriting, we first write down the qualified name of the *RuntimeCastChecker* class and “invoke” its *check* method. Then for the params for *check* we first write the

```
1  public class GrammarCastExt_c extends GrammarExt {
2      @Override
3      public void translate(CodeWriter w, Translator tr)  {
4          GrammarTypeSystem ts = (GrammarTypeSystem) tr.typeSystem();
5          Cast c = (Cast) node();
6          Type rtype = c.expr().type();
7          Type ltype = c.castType().type();
8          if (ts.isSubtype(ltype, rtype) && ltype instanceof GrammarType) {
9              // rewrite "(String[["a*b"]]) foo" to "grammar.runtime.
    RuntimeCastChecker.check("a*b", foo)"
10             w.write("grammar.runtime.RuntimeCastChecker.check(\"");
11             w.write(((GrammarType) ltype).getRegexString());
12             w.write("\", ");
13             c.expr().translate(w, tr);
14             w.write(")");
15         } else {
16             superLang().translate(node(), w, tr);
17         }
18     }
19 }
```

Listing 21: Implementation of *translate* in GrammarCastExt_c.java

regex *String* of the *ltype*, then a comma, then the translation of *rtype*, and finally a right parenthesis. With the runtime cast checker in place, example programs 3 and 6 in Table 4 would pass compile-time check and generate Java code with the runtime cast check, and would get "Invalid Runtime Cast" exception during runtime.

Chapter V.

Regex Containment Problem

At the core of the type system for the Grammar Type language extension is the regex containment problem. Specifically, in order to know whether a Grammar Type $String[[regex1]]$ is a subtype of $String[[regex2]]$, we need to be able to determine if a regular expression $regex1$ is a sublanguage of another regular expression $regex2$ (in other words, whether $regex2$ "contains" $regex1$). We define $regex1$ to be a sublanguage of $regex2$ if the language (all the *String*s) represented by $regex1$ is a subset of the language represented by $regex2$. As an example, the regex $aa*b$ is a sublanguage of the regex $a*b$ because $aa*b$ represents all the *String*s with at least one a followed by a b, and $a*b$ represents everything $aa*b$ represents, plus an additional *String* "b".

There is no built-in Java library for answering this regex containment question, so we implement our own custom logic to tackle this problem. In order to determine whether $regex1$ is a sublanguage of $regex2$, there are a couple of steps involved: (i) we will convert each regex into a syntax tree. (ii) we will convert each syntax tree to an NFA (non-deterministic finite automata). (iii) we will convert each NFA to a

DFA (deterministic finite automata). (iv) solve the DFA containment problem.

5.1. Regex to Syntax Tree

In our design, a *SyntaxTree* for regex is a binary tree and is composed of *SyntaxTreeNode*s. Each *SyntaxTreeNode* is either an *Operator* or an *Operand* (shown in Listing 22). *Operator* is an enum that represents the operators in regular expressions ($*$, $|$, or concatenation). An *Operand* is either a normal *char*, or the wildcard character(.) that matches any *char*.

```
1  public class SyntaxTreeNode {
2      // if this SyntaxTreeNode is an operator
3      private final Operator operator;
4      // if this SyntaxTreeNode is an operand (i.e. a char)
5      private final Operand operand;
6      private final SyntaxTreeNode leftChild;
7      private final SyntaxTreeNode rightChild;
8  }
9
10 enum Operator {
11     CONCAT,
12     OR, // lowest precedence
13     KLEENE // highest precedence
14 }
15
16 public static class Operand {
17     private char c;
18     private
19 }
```

Listing 22: Excerpt from SyntaxTree.java

Since a regex *SyntaxTree* is a binary tree, each *SyntaxTreeNode* has a left child *SyntaxTreeNode* and (optionally) a right child *SyntaxTreeNode*. Specifically, the right child will be present if the *Operator* is *OR* or *CONCAT*, and missing if the *Operator* is *KLEENE*. To string all these together, let's take the regex $a*b|c$

as an example. This regex has 3 operations. The 1st operation is a Kleene star on the character a. This is represented by a *SyntaxTreeNode* with a *KLEENE Operator*, and a left child *SyntaxTreeNode* with a *char a Operand* (Figure 6 a). The 2nd operation is a concatenation of the previous result ($a*$) with the character b. This is represented by a *SyntaxTreeNode* with a *CONCAT Operator*, a left child *SyntaxTreeNode* that was the result from the 1st operation, and a right child *SyntaxTreeNode* with a *char b Operand* (Figure 6 b). The 3rd and final operation is an alternation between the previous result ($a*b$) and the character c. This is represented by a *SyntaxTreeNode* with an *OR Operator*, a left child *SyntaxTreeNode* that was the result from the 2nd operation, and a right child *SyntaxTreeNode* with a *char c Operand* (Figure 6 c).

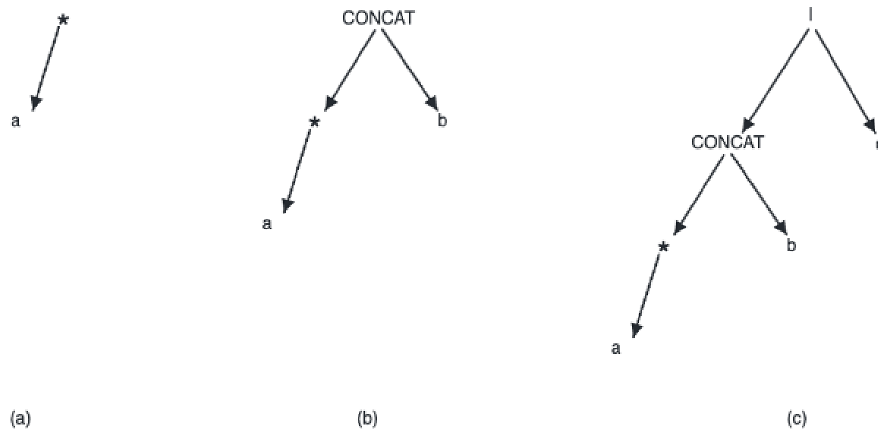

Figure 6: Syntax tree for regex $a*b|c$

(a) shows the syntax tree for $a*$. (b) shows the syntax tree for $a * b$. (c) shows the syntax tree for $a*b|c$

Now that we have an understanding of the structure of the regex syntax tree,

we will talk about how to construct them from regexes. The first step is a preprocessing pass. In this pass we sanity check the regex for syntactic errors and weed out invalid regexes. We also clean up the regex by removing useless parenthesis(ex. (), $((a))$, etc.). In addition, we undo any syntax sugar in the regex(ex. converting $a-z$ back to $a|b|c\ldots|y|z$). After the preprocessing pass, we start the actual building of the syntax tree using a recursive function that takes in a stream of elements, where each element is either a *char* or a *SyntaxTreeNode*. The method runs 3 passes, corresponding to the 3 types of *Operator*s. The order of the passes is determined by the precedence of the different *Operator*s: *KLEENE*, then *CONCAT*, then *OR*. Initially when *buildSyntaxTree* is called for the first time, all the elements in the stream are *char*s. And as the passes progress, more and more elements become *SyntaxTreeNode*s until there is one root *SyntaxTreeNode* left, and that is our result *SyntaxTree*.

5.2. Syntax Tree to NFA

To help understand our goal here, let's look back at our previous example regex $a{*}b|c$. Now we know what the syntax tree would look like, we look at what the NFA will look like.

As Listing 23 shows, both NFA and DFA are graphs with nodes and edges, and extend the abstract class *FA*. An *FA* has one start *Node*, and a list of accept *Node*s. Each *Node* stores all of its neighbors in a HashMap where the key is the edge from this

```java
public abstract class FA {
    Node startNode;         // the start node of this FA
    List<Node> acceptNodes; // a list of accept nodes of this FA
    List<Node> allNodes;    // list of all nodes in this FA
}

public abstract class Node {
    protected LinkedHashMap<Edge, Node> nextNodes;
    boolean isStartNode;
    boolean isAcceptNode;
}

public abstract class Edge {
}


public class NFAEdge extends Edge {
    private SyntaxTree.Operand transitionChar;
    private boolean isEpsilonTransition;
}
```

Listing 23: Definitions of FA, $Node$, $Edge$, and $NFAEdge$

node to the neighbor, and the value is the neighboring node. Each node also denotes whether it is a start node and accept node. An $NFAEdge$ represents the transition from one $NFANode$ to the next. The transition is either a normal transition (a *char* literal or the wildcard(.)), or an epsilon transition (an epsilon transition is an immediate jump from one $NFANode$ to the next). To make things more concrete, the right side of Figure 7 is a demonstration of the NFA for the example regex $a{*}b|c$.

To construct an NFA from the syntax tree, we start from the root $SyntaxTreeNode$. There are 4 main cases: (i) if the $SyntaxTreeNode$ is an *Operand* instead of an *Operator* (Figure 8 a), we know it's a *char*. In this case, we create a start $NFANode$ with an $NFAEdge$ pointing to an accept $NFANode$. The edge will have its $transitionChar$ set to be the *char* in the *Operand*. (ii) if the $SyntaxTreeNode$ is an *OR Operator* (Figure 8 b), we create a start $NFANode$, and recursively build NFAs for the left

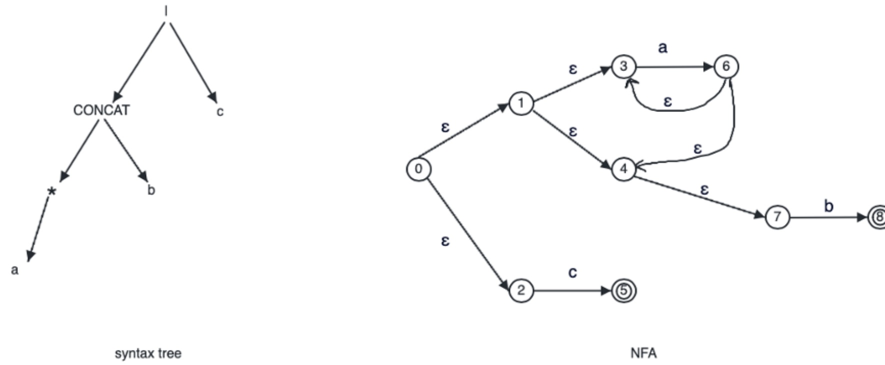

Figure 7: NFA for regex $a{*}b|c$

Left side shows the syntax tree for regex $a{*}b|c$. Right side shows the NFA. Each circle represents an $NFANode$. Double circle represents an accept node. 0 is always the start node. Each edge represents an $NFAEdge$.

and right children of the *OR Operator*. The start node will have an epsilon transition to the 2 original start nodes in the sub NFAs. And the original accept nodes in the 2 sub NFAs will continue to be accept nodes. (iii) if the $SyntaxTreeNode$ is a *CONCAT Operator* (Figure 8 c), we first build NFAs for the left and right children of the *CONCAT Operator*. The original start node in the left NFA will continue to be the start node of the result NFA, and the original accept nodes in the right NFA will be accept nodes of the result NFA. An epsilon transition is added from each of the original accept nodes in the left NFA to the original start node in the right NFA. (iv) if the $SyntaxTreeNode$ is a *KLEENE Operator* (Figure 8 d), we first build an NFA for the left (and only) child of the *KLEENE Operator*. We also create a new start node and a new accept node. The new start node will epsilon transition to the new accept node as well as the original start node in the sub NFA. The original accept nodes in the sub NFA will have epsilon transitions to the original start node

as well as the new accept node.

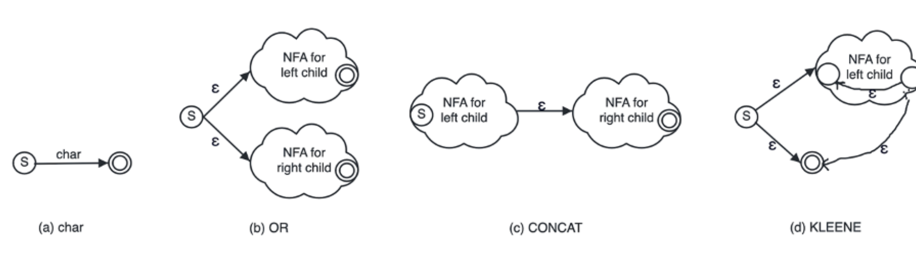

Figure 8: Fundamental NFA structures for different $SyntaxTreeNode$ types

(a) shows the NFA structure for $char$. (b) shows the NFA structure for OR. (c) shows the NFA structure for $CONCAT$. (d) shows the NFA structure for $KLEENE$.

5.3. NFA to DFA

Due to the presence of epsilon transitions, NFAs are nondeterministic. Therefore we convert NFA to DFA for a more structured and deterministic representation, enabling more efficient containment checking. The idea is to group all of the $NFANode$s reachable by epsilon transition into one $DFANode$ to eliminate nondeterminism. Each $DFANode$ contains a list of $NFANode$s that are reachable from each other via epsilon transitions (shown in Listing 24).

```
1 public class DFANode extends Node {
2     List<NFANode> nfaNodes;
3     boolean isD0;
4 }
```

Listing 24: Structure of $DFANode$

For easier understanding, right side of Figure 9 below shows the result DFA corresponding to our example NFA.

The process for converting NFA to DFA is quite straightforward. We start

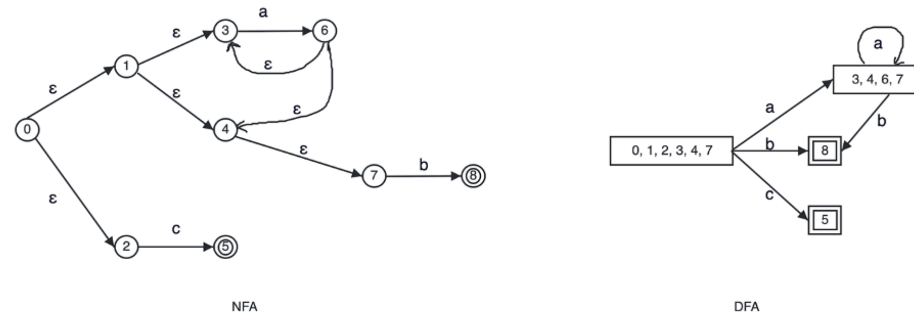

Figure 9: DFA for regex $a{*}b|c$

Left side shows the NFA for regex $a{*}b|c$. Right side shows the DFA. Each rectangle represents a $DFANode$. Double rectangle represents an accept node. Leftmost rectangle is always the start node. Each edge represents a $DFAEdge$.

traversing the NFA from the root node and get the epsilon closure of the root node. The epsilon closure contains every node reachable from the root node via epsilon transition. In the example case above, it contains $NFANode$ 0, 1, 2, 3, 4, 7. So we create the start $DFANode$ of our DFA and it contains the list of $NFANode$s in the epsilon closure. Then, we maintain a queue of $DFANode$s we want to explore, initialized with the start $DFANode$. While the queue is not empty, we keep popping $DFANode$s from the queue. For each $DFANode$ popped, we iterate over each of the $NFANode$ it contains. For each of that $NFANode$'s neighbors, if the $NFAEdge$ is not an epsilon transition (meaning it's a normal $char$), then we create a new potential $DFANode$ with the epsilon closure of that $NFANode$'s neighbor. In the example case above, when we were inspecting $DFANode$ 0, 1, 2, 3, 4, 7 from the queue, because $NFANode$ 3 has an $NFAEdge$ a to $NFANode$ 6, we get the epsilon closure of $NFANode$ 6 which contains $NFANode$ 3, 4, 6, 7, and then we create a new $DFANode$ containing these $NFANode$s. Finally we create a $DFAEdge$ a pointing

from the start $DFANode$ to this new $DFANode$. Another way to understand this is: at start, we could be in any of state 0, 1, 2, 3, 4, 7. And after taking in a $char$ a, we could be in any of the state 3, 4, 6, 7. During the process, if any $NFANode$ inside a $DFANode$ is an accept node, that makes that $DFANode$ also an accept node. In the example in Figure 9, because the $NFANode$ 8 is an accept node, the $DFANode$ 8 is also an accept node. We repeat this process until the queue of $DFANode$s to process is empty, and that's when we successfully built our DFA.

5.4. DFA Containment Problem

Now that we are able to convert regexes into DFAs, the last step in determining whether $regex2$ is a sublanguage of $regex1$ is to check if $DFA1$ contains $DFA2$ (i.e. whether all $String$s accepted by $DFA2$ are also accepted by $DFA1$). This is a 4 step process: (i) make $DFA1$ and $DFA2$ total. (ii) compute the complement of $DFA1$, call if $DFA3$. (iii) compute the intersection of $DFA2$ and $DFA3$. (iv) if the intersection accepts no $String$s, then $DFA1$ contains $DFA2$.

A DFA is total if for every $DFANode$, there is a transition defined for every possible input $char$. We will use the word $ALPHABET$ as a syntax sugar to represent the set of all possible input $char$s, and use $ALPHABET - \{char1, \ldots, charN\}$ to represent the set of all $char$s in $ALPHABET$ except $char1$, ..., and $charN$. To illustrate, Figure 10 shows the transition from a non-total DFA we computed in the previous example to the corresponding total DFA.

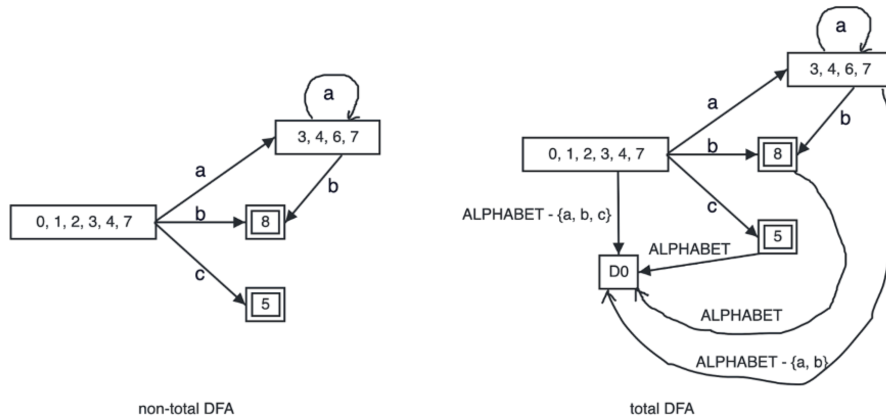

Figure 10: Total DFA for regex $a{*}b|c$

Left side is the original non-total DFA. Right side is the total DFA

To make a DFA total, we first introduce a new dummy $DFANode$ $D0$. We then iterate over each $DFANode$ in the original DFA. For each $DFANode$, we compute the list of $char$s it doesn't have a transition for, and create a $DFAEdge$ containing those $char$s from that $DFANode$ to $D0$. For example, because the non-total $DFANode$ 0, 1, 2, 3, 4, 7 have transitions for a, b, and c, we created a $DFAEdge$ containing $ALPHABET - \{a, b, c\}$ from $DFANode$ 0, 1, 2, 3, 4, 7 to $D0$.

The 2nd step is computing the complement of $DFA1$. This is quite straightforward as we simply need to convert all the accept nodes in $DFA1$ to be non-accept nodes, and all the non-accept nodes to be accept nodes. This matches the intuition for the complement of a DFA: any $String$ that was accepted before will not be accepted now, and any $String$ that was not accepted before will be accepted now. Figure 11 illustrates the transition of our example total DFA to its complement.

The 3rd step is to compute the intersection between $DFA2$ and the comple-

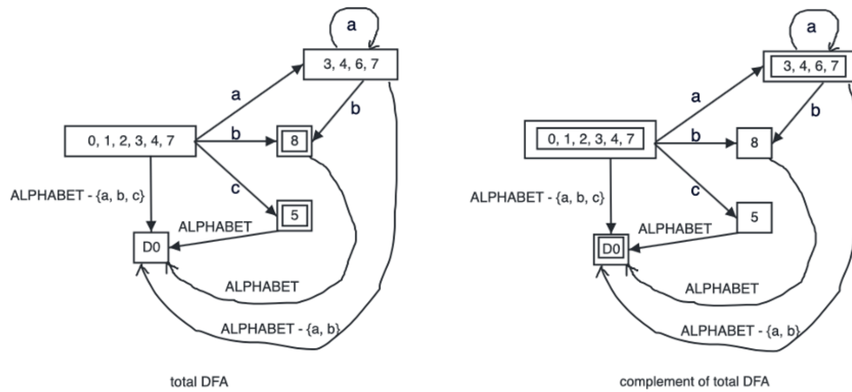

Figure 11: Complement of total DFA for regex $a*b|c$

Left side is the original total DFA. Right side is the complement of the total DFA

ment of $DFA1$ (call it $DFA3$ for convenience). Intuitively, the intersection between $DFA2$ and $DFA3$ represent the set of $String$s accepted by both DFAs. For illustration purpose, let's assume our $regex1$ is still $a*b|c$, and $regex2$ is aab (this is the case if we are doing type checking on a statement like $String[[\text{“}a*b|c\text{”}]]\ foo = \text{“}aab\text{”}$). We already have $DFA3$ (the complement of the total DFA for $a*b|c$), so we just need $DFA2$ (the DFA of aab) before we calculate the intersection between $DFA3$ and $DFA2$. Right side of Figure 12 shows the derivation of $DFA2$.

To compute the intersection of two DFAs ($DFA3$ and $DFA2$ in our example), we create a new intersection DFA composed of $IntersectionDFANode$s (shown in Listing 25). Every pair of nodes (one in each DFA) creates one $IntersectionDFANode$ (in our case $dfaNode1$ would represent one $DFANode$ in $DFA3$, and $dfaNode2$ would represent one $DFANode$ in $DFA2$). As a shorthand, we will write $IntersectionDFANode$ $(a_1, \ldots, a_n \backsim b_1, \ldots, b_m)$ to correspond to $DFANode$ $a_1, \ldots, a_n$ in $DFA3$ and $DFANode$

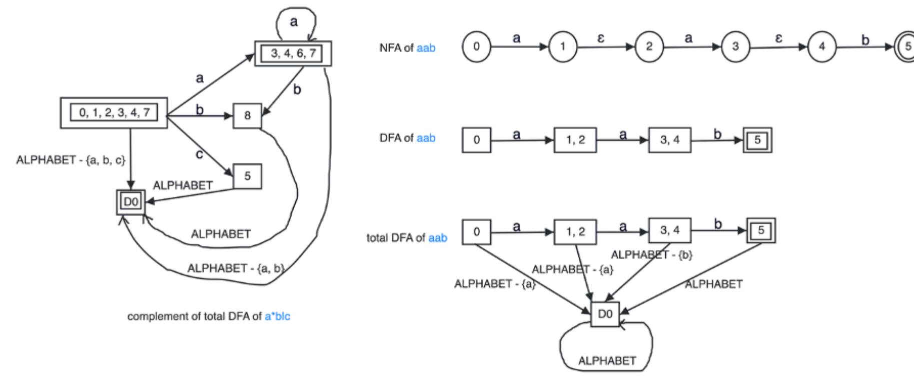

Figure 12: $DFA3$ and $DFA2$

Left side is a recap of the complement of total DFA of $a*b|c$. Right side is the derivation of DFA of aab

$b_1, \ldots, b_m$ in $DFA2$.

```
public class IntersectionDFANode extends Node {
    DFANode dfaNode1;
    DFANode dfaNode2;
}
```

Listing 25: Structure of $IntersectionDFANode$

So in our example, since $DFA3$ has 5 nodes and $DFA2$ also has 5 nodes, the result $IntersectionDFA$ has 25 $IntersectionDFANode$s. For each $IntersectionDFANode$, we extract all groups of common transition $char$s between $dfaNode1$ and $dfaNode2$, and create a $DFAEdge$ from the current $IntersectionDFANode$ to the $IntersectionDFANode$ whose $dfaNode1$ is pointed to by the common transition $char$s in $DFA3$, and whose $dfaNode2$ is pointed to by the common transition $char$s in $DFA2$. In our example, for the $IntersectionDFANode$ $(0, 1, 2, 3, 4, 7 \backsim 0)$, the groups of common transition $char$s are: $\{a\}$, $\{b\}$, $\{c\}$, and $ALPHABET - \{a, b, c\}$. This is because: in $DFA3$, $DFANode$ 0, 1, 2, 3, 4, 7 points to $DFANode$ 3, 4, 6, 7 if the transition $char$ is a, points to $DFANode$ 8 if the transition $char$ is b, points to $DFANode$ 5 if

the transition $char$ is c, and points to $DFANode$ $D0$ if the transition $char$ is any of $ALPHABET$–$\{a, b, c\}$. In $DFA2$, $DFANode$ 0 points to $DFANode$ 1, 2 if the transition $char$ is a, points to $DFANode$ $D0$ if the transition $char$ is b or c, and also points to $DFANode$ $D0$ if the transition $char$ is any of $ALPHABET$–$\{a, b, c\}$. After determining the groups of common transition $char$s and the $DFANode$s they point to in the corresponding DFAs, we add corresponding $DFAEdge$s for the $IntersectionDFA$ (in our example, for $IntersectionDFANode$ $(0, 1, 2, 3, 4, 7 \backsim 0)$, we add a $DFAEdge$ a to $IntersectionDFANode$ $(3, 4, 6, 7 \backsim 1, 2)$, a $DFAEdge$ b to $IntersectionDFANode$ $(8 \backsim D0)$, a $DFAEdge$ c to $IntersectionDFANode$ $(5 \backsim D0)$, and a $DFAEdge$ $ALPHABET$–$\{a, b, c\}$ to $IntersectionDFANode$ $(D0 \backsim D0)$. We repeat this process to add all $IntersectionDFANode$s and $DFAEdge$s to complete the $IntersectionDFA$.

The 4th and last step is to check if the intersection of $DFA3$ (the complement of $DFA1$) and $DFA2$ accepts no $String$s. If so then $DFA1$ contains $DFA2$, which means $regex1$ contains $regex2$, which means $regex2$ is a sublanguage of $regex1$. Figure 13 visualizes the case where $DFA3$ contains $DFA2$, and the case where $DFA3$ does not contain $DFA2$.

The intuition is this: when $DFA1$ contains $DFA2$, the set of $String$s accepted by $DFA1$ is a superset of the set of $String$s accepted by $DFA2$. Therefore, $DFA3$, the complement of $DFA1$ (the shaded area on the left side of Figure 13) has no overlap with $DFA2$. However, when $DFA1$ does not contain $DFA2$, the set of $String$s accepted by $DFA1$ does not fully contain the set of $String$s accepted by

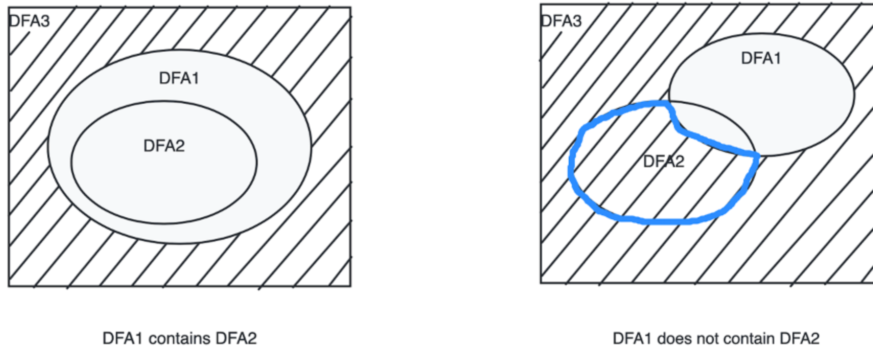

Figure 13: Intersection of $DFA3$ and $DFA2$

Left side depicts the intersection when $DFA1$ contains $DFA2$. Right side depicts the intersection when $DFA1$ does not contain $DFA2$

$DFA2$. As a result, the complement of $DFA1$ (the shaded area on the right side of Figure 13) overlaps with $DFA2$. Namely, the area circled in blue. To check if an FA (in this particular case a DFA) accepts no $String$, we iterate over each of the accept nodes of the FA. If none of them is reachable, then the FA accepts no $String$. A node is reachable if there exists a path from the start node of the FA to that node.

Chapter VI.

Example Use Cases

We can put our Grammar Type for String to use in many different use cases – all we need is to define a regex to convey the type of grammar we want to enforce, and declare variables to be of that Grammar Type. In order to enable more concise regex writing, we implemented several enhancements/syntax sugars on top of the standard characters and operators we support. Just to recap, in the base form of our regex parser we supported any lower case character from a to z, any upper case character from A to Z, any digit from 0 to 9, the left and right parentheses (and), the Kleene star character $*$, and the Or character |. This works but is very cumbersome if we want to say write a regex that allows any character. In that case our regex would be $a|b|\ldots|A|B\ldots|Z|0|1|\ldots 9$, a whopping 123 characters long. To alleviate such inconvenience, we added the following enhancements/syntax sugars: (i) we used $a{-}z$ as a shorthand for $a|b|\ldots|z$, $A-Z$ as a shorthand for $A|B|\ldots|Z$, and $0-9$ as a shorthand for $0|1|\ldots|9$. (ii) we added support for the dot (.) character to mean any of the supported characters. However, just like other meta characters (| or $*$), if we wanted it to represent its literal form, we need to be able to escape it. (iii) that is why

we also added support for escaping meta characters using the escape character (\). So just a single dot (.) in the regex would mean any character, but an escaped dot (\.) would mean the literal dot (.) character. (iv) finally, we added support for specifying the minimum and maximum number of characters using the syntax $\{min, max\}$. So the regex $(foo)\{1,3\}$ would mean 1 to 3 foos.

6.1. Valid Email Addresses

One example use case of Grammar Type is for validating email addresses. On a fundamental level, we will consider an email to be valid if it is of the form: [between 1 and 64 lowercase or uppercase characters or digits], followed by the at sign (@), followed by [between 1 and 255 lowercase or uppercase characters or digits], followed by dot (.), followed by [between 2 and 63 lowercase or uppercase characters]. Based on this, we can define the regex for valid email addresses to be:

$$(a-z|A-Z|0-9)\{1{,}64\}@(a-z|A-Z|0-9)\{1{,}255\}\backslash.(a-z|A-Z)\{2{,}63\}$$

With this regex defined, we can test out some example programs where we need valid email addresses (shown in Table 5). For brevity, we will use ER (for email regex) as shorthand for the above regex.

When we compiled and then executed the test programs in Table 5, programs 1, 3, and 5 succeeded. Program 2 failed during compile time for "type of the variable initializer $String[[abc@invalid]]$ does not match that of the declaration $String[[ER]]$".

#	Program
1	$String[[ER]]\ email = $ "*abc123@gmail.com*";
2	$String[[ER]]\ email = $ "*abc123@invalid*"; // compile-time error
3	$String[[ER]]\ email;$ $email = $ "*abc123@gmail.com*";
4	$String[[ER]]\ email;$ $email = $ "*abc123@invalid*"; // compile-time error
5	$String\ s = $ "*abc123@gmail.com*"; $String[[ER]]\ email = (String[[ER]])\ s;$
6	$String\ s = $ "*abc123@invalid*"; $String[[ER]]\ email = (String[[ER]])\ s;$ // runtime error

Table 5: Test programs to validate email addresses

Program 4 failed during compile time for "cannot assign *String abc@invalid* to $String[[ER]]$". Program 6 failed during runtime for "Invalid Runtime Cast". All of the results were as expected, which proved that the Grammar Type for String works for validating email addresses.

6.2. Valid User Inputs

Recall form the Introduction chapter, we looked at an example program that was prone to SQL injection attacks. The part where it reads user input and constructs the SQL query is shown in Listing 26.

```
1 System.out.print("Enter username: ");
2 final String username = scanner.nextLine();
3 System.out.print("Enter password: ");
4 final String password = scanner.nextLine();
5 final String sql = "SELECT * FROM users WHERE username = \"" + username + "\"
      AND password = \"" + password + "\"";
```

Listing 26: Original Java code with no input validation

Because the program simply stores username and password as *String*s, an

attacker can enter inputs such as *anything*" *OR* "1" = "1 for username field, and *anything*" *OR* "1" = "1 for password field. The constructed SQL query would be *SELECT* ∗ *FROM users WHERE username* = "*anything*" *OR* "1" = "1" *AND password* = "*anything*" *OR* "1" = "1". Due to the *OR* "1" = "1" conditions, the query effectively becomes *SELECT* ∗ *FROM users WHERE true AND true*. With our Grammar Type for String implementation, we are ready to tackle this problem. Instead of storing username and password as *String*s, we store them as Grammar Types, where we define the appropriate regex for a valid username or valid password. Let's define the requirement for a username or password to be 3 to 64 lowercase or uppercase characters or digits. Then we can come up with the regex grammar for a valid username or password:

$$(a-z|A-Z|0-9)\{3,64\}$$

So we rewrote our program (shown in Listing 27) utilizing the regex. When we ran our program and entered *anything*" *OR* "1" = "1 as input for username or password, the program failed immediately at the same line for "Invalid Runtime Cast" errors, and it never tried to execute (or even construct) the SQL query. This is because the white space, the quotes, and the equal sign in the user input are all illegal according to the regex defined for the Grammar Type. With this Grammar Type for valid inputs, we can now be confident that we are protected from this type of SQL injection attacks.

```
System.out.print("Enter username: ");
final String[["(a-z|A-Z|0-9)*]] username = scanner.nextLine();
System.out.print("Enter password: ");
final String[["(a-z|A-Z|0-9)*]] password = scanner.nextLine();
final String sql = "SELECT * FROM users WHERE username = \"" + username + "\"
    AND password = \"" + password + "\"";
```

Listing 27: Same code rewritten in the Grammar language

6.3. Valid URLs

Another example use case of Grammar Type comes from work. In this case, when a web client A sends an HTTP request to service B, the query param contains a redirect URL indicating after service B processes the request, where service B should redirect the browser to. However, the redirect URL can only be a subset of all valid URLs. Specifically, for each original web client *subdomain.snapchat.com*, the redirect URL has the following restrictions: (i) the domain still has to be *snapchat.com*, (ii) the subdomain can either remain the same as the original URL, or have *–prod*, *–alpha*, *–gold* suffix (indicating whether it's a production or internal build), (iii) the URL can optionally end with a question mark (?) or forward slash (/) followed by more characters. For example, if web client A was *web.snapchat.com*, then the regex for valid redirect URL would be:

$$https://web(-(prod|alpha|gold))?\backslash.snapchat\backslash.com([/?].*)?$$

With this regex defined, we can test out some example programs where we validate redirect URLs (shown in Table 6). For brevity, we will use RR (for redirect

regex) as shorthand for the above regex.

#	Program
1	$String[[RR]]\ url$ = "$web{-}prod.snapchat.com/anything$";
2	$String[[RR]]\ url$ = "$web3.snapchat.com$"; // compile-time error
3	$String[[RR]]\ url$; url = "$web{-}gold.snapchat.com?anything$";
4	$String[[RR]]\ url$; url = "$web.snap.com$"; // compile-time error
5	$String\ s$ = "$web.snapchat.com$"; $String[[RR]]\ url = (String[[RR]])\ s$;
6	$String\ s$ = "$web3.snapchat.com$"; $String[[RR]]\ url = (String[[RR]])\ s$; // runtime error

Table 6: Test programs to validate redirect URLs

When we compiled and then executed the test programs in Table 6, programs 1, 3, and 5 succeeded. Program 2 failed during compile time for "type of the variable initializer $String[[web3.snapchat.com]]$ does not match that of the declaration $String[[RR]]$. Program 4 failed during compile time for "cannot assign $String$ $web.snap.com$ to $String[[RR]]$". Program 6 failed during runtime for "Invalid Runtime Cast". All of the results were as expected, which showed that the Grammar Type for $String$ can be useful for validating URLs.

Chapter VII.

Conclusion

In conclusion, we designed and implemented a solution (Grammar Type for String) that enhanced the Java language by extending the Polyglot compiler frontend. We introduced a new Grammar Type with the syntax $String[[regex]]$. Grammar Types are subtypes of *String*s so they inherit all behaviors of *String*s. But at the same time, with Grammar Types, we are able to use regexes to enforce any "grammar" we want for a *String*. We ensure this by implementing both compile time type checking for subtyping and casting, and runtime type checking for casting. We also dived deep into the regex containment problem in order to solve the Grammar Type subtyping problem. As demonstrated in the Example Use Cases chapter, we proved Grammar Types can be extremely useful in real life applications. Some of the reasons include: (i) we no longer need to write validation code for different *String* inputs. (ii) we won't need to rely on application developers to always remember to validate the *String* inputs. (iii) we won't need an extra class to represent each *String* input type. (iv) in many cases we can catch the problem during compile-time, instead of at runtime.

It's worth mentioning that in certain scenarios, we might want to be able to use more precise/powerful "grammars". However, due to the limitation of regular expressions, a lot of these are not possible. For example, if we wanted to specify a *String* with equal number of 0's followed by equal number of 1's (i.e. 0^n1^n), that is not possible to represent in regular expression. However, context free grammars (CFGs) would be able to fill in the gap by being both more powerful than regular expressions, but still reasonable enough that we can type check during compile time.

For the above example, we can define the production rules of the CFG to be:

$$S \rightarrow 0S1 \qquad S \rightarrow \epsilon$$

This grammar says that a *String* in this CFG (represented by S) is either an empty *String* (represented by the ϵ), or a 0 followed by S followed by a 1, where S is recursively defined just above. As a result, this CFG can represent the empty *String*, 01, 0011, 000111, etc., satisfying our specification. We can follow the footsteps of our regex Grammar Type and define the syntax for this new Grammar Type to be $String[[S \rightarrow 0S1, S \rightarrow \epsilon]]$. With this new enhancement, we would retain most of the benefits of the regex Grammar Type, but also gain more capability in checking *String* validity.

References

Ali, M., Zibin, Y., Papi, M. M., & Ernst, M. D. (2008). Enforcing reference and object immutability in java. In *Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, OOPSLA Companion '08 (pp. 725–726). New York, NY, USA: Association for Computing Machinery.

Andreae, C., Noble, J., Markstrum, S., & Millstein, T. (2006). A framework for implementing pluggable type systems. *SIGPLAN Not.*, 41(10), 57–74.

Barlas, E., Du, X., & Davis, J. C. (2022). Exploiting input sanitization for regex denial of service. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22 (pp. 883–895). New York, NY, USA: Association for Computing Machinery.

Chong, S. (2019). Basic architecture. `https://groups.seas.harvard.edu/courses/cs153/2019fa/lectures/Lec01-Intro.pdf`, retrieved Aug 2023.

Christensen, A. S., Møller, A., & Schwartzbach, M. I. (2003). Precise analysis of string expressions. *BRICS Report Series*, 10(5).

Cook, W. R. & Rai, S. (2005). Safe query objects: Statically typed objects as remotely executable queries. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05 (pp. 97–106). New York, NY, USA: Association for Computing Machinery.

Costantini, G., Ferrara, P., & Cortesi, A. (2011). Static analysis of string values. In *Proceedings of the 13th International Conference on Formal Methods and Software Engineering*, ICFEM'11 (pp. 505–521). Berlin, Heidelberg: Springer-Verlag.

Costantini, G., Ferrara, P., & Cortesi, A. (2015). A suite of abstract domains for static analysis of string values. *Software: Practice and Experience*, 45(2), 245–287.

Eghbali, A. & Pradel, M. (2021). No strings attached: An empirical study of string-related software bugs. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20 (pp. 956–967). New York, NY, USA: Association for Computing Machinery.

Greenfieldboyce, D. & Foster, J. S. (2007). Type qualifier inference for java. *SIGPLAN Not.*, 42(10), 321–336.

Kiezun, A., Ganesh, V., Guo, P. J., Hooimeijer, P., & Ernst, M. D. (2009). Hampi: A solver for string constraints. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09 (pp. 105–116). New York, NY, USA: Association for Computing Machinery.

Kim, H., Doh, K.-G., & Schmidt, D. A. (2013). Static validation of dynamically generated html documents based on abstract parsing and semantic processing. In F. Logozzo & M. Fähndrich (Eds.), *Static Analysis* (pp. 194–214). Berlin, Heidelberg: Springer Berlin Heidelberg.

Kim, S.-W., Chin, W., Park, J., Kim, J., & Ryu, S. (2014). Inferring grammatical summaries of string values. In J. Garrigue (Ed.), *Programming Languages and Systems* (pp. 372–391). Cham: Springer International Publishing.

Minamide, Y. (2005). Static approximation of dynamically generated web pages. In *Proceedings of the 14th International Conference on World Wide Web*, WWW '05 (pp. 432–441). New York, NY, USA: Association for Computing Machinery.

Nystrom, N., Clarkson, M. R., & Myers, A. C. (2003). Polyglot: An extensible compiler framework for java. In *Proceedings of the 12th International Conference on Compiler Construction*, LNCS 2622 (pp. 138–152). Warsaw, Poland.

Papi, M. M., Ali, M., Correa, T. L., Perkins, J. H., & Ernst, M. D. (2008). Practical pluggable types for java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08 (pp. 201–212). New York, NY, USA: Association for Computing Machinery.

Santino, J. (2016). Enforcing correct array indexes with a type system. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016 (pp. 1142–1144). New York, NY, USA: Association for Computing Machinery.

Tabuchi, N., Sumii, E., & Yonezawa, A. (2003). Regular expression types for strings in a text processing language. *Electronic Notes in Theoretical Computer Science*, 75, 95–113.

Thiemann, P. (2005). Grammar-based analysis of string expressions. In *Proceedings of the 2005 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '05 (pp. 59–70). New York, NY, USA: Association for Computing Machinery.

Wassermann, G. & Su, Z. (2007). Sound and precise analysis of web applications for injection vulnerabilities. *SIGPLAN Not.*, 42(6), 32–41.