A Doctor Thesis

博士論文

A Study on Static and Dynamic Programming Assistance for Embedded Domain-Specific Languages

埋込みドメイン特化言語向け静的および動的プログラミング支援に関する研究

by

Katsumi Okuda

奥田 勝己

Submitted to

the Graduate School of the University of Tokyo

on June 4, 2021

in Partial Fulfillment of the Requirements

for the Degree of Doctor of Information Science and Technology

in Creative Informatics

Thesis Supervisor: Shigeru Chiba　千葉 滋

Professor of Creative Informatics

ABSTRACT

This dissertation presents our research to improve the domain-specific programming assistance for embedded domain-specific languages (DSLs). Using embedded DSLs is a promising approach to efficient software development for specific application domains. The developer of an embedded DSL can implement it with less effort compared to external DSLs. For example, the embedded DSL designer does not need to develop parsers for the embedded DSL since it is implemented as a library or framework for the host language. Moreover, the embedded DSL designer does not need to develop an integrated development environment (IDE) since the DSL user can use IDEs for the host language. However, using the syntax and IDEs of the host language limits the domain-specific assistance for the embedded DSL since only programming assistance for the host language is available in the embedded DSL. To provide better programming assistance for the embedded DSL, the embedded DSL developer has to implement additional tools. This effort reduces the benefit of the embedded DSL. To address this problem, we categorized programming assistance into static one and dynamic one and developed efficient methods for implementing each assistance. The dissertation includes three studies: (1) proposal of lake symbols for island parsing, (2) proposal of interactive grammar editing for island grammars, and (3) design approach to an embedded DSL for dynamic programming assistance. Study (1) and Study (2) achieve reducing the effort to implement static programming assistance for embedded DSLs. On the other hand, Study (3) proposes the importance of dynamic domain-specific assistance for embedded DSLs and the language design that can exploit the IDE for domains-specific assistance.

We define static programming assistance as domain-specific abstraction provided in the syntax level of the language. The key to an efficient implementation of syntax extension is to decrease the effort for developing a parser. The number of rules in the grammar for generating the parser reflects the effort to developing the parser. The island grammar is a promising technique to reduce the number of rules in the grammar by omitting the rule for the uninteresting part of the language. Island grammars are suitable for syntax extension to reduce the number of rules since only extended programming constructs are interesting and the remaining parts are uninteresting. However, the description of practical island grammar is complex because it requires complex definitions of rules to skip the uninteresting part of the language.

The lake symbol proposed in Study (1) eases the description of island grammar. The lake symbol is a novel grammatical symbol similar to nonterminal symbols. The embedded DSL designer can use lake symbols as a wildcard symbol at the place in the grammar where she wants the parser to skip the input until it finds an extended programming construct of interest. The lake symbol automatically calculates symbols called alternative symbols that prevent lake symbols from skipping the interesting part of language as a wildcard. Without lake symbols, the embedded DSL designer must find alternative symbols manually and specify them in the island grammar. Previous work has been tackled the same problem to ease the description of island grammars. However, it calculates the subset of alternative symbols. This limits the place in the grammar where the parser can skip the uninteresting part of the language. Our lake symbols relax this imitation.

While the lake symbols ease the description of island grammar, writing island grammar is not easy. The description of correct island grammar requires iterations of trial-and-error. Hence, an efficient way to editing island grammars is required. Based on this motivation, Study (2) proposes the interactive editing method and tool called PEGSEED. With PEGSEED, the language designer can write a working island grammar in a step-by-step manner. In each step, she adds a rule for a new island.

After adding a new rule, she can test the grammar on an example text by highlighting the text area recognized by the latest rule. A rule for a new island can be added by concatenating already tested islands. By incrementally refining the island grammar tested in each step, DSL designers can efficiently get the expected island grammar. PEGSEED also provides GUI operations to add a new rule by using an example text. By selecting a text area and applying one of the GUI operations. The user can add a new rule without writing it by hand. Our case study shows that the parsers for syntax extension can be available only with the GUI operations provided by PEGSEED.

In Study (3), we introduce the importance of domain-specific programming assistance for embedded DSLs. We define auto-completion and error checking provided by IDEs as dynamic programming assistance. Careful language design enables dynamic domain-specific programming assistance via an IDE for the host language. We demonstrate this with our practical processor description language called MELTRANS. Our case study shows that domain-specific assistance can be available by exploiting an IDE for the host language. Moreover, because our design approach does not need to customize the IDE or develop a specialized IDE, it does not sacrifice the benefit of embedded DSLs.

Acknowledgements

Professor Shigeru Chiba supervised this dissertation. I would like to express my deep gratitude to him for his thoughtful guidance. His insightful advice always helped me expand my ideas related to this dissertation.

I greatly thank my thesis committee, Hiroshi Saruwatari, Hiroshi Esaki, Kenjiro Taura, Ryota Shioya, and Tomoharu Ugawa. They gave me their constructive suggestions through the review process.

I thank Haruhiko Takeyama, Nobutoshi Todoroki, Norihiro Nishiuma, and Akihiko Higuchi. They were managers in Advanced Technology R&D Center, Mitsubishi Electric Corporation, while I was a student in the doctoral course at the University of Tokyo. Without their special treatment, I would not have concentrated on my research for this dissertation.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

It is a common understanding that both software scale and the application domains are increasing more than ever. Developing an efficient method to implement software is one of the most critical research topics in software engineering. One promising approach for efficient software development is using domain-specific languages (DSLs) for software construction. A DSL is a programming language that is designed for a specific application domain. In general, a DSL provides easy-to-use programming interfaces to develop software for the domain. These programming interfaces enable the user to develop her software efficiently.

The efficient development of DSLs is also essential to apply them to software development for their domains. A DSL is designed and implemented to develop software for a specific domain efficiently. Therefore, the higher the labor to develop the DSL becomes, the smaller the benefit of using the DSL is.

Developing a DSL as an embedded DSL is an efficient method to develop the DSL. Embedded DSLs are DSLs that are embedded in a general-purpose programming language. In many cases, embedded DSLs are implemented as libraries or frameworks written in the host language. In contrast to embedded DSLs, DSLs implemented from scratch are called external DSLs. The development of an external DSL resembles the development of a general-purpose programming language. For example, the language developer has to implement a parser to analyze a source program. On the other hand, when developing an embedded DSL, the developer does not need to implement a parser since the embedded DSL exploits the syntax of the host programming language. For the same reason, the integrated development environment (IDE) for the host language can be used when writing the program in the embedded DSL. Hence, the language developer does not need to implement an IDE for the embedded DSL. In general, development efforts for embedded DSLs are lower than for external DSLs.

1.1 Motivation

The motivation of this dissertation is to reduce the effort to implement domain-specific programming assistance in embedded DSLs. Though sophisticated programming assistance is essential to develop a program efficiently, it is not adequate in embedded DSLs due to its implementation cost. Domain-specific assistance includes syntax level support, auto-completion, and error checking available at programming. While an embedded DSL can exploit the general-purpose

syntax level support such as the *class* mechanism or the *lambda* expression, the expressiveness of the embedded DSL is limited to the host language. Similarly, while IDEs for the host language may provide auto-completion and error checking, the assistance is limited to general-purpose ones for the host language.

In principle, any domain-specific assistance provided in external DSLs can also be implemented in an embedded DSL. However, it is not practical due to its additional implementation efforts. To provide domain-specific assistance in an embedded DSL, the language developer must implement additional tools equivalent to tools for external DSLs. These implementation efforts of additional tools diminish the benefit of the low implementation cost of embedded DSLs. To implement domain-specific programming assistance in embedded DSLs, we need reasonable methodologies for developing it.

1.2 Approaches

We categorized domain-specific programming assistance into static one and dynamic one and studied them, respectively. We define static programming assistance as support available in all life-cycle of programs. On the other hand, we define auto-completion or error checking as dynamic ones since they are available while the user writes the program.

1.2.1 Approaches to Static Programming Assistance

Static programming assistance includes library support for the language, documentation of the language, and syntax-level support for the language. Methodologies for supports of library and documentation in embedded DSLs are not different from ones for the general-purpose programming languages. Therefore we focused on improving the syntax-level support.

For example, even if the use of the `unless` statement is more natural than the use of the if statement in the application domain, the use of the `unless` statement is impossible if the host language does not support it. For example, even if the user wants to write a program with the `unless` statement as follows:

```
unless (isConditionPassed()) {
    doit();
}
```

, she may have to write a program without unless statement as follows:

```
if (!isConditionPassed()) {
    doit();
}
```

The difference may look small. However, if there are many places where the `unless` statement is natural, the difference will not be neglectable. If we designed the DSL as external DSL, we would implement the unless statement in the DSL.

Syntax extension is a technique to improve the expressiveness of programs in embedded DSLs. It enables the use of domain-specific programming constructs in a program written in the embedded DSL. For example, syntax extension enables the use of the `unless` statement even if the host programming language does not support it.

Syntax extension is not widely used in embedded DSLs due to the implementation cost of a parser. Syntax extension is implemented by an external tool called a program transformer. Program transformers take a program with domain-specific programming constructs and convert it into an equivalent program without domain-specific programming constructs. The structure of program transformers resembles compilers. They convert an input program into a tree with their parsers and generate a lower-level program by traversing it. The efficient development of program transformer is a key to make syntax extension applicable to embedded DSLs. Language tools such as MetaBorg [15] and TXL[18] have been proposed to develop program transformers easily. They provide language support to write transformation rules for programming constructs. The user of these tools has to write only transformation rules for extended programming constructs in addition to a grammar for the parser. Nevertheless, these tools have not been widely used in practical embedded DSLs. These tools do not reduce the effort to write a parser.

Grammar-based parser construction is a best practice to obtaining parser today. There are a lot of parsing algorithms based on grammar. Many parser generators, such as Yacc based on LALR(1) grammar and ANTLR based on LL(*) grammars, have been developed. These grammars are not ambiguous in common. However, extending the syntax of an existing language tends to result in ambiguous grammar. Hence, unambiguous grammar-based parsing algorithms are not suitable for syntax extension. Instead, more flexible parsing algorithms such as Cocke-Younger-Kasami (CYK)[96], GLL[83], GLR[88], or packrat parsing have been used for syntax extension. The effort to implement a parser is reflected in the grammar size regardless of which parsing algorithm is used.

An island grammar is a well-known technique to reduce the size of grammar for a parser. Island parsers extract only programming constructs of interest as *islands* from an input text, and they ignore the rest of the text as *water*. Such *incomplete* parsers are often easy to develop but still useful enough for many software engineering tools, including syntax extension. These tools do not need the complete parse tree or the abstract syntax tree (AST) of the input program. They only need a parse tree or an AST for the interesting parts of the program. Suppose that we develop a program transformer for the `unless` statement. The program transformer only needs a parse tree for `unless` statements. We can ignore the rest of the program. An island grammar fits this application; the statements like `unless` are *islands* while the rest is *water*. An island parser is also suitable for source model extraction from incomplete program [67], multilingual parsing [87, 90, 2], extracting programming constructs from documentations [9, 81], and lightweight impact analysis [68].

Island grammars are described in a formal language such as PEG (Parsing Expression Grammar) [32], SDF (Syntax Definition Formalism) [44], or TXL [20, 19]. These grammars consist of the rules for islands and water. The number of rules in an island grammar is often smaller than the corresponding full-featured grammar, which is for parsing a whole program. The rules for the water are usually simple, in an ideal case, just one wildcard character, and thus the grammar for island parsing consists of only a small number of rules.

In practice, the grammar for island parsing consists of a small number of rules only when most parts of a program are recognized as water. To do so, some inner parts of the island must also be water, which is not fully parsed. For example,

if an island is a `if/else` statement, the expressions included in that statement should be recognized as water. However, the rules for such water in the middle of an island are complicated and difficult to describe. Due to this difficulty, island parsing seems to be not widely used today despite its applicability to software engineering tools.

In this dissertation, we propose the lake symbols to mitigate the difficulty in describing the rules for the water in the middle of an island. We call this water a *lake*. A lake symbol is a special symbol used in a grammar to represent a lake in the grammar. In the simplest case, the lake symbol works as a special wildcard that matches any character except the parts of the island. The parser needs to know what text patterns must not be taken as water to exclude the parts of the island from the lake. These patterns are given by terminal or non-terminal symbols, which we call the *alternative symbols*. The lake symbol automates the enumeration of the alternative symbols so that it can be used as a simple wildcard-like symbol. The user can also specify inner programming constructs in the lake by writing the rule for the lake symbol. This feature enables the handling of nested islands inside the lake. We also propose an extension to PEG for supporting our lake symbols. We present an algorithm to translate this extended PEG into the normal PEG, which can be used as an input to an existing PEG-based parser or parser-generator.

Furthermore, we implemented *PEGIsland*, an island-parser generator supporting our extended PEG. By using PEGIsland, we implemented 36 island parsers for Java programs and 20 ones for Python programs. We compared the number of their grammar rules with the number of the rules for the full-featured Java/Python parser. The comparison revealed that our extended PEG effectively reduces the number of rules.

While lake symbols ease the creation of lakes in islands, debugging island grammar is still not easy. If we fail to write some rules in the grammar correctly, the parsing result is not what we expect. If the grammar were not an island grammar but a complete one, the debug would be easier. We could check that each rule in the grammar is correct by comparing it with the language specification. In the case of the island grammar, we cannot compare the rules in the island grammar with the language specification since most parts of original grammar are omitted and modified in the island grammar.

To check whether the island grammar is correct or not, we repeat trial-and-errors. Therefore we need a way to confirm the parsing result interactively and effectively step by step while editing an island grammar. While interactive tools for generating parsers such as Parsify [61] and Parsimony [60] have been proposed, they do not support island grammars. They provide a way to editing grammar interactively through GUI operations by using an example program. For example, the user selects expressions in the example program and labels them as expressions. The user can get an unambiguous grammar by repeatedly applying operations. However, the grammar is ambiguous until the user finishes labeling all the programming constructs in the language. The resulting grammar is not an island grammar but a complete grammar.

We propose an incremental description of island grammars with lake symbols and tool support for it. Our approaches enable the description of an island grammar in a step-by-step manner. In each step, the user can add a rule to the grammar and immediately test the rule. This enables the island parser to

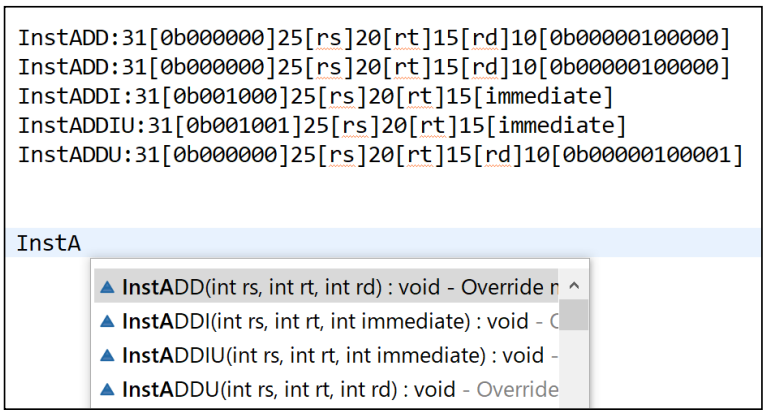

Figure 1.1: Auto completion provided by the IDE

always partially work as what the user expects. This prevents us from repeating inefficient trial-and-error to write an island grammar. We also propose GUI operations for the user to add a new rule to the grammar. These GUI operations take an example text as input and generate a new rule based on the user's selection. This prevents the user from making a specific type of mistake when editing an island grammar.

1.2.2 Approaches to Dynamic Programming Assistance

We define dynamic programming assistance as support that is available while the programmer is writing a source code. Usually, such assistance is provided by editors or IDEs for a language. While programmers write a source code in an IDE, the IDE provides auto-completion, error checking, and text highlighting.

While embedded DSL users can use IDEs for the host language, domain-specific assistance tends to be insufficient. Let us suppose that we design a processor description language. In this language, the user writes the formats and semantics of each instruction. The semantics of an instruction are described with the instruction's operands specified in the corresponding instruction format. Hence, the IDE for this language should provide auto-completion when the user writes the semantics of an instruction by using the format description of the same instruction as shown in Figure 1.1. In the DSL program in Figure 1.1, formats of several instructions have already been defined. The IDE shows the list of possible instructions to be defined by using the formats of instructions. However, if we implement equivalent processor description language as embedded DSLs, this type of auto-completion will no be available. Available dynamic assistance in embedded DSLs is limited to the one provided for the host programming language. To provide domain-specific assistance not covered by assistance for general-purpose one, the language developer must implement a domain-specific IDE for this purpose.

While methodologies for developing IDEs have been proposed, they are not suitable for embedded DSLs. Language workbenches such as Xtext and JetBrain MPS provide a way to develop an IDE for DSLs. For example, the user of Xtext can generate a plugin for Eclipse IDE for her DSL by writing a grammar for her DSL. However, the grammar must be a complete grammar of the DSL. In the

case of embedded DSL, writing the complete grammar is not realistic. Even if the DSL designer uses Java, she will not want to describe a complete grammar for Java. What we want is a way that does not require the development of a specific IDE.

To provide domain-specific assistance with minimum effort, we propose a language design pattern called the *divide-and-generate* pattern. In the *divide-and-generate* pattern, the embedded DSL program is divided into multiple concerns. The DSL runtime takes concern and generates a support code for another concern. When the DSL user writes the subsequent concern, the domain-specific assistance is provided via the support code. The support code is written in the host language. Therefore IDEs for the host language can provide auto-completion and error checking. This assistance is for the host language, but we can also regard it as a domain-specific one. An example of a support code is a superclass that declares methods that the subclass should implement.

To demonstrate our idea, we implemented practical processor description language MELTRANS and experimented on it. Processor description is one of the successful domains for which many DSLs have been developed. An example of processor description language implemented as an embedded DSL is Pygin hosted in Python. We implemented our MELTRANS as an embedded DSL hosted in Java with our *divide-and-generate* pattern. MELTRANS is used for generating fast processor emulators from the specifications of instruction set architectures. We divide processor description into six concerns and implement MELTRANS's runtime that generates superclasses for these concerns. We confirmed that MELTRANS could provide auto-completion and error checking, which are not available in Pygin processor description language. A research question is whether domain-specific assistance does not have a negative impact on other aspects of processor description language. To answer this question, we count the lines of code (LOC) for the ARM architecture and compare the results with Pygin. We also measured the performance of generated and compared the results with Pydgin and the state-of-the-art QEMU. The results show that the *divide-and-generate* pattern does not negatively impact the program's code metrics and the performance of the generated emulator. The LOC of the program for ARM is equivalent to the LOC of the program written in Pydgin. Similarly, the performance of the emulator generated by MELTRANS is equivalent to the other emulators.

1.3 Positioning

1.3.1 Study on Languages for Describing Grammar

Our study on lake symbols for syntax extension is a kind of study on languages for describing grammar. The position of our study is depicted in Figure 1.2. Tools for syntax extension have been proposed, such as MetaBorg [15] and TXL [18]. These tools provide a way to write a grammar and rewriting rules for extended syntax. While the effort to write rewriting rules is small, the effort to write a grammar is not diminished since these tools require a complete grammar for the extended syntax.

TIGEREYS [26] proposed the use of an island grammar in syntax extension to reduce the effort to write many grammar rules for syntax extension. Island

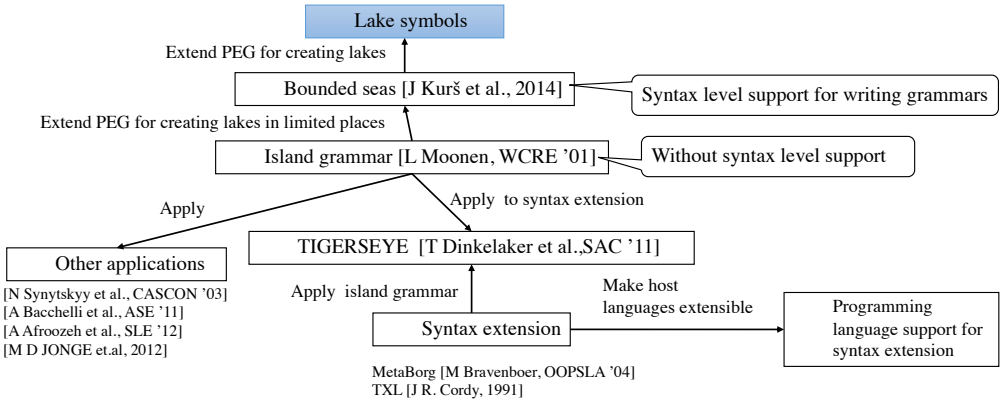

Figure 1.2: Study on languages for describing grammar

grammar is introduced by Moonen [67] as a technique to reduce the effort to write a grammar for an application that is only interested in specific programming constructs. While we focus on syntax extension as an application of island grammar, there are studies on other applications including source model extraction from incomplete program [67], multilingual parsing [87, 90, 2], extracting programming constructs from documentations [9, 81], and lightweight impact analysis [68].

Bounded seas introduced the difficulty in writing island grammars and proposed water operator in PEG. The user can use the water operator as a special wildcard to skip uninteresting parts of an input program as water. While it eases the description of island grammar, the place where the water operator can be used is restricted. For example, the water operator does not work as expected at the beginning of an operand of an ordered choice operator. Our lake symbol is yet another syntax support for writing island grammars. While lake symbols can be used similar to the water operator, they can be used almost anywhere in a grammar as a special wildcard symbol for an advanced shortest match for writing an island grammar.

1.3.2 Study on Grammar Editors

Related work of our study on grammar editor is depicted in Figure 1.3. GUI-based parser generators have been proposed, such as Parsify [61] and Parsimony [60]. They provide operations to refine the grammar for a parser by applying the programming by example technique to parser generation. These tools aim to obtain a fully functional parser with a complete grammar. Hence, the working parser is not available until the user teaches all the programming constructs in a language.

Our interactive grammar editor proposed in Chapter 4 is based on our lake symbols proposed in Chapter 3. Our iterative grammar editor called PEGSEED uses a lake symbol like a test stub for software under testing. Therefore, a functional parser is always available from a grammar being edited in PEGSEED. This feature is convenient to write a grammar for extracting only extended programming constructs.

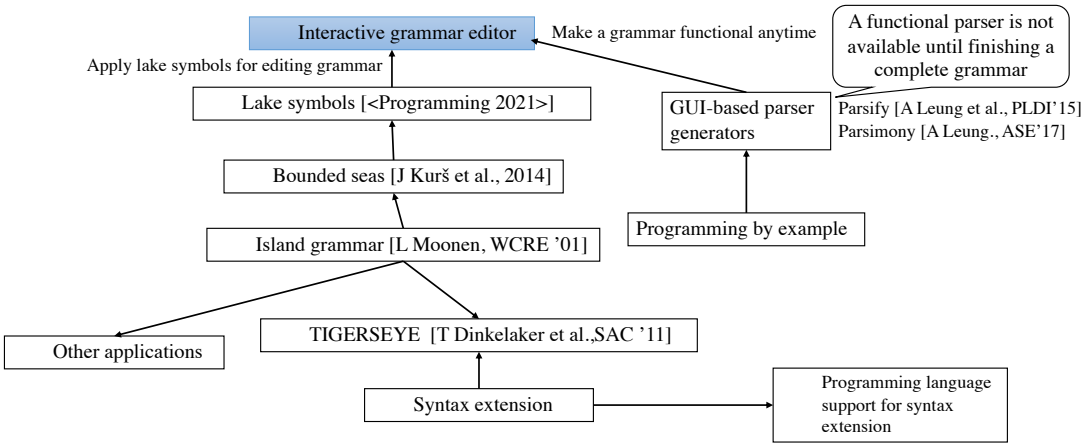

Figure 1.3: Study on grammar editors

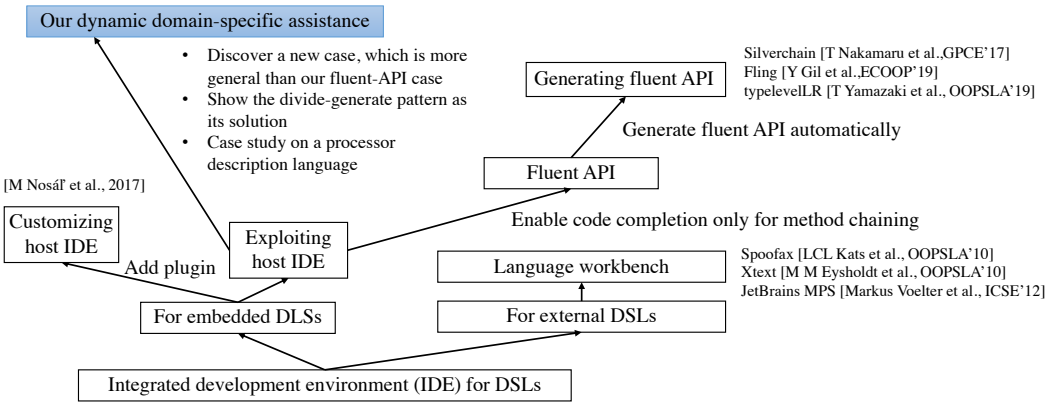

Figure 1.4: Study on dynamic domain-specific assistance

1.3.3 Study on Dynamic Domain-Specific Assistance

Related work of our study on dynamic domain-specific assistance is depicted in Figure 1.4. Dynamic assistance is usually provided by integrated development environments (IDEs). Language workbenches that support efficient development of IDEs for external DSLs have been proposed, such as Xtext [30] and Spoofax [50]. While these techniques can also be used for embedded DSLs, developing an IDE for an embedded DSL is too expensive.

For providing dynamic domain-specific assistance for embedded DSLs, two approaches have been proposed. The first approach is customizing an IDE for a host language. Nosal et al. [70] have reported their study on extending NetBeans IDE for non-programmers.

The other approach is exploiting an IDE of a host language. Fluent API can be used to enable code completion and error checking for method chaining. While the implementation of fluent API is not easy, tools for generating fluent API have been proposed, such as Fling [36], Silverchain [69], and typelevelLR [94].

Our study on dynamic domain-specific assistance also focuses on exploiting an IDE for a host language. Our study discovers a new case for code completion

and error checking, which is more general than the fluent API case. We also propose the *divide-and-generate* pattern for its solution and validate our pattern through a case study on a practical processor description language.

1.4 Contributions

The contributions of this dissertation are summarized as follows:

- We propose the lake symbols and an extended PEG supporting the lake symbols. It mitigates the difficulty in describing the grammar rule for the water in the middle of an island. This contributes to reducing the total number of rules for an island parser and thereby improving the usefulness of the island parser.

- We introduce a novel semantics of the shortest match suitable for island parsing. Our shortest match with a lake symbol consumes an arbitrary input character until a part of the whole of an island appears.

- We implemented island parsers based on the extended PEG and revealed that the extended PEG effectively reduced the size of the grammar for island parsing.

- We propose an incremental description of island grammars with lake symbols and tool support for it. Our approaches enable the description of an island grammar in a step-by-step manner. In each step, the user can add a rule to the grammar and immediately test the rule. This enables the island parser always to work correctly as the user expects. This prevents us from doing inefficient trial-and-error to write an island grammar.

- We propose an example-based GUI tool for editing grammar. This tool provides three GUI operations for the user to add a new rule to the grammar. These GUI operations take an example text as input and generate a new rule based on the user's selection. This prevents the user from making a specific type of mistake when editing the island grammar.

- We implemented our tool called PEGSEED that supports incremental editing and GUI operations for editing island grammars. We developed a framework for syntax extension based on PEGSEED. We confirmed that the possible mistake is detected soon in our PEGSEED. We also confirmed that we could construct the parser for syntax extensions only with proposed GUI operations.

- We reveal that domain-specific programming assistance by IDEs is inadequate for embedded DSLs and propose the *divide-and-generate* pattern to addressing this problem. The *divide-and-generate* pattern exploits program generation such that programming assistance for the host language can be regarded as domain-specific.

- We conduct a case study by developing a practical embedded DSL called MELTRANS and demonstrate the domain-specific assistance that is available when the user is writing a program in MELTRANS.

- We experimentally confirm that the domain-specific assistance provided by MELTRANS can effectively reduce the amount of code to be written by the user, the generated emulator can emulate a processor in over 1,000 MIPS, and MELTRANS is general enough to generate an emulator for several commercial instruction sets (ARM, MIPS64, SH, RH850, RISC-V, and RX).

1.5 Structure of This Dissertation

The remaining part of this dissertation is composed as follows:

Chapter 2: Embedded DSLs

We introduce embedded DSLs and existing approaches for supporting domain-specific assistance in them. We clarify what problems have not been solved yet and what we address in this dissertation.

Chapter 3: Lake Symbols for Island Parsing

Chapter 3 proposes a methodology to reduce the effort to implement static programming assistance in embedded DSLs. We propose the lake symbols for island parsing. The lake symbol is a novel grammatical symbol similar to a nonterminal symbol. Embedded DSL developers can use lake symbols to describe an island grammar for syntax extension. The Lake symbol can be used in a similar manner with a wildcard character to skip uninterest parts of the program. In contrast with a wildcard character, a lake symbol does not consume island parts of the input program. This feature is archived by automatically calculating a set of alternative symbols, which recognize parts of islands. Our algorithm proposed in this chapter takes grammar with lake symbols and generates an equivalent grammar without lake symbols. Automatically calculated alternative symbols are embedded in the rules derived from lake symbols. Without lake symbols, the user must enumerate all alternative symbols and explicitly specify them in the grammar.

Chapter 4: Interactive Editor for Island Grammars

Chapter 4 also proposes a methodology to reduce the effort to implement static programming assistance in embedded DSLs. We propose an interactive editor called PEGSEED for island grammar. In PEGSEED, the user incrementally refines island grammar from the initial grammar, which skips a whole program as water. The user adds a rule that recognizes small islands to the grammar in a step-by-step manner. In each step, the user can test the latest rule with text highlighting provided by PEGSEED. We also propose example-based GUI operations to update an island grammar. By using GUI operations, embedded DSL developers can describe an island grammar for syntax extension without directly editing the grammar.

Chapter 5: Dynamic Domain-Specific Programming Assistance

In Chapter 5, we reveal that dynamic domain-specific assistance is poor in an embedded DSL. We also propose a design approach for developing an embedded DSL with dynamic domain-specific programming assistance. To demonstrate the proposed approach, we describe the design of our processor description language called MELTRANS, which is an embedded DSL hosted by Java. MELTRANS is used to generate a fast processor emulator with dynamic binary translation.

Chapter 6: Conclusion

We conclude this dissertation and explain the future work related to our studies.

Chapter 2

Domain-Specific Assistance in Embedded DSLs

2.1 Domain-Specific Languages

Domain-specific languages are languages that are developed for providing better domain-specific abstraction than general-purpose programming languages. For example, Dot is a domain-specific language for generating a graph image. The PNG file corresponding to Figure 2.1 can be generated from the following program written in Dot:

```
digraph {
  Car -> Vehicle;
  Motorbike -> Vehicle;
  Vehicle -> Machine;
}
```

Dot provides a domain-specific abstraction for the user to draw a graph. The programmer of Dot specifies the relations between nodes in the graph. The Dot interpreter Graphviz layouts each node on behalf of the programmer and draws the graph in the PNG file from the above program. Similarly, LaTex is a DSL for generating a well-formatted document and SQL is a DSL for manipulating data held in a relational database management system.

DSLs are categorized into external DSLs and embedded DSLs based on how they are implemented. External DSL is implemented similarly with general-purpose programming languages, as shown in Figure 2.2. The developer of an external DSL implements a DSL interpreter or compiler. Figure 2.2 also shows the structure of the DSL interpreter. The DSL interpreter consists of a lexer, parser, and evaluator, as same as interpreters for general-purpose programming languages. The lexer takes a DSL program and creates a stream of tokens. The parser takes the stream as input and creates an abstract syntax tree (AST) representing the program's semantics. The evaluator traverses the AST and does its domain-specific tasks, such as generating a graph image.

On the other hand, an embedded DSL is implemented as a framework or a library for a general-purpose programming language, as shown in Figure 2.3. The general-purpose programming language is called the host language of the embedded DSL. Embedded DSL exploits the syntax of the host language. Therefore, the embedded DSL developer does not need to implement a lexer and a

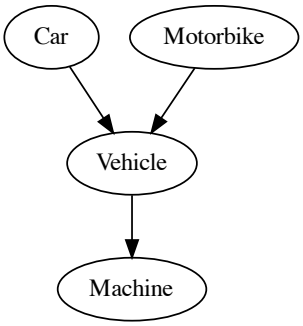

Figure 2.1: Example graph generated by a program written in Dot

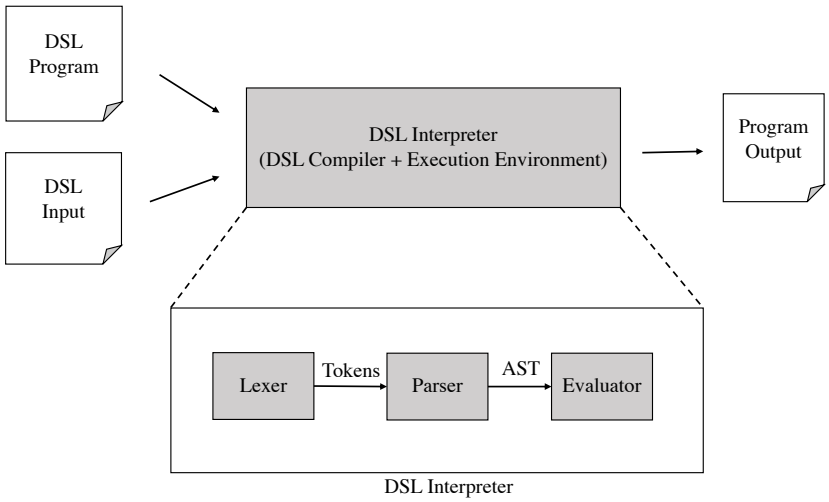

Figure 2.2: External DSL developers implement an interpreter

parser for her embedded DSL. Domain-specific functionality corresponding to the evaluator for external DSL is implemented in the host library or framework written in the host language.

2.2 Why Embedded DSLs?

An essential factor of a DSL is its implementation cost. A DSL is developed to accelerate the development of software in a specific domain. In many cases, a DSL is developed at the same time with its application. Therefore, the effort to develop DSL must be reasonable.

The benefit of developing a DSL as embedded DSL is its low implementation cost. In general, the effort to developing an embedded DSL is lower than external DSLs. As mentioned earlier, since an embedded DSL exploits the syntax of

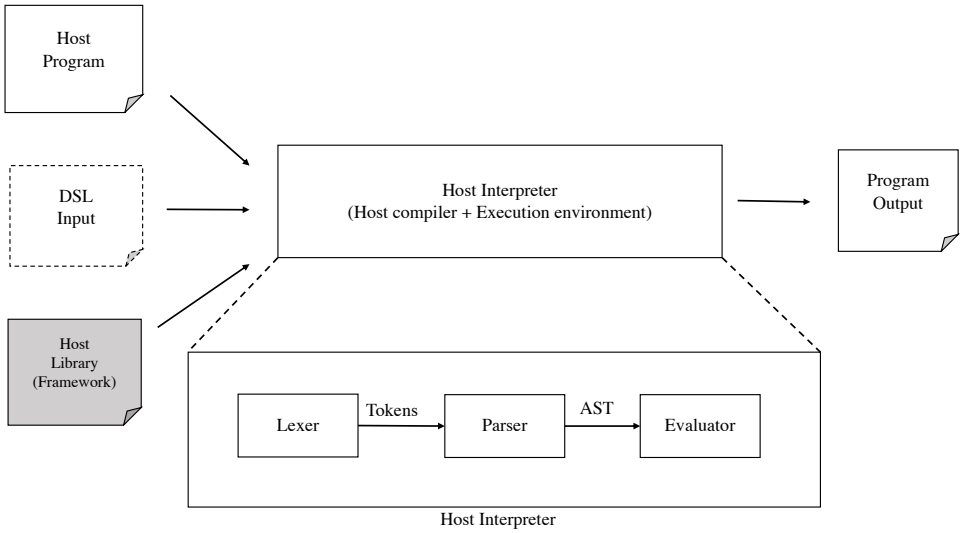

Figure 2.3: Embedded DSL developers implement a library

the host language, the development of the embedded DSL does not include implementing a lexer and a parser, which is needed for external DSLs. Moreover, exploiting the syntax of the host language enables the use of IDEs for the host language. The programmer of the embedded DSL can use IDEs for the host language when writing a program. Therefore, the DSL developer does not need to provide a specific IDE for the DSL.

An example of embedded DSLs is Rake[1] hosted in Ruby. Rake is a language for building software like Make. The following program is an example program written in Rake:

```
file "message.txt" do |task|
  system("echo Hello!> #{task.name}")
end
```

The code specifies the rule for creating a file named "message.txt". The `file` command in Line 1 is just implemented as a library function and provides a domain-specific abstraction to the user for specifying the rule to generate a file. The above Rake program is equivalent to the following program in Make, which is an external DSL for software building:

```
"message.txt":
    echo Hello! > $@
```

While the program written in Make is slightly simpler than the one in Rake, the development effort of Rake is lower than Make. The development of Rake does not require writing a lexer and parser for Ruby. Instead, it requires the implementation of library functions, such as the `file` function in the above example. Moreover, when the user writes a program in Rake, she can use an IDE for Ruby.

[1] `https://docs.ruby-lang.org/ja/latest/library/rake.html`

2.3 Programming Assistance in DSLs

Programming assistance helps the user of a DSL to write a program in the DSL. Depending on when the assistance is available, programming assistance in DSLs is divided into static and dynamic ones. In this thesis, we define static assistance as support available throughout the life-cycle of a program written in a DSL. Similarly, we define dynamic assistance as dynamically available support while the programmer is editing a program in a DSL in a specific environment. Libraries, documentation, and syntax of the host language provide static programming assistance. They help the user of the language throughout the life-cycle of a program. On the other hand, auto-completion, error checking, and syntax highlighting, which are typically provided by IDEs, are dynamic programming assistance. They are available only when the DSL user edits a program.

Programming assistance in an embedded DSL is usually provided by the host programming language and tools for it. For example, dynamic programming assistance in an embedded DSL hosted by Java is provided by the Eclipse IDE for Java. Programming assistance available in an embedded DSL is not domain-specific but general-purpose for the host language. To provide domain-specific programming assistance in embedded DSLs, many researchers have studied so far. The remaining parts of this chapter introduce existing research on programming assistance for embedded DSLs.

2.4 Static Domain-Specific Programming Assistance

Since static programming assistance provided by libraries or documentation is not specific to embedded DSL, we focus only on syntax-level support for programming. Syntax-level support in embedded DSLs is relatively poor. While external DSLs can provide rich syntax-level support in nature, the degree of syntax-level support available in an embedded DSL depends on its host programming language. Usually, the syntax-level support in an embedded DSL is only limited to the general-purpose ones provided by the host language. While an embedded DSL user can use programming constructs such as *class* or *lambda* provided by the host language, she cannot use any domain-specific programming construct. If the embedded DSL developer wants to add domain-specific programming constructs to the embedded DSL, extra effort is required for syntax extension.

2.4.1 Syntax Extension

Syntax extension is a solution to provide domain-specific syntax-level support in embedded DSLs. DSL developers extend the syntax of the host language by adding new programming constructs to the host language. Suppose that `unless` statements are essential for writing a natural program in a target domain when the host language does not support `unless` statements. An `unless` statements execute its block statement only when a given condition is not satisfied. An example of an unless-statement is as follows:

```
unless (isConditionPassed()) {
    doit();
```

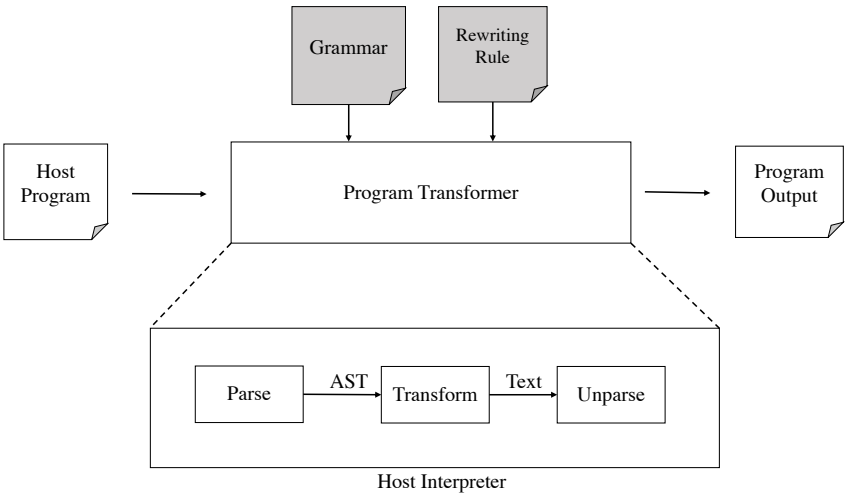

Figure 2.4: Embedded DSL developers write a grammar and rewriting rules

```
}
```

In this case, `doit()` is called only when `isConditionPassed()` returns false. This program is equivalent to the following program:

```
if (!isConditionPassed()) {
    doit();
}
```

The above program does not contain any `unless` statement. Hence, compilers or interpreters of the host language can handle this program. Syntax extension with `unless` statement can be implemented by a *program transformer* that converts a program with `unless` statement into an equivalent program without `unless` statement.

2.4.2 Program Transformer

Program transformers are tools for syntax extension. They convert a program written in language A into an equivalent program written in language B. Syntax extension is implemented with a program transformer. The program transformer can transform a program written in an extended language into an equivalent program written in the original language without extension. The typical structure of a program transformer is shown in Figure 2.4. A program transformer is configured by a grammar and rewriting rules. The grammar specifies the grammar of the input program, and the rewriting rules specify how to convert input programs. Embedded DSL developers can implement extended syntax by writing the grammar of the extended language and rewriting rules for new constructs in the extended language. The program transformer parses the input program based on the given grammar and generates an AST of the program. Then, the program transformer rewrites the AST based on the given rewriting rules. Finally, the program transformer unparses the AST and generates a program written in the language without extended syntax.

MetaBorg [15] and TXL[18] are language tools that can be used as a program transformer. MetaBorg takes a grammar written in the syntax definition formalism (SDF) [89] and rewriting rules written in the program transformation language Stratego/XT [14]. TXL is a language for language prototyping. It provides a syntax for a grammar and rewriting rules.

Describing grammar requires more effort than describing rewriting rules for a program transformer. While syntax extension requires only a rewriting rule for newly added programming constructs, it usually requires a complete grammar containing all the extended language's grammatical rules.

2.4.3 Grammar for Parsing

A grammar for obtaining a parser is written in a specific metalanguage. For example, Yacc takes a grammar written in metalanguage similar to Backus-Naur form (BNF) [53]. Many software translation systems adopt syntax definition formalism (SDF) as a metalanguage for describing a grammar. Metalanguage such as BNF and SDF can be used to describe a context-free grammar (CFG). Another option for describing a grammar is using parsing expression grammar (PEG) [32]. The word "PEG" is used to represent two different things. First, PEG means one of the metalanguages such as BNF for describing grammar. Second, a PEG represents one of the formal grammars such as CFG.

2.4.4 Parsing Algorithm for Syntax Extension

Parsing algorithms that can handle CFG or PEG are suitable for syntax extension, while many parsing algorithms can be used to parse a program written in programming languages. For example, scanner-less GLR [92] is used in MetaBorg, and GLL [84] is used in TXL. Scanner-less GLR and GLL are CFG parsers. GLR [88], Cocke-Younger-Kasami (CYK) [96], and Early algorithms are also CFG parsers. These CFG parsers can handle any context-free languages. If the PEG is used for syntax extension, packrat parsing [31] can be used as a parsing algorithm.

Other algorithms such as LL(1) or LR(1) are not suitable for syntax extension, while they are often used for programming languages. LL(1)/LR(1) parsers can handle only LL(1)/LR(1) languages. However, adding a domain-specific programming construct in a host language tends to result in a language in a different class of the host language. For example, adding a new construct in a LL(1) language such as Pascal may result in a language that is not an LL(1) language. Whether the resulting language is LL(1) depends on how complex the added programming construct is. Hence, the expressive of extended language is restricted by a parsing algorithm. The more languages the parsing algorithms can handle, the more expressiveness is available by syntax extension.

2.4.5 Island Grammar

An island grammar is a promising technique to reduce the labor for describing grammar for syntax extension. The name "island grammar" was initially introduced by Van Deursen *et al.* [90] as a grammar development technique to reduce the grammar development time. Moonen [67] defined island grammar as quoted as follows:

> An island grammar is a grammar that consists of detailed productions describing certain constructs of interest (the islands) and liberal productions that catch the remainder (the water).

If the application program is only interested in specific constructs in a language, the grammar for the application does not need to be complete grammar for the language. Instead, the grammar contains the detailed rules for islands that the application is interested in and the rules to skip the rest of the text. Such a grammar is called an island grammar. In many cases, the rules for water are not complicated as corresponding rules in the complete grammar. Therefore, the island grammar is relatively smaller than the complete grammar.

An island grammar does not depend on any specific formalism for the definition of syntax. In literature, SDF and PEG are often used to describe island grammars.

For example, the following grammar is typical island grammar written in PEG:

```
program <- sea*
sea <- island / water
```

`program` is recognized as repetition of `sea` in the grammar. The sea is recognized when the parser recognizes islands or water. When both island and water match the substring starting with the current location in parsing, an island is always preferred because the left-hand operand of choice operator `/` in PEG is prioritized to the right-side operand.

2.4.6 Application of Island Grammar to Syntax Extension

The idea that using island grammar for syntax extension has been introduced by Afroozeh *et al.* [2]. They reported a case study on an island grammar for embedding the Tom language [76] in some host languages. Tom can be embedded in a host language such as Java without a complete grammar for the host language by using island grammar. Hence, in the island grammar, Tom's program constructs are described as islands, and water is defined to skip the parts of a program written in the host language. The resulting grammar is almost host language agnostic. Hence, the grammar can be reused for embedding Tom to multiple host languages. The island grammar resulted in having ambiguity. Afroozeh *et al.* [2] reported disambiguation techniques that can be used with GLL parsing to solve the conflicts in the grammar for embedding Tom. The techniques include accommodating the input grammar or postprocessing a parse forest based on pattern matching within the parse forest. Tom's construct starts with a specific delimiter and ends with the closing brace. While postprocessing a parse forest is general enough to be used in other guest languages, defining string literal start withs '' and ends '' as water is a useful technique that can be used as a universal technique.

2.4.7 Bounded Seas for Island Grammar

Defining water in an island grammar is not easy. The definitions of water differ depending on where each water is in the grammar. Bounded seas are a technique that eases the definition of water in an island grammar. Bounded seas

introduced the sea operator ~ for defining a bounded sea in an island grammar. A bounded sea is defined as an island surrounded by context-aware water. For example, `~island~` is a bounded sea defined by the nonterminal symbol `island`. A bounded sea can be used in an island grammar instead of specifying the fully detailed rules for parsing the corresponding parts of the language.

For example, we can write an island grammar for extracting nestable blocks that begin with `'{'` and end with `'}'` as follows:

```
program <- (block / .)*
block <- '{' ~(block / ϵ)~* '}'
```

The parser corresponding to the grammar recognizes a program as repetition of block or any character. Note that the above grammar specifies a bounded sea instead of specifying the details of the inner parts of the block. In this case, the bonded sea is represented as `~(block / ϵ)~`. When the parser parses the inner parts of a block surrounded by `'{'` and `'}'`, the parser skips the input until it encounters an inner block or closing `'}'`. The grammar is equivalent to the following PEG without the sea operator:

```
program <- (block / .)*
block <- '{' ((!'}' !'{' .)* (block / ϵ) (!'}' !'{' .)*)* '}'
```

In the above grammar, the sea operator ~ is converted to parsing expression `(!'}' !'{' .)*`, which consumes any characters until it encounters `'}'` or `'{'`. `'}'` and `'{'` represent the boundaries of the sea and shore parts of islands. In bounded seas, the expressions that recognize the shore parts immediately follow the bounded sea `~e~` represented by the $NEXT(e)$ function. In general, `~e~` is equivalent to:

```
(!next1 !next2 ...  .)* e (!next1 !next2 / ...  .)*
```

where next1, next2, ... are members of $NEXT(e)$. $NEXT(e)$ is similar to traditional $FOLLOW(X)$ for CFG. However, $NEXT(e)$ is different from $FOLLOW(X)$ in that $NEXT(e)$ contains not only terminal symbols but also nonterminal symbols, while $FOLLOW(X)$ contains only terminal symbols.

2.4.8 Tools for Editing Grammar

Another approach for reducing the effort to write a grammar is providing tools for editing the grammar. Parsify [61] and Parsimony [60] are grammar editors with a graphical user interface for non-expert users. The users can construct a grammar definition by interactively showing an example of how it is parsed. The user selects a text area on an editor screen and labels it with nonterminal. The editor automatically infers the rule for the nonterminal and shows possible parse trees. Typically, the user labels the text area in a bottom-up manner from leaves to the root of the parse tree. Until the user finishes teaching all the rules for the grammar, grammar is ambiguous. The grammar under editing generates a parse forest instead of a parse tree. To handle this ambiguity, Parsify uses CYK algorithm, and Parsimony uses GLR algorithm inside them, respectively. After teaching all the grammar rules, the user can get a functional parser that creates a unique parse tree from an input program.

2.5 Dynamic Domain-Specific Programming Assistance

We define dynamic programming assistance as support that is available while the user writes a program. They are usually provided by editors or IDEs, which programmers use to write a program. Assistance provided by modern IDEs includes text highlighting, auto-completion, and error checking. This section introduces existing approaches for providing dynamic domain-specific programming assistance in embedded DSLs.

2.5.1 Language Workbench

Language workbenches have been developed and used in many external DSLs. These language workbenches can generate tools for DSLs, including an IDE for the language. Xtext [30] and Spoofax [50] are grammar-based language workbenches that can support the development of IDEs. For example, Xtext generates a plugin for the Eclipse IDE from a grammar for a DSL. The generated plugin provides assistance such as code completion when the DSL user writes a program. Similarly, Spoofax also generates Eclipse plugins from declarative specifications for languages.

2.5.2 Customizing IDEs for the Host Languages

Customizing host IDEs is another option for better programming assistance in embedded DSLs. Nosal *et al.* [70] have reported their study on extending NetBeans IDE for non-programmers. They pointed out that programs written in embedded DSLs are too noisy or verbose for domain exports that are not programmers. They addressed this problem by extending an IDE for the host language with their plugin. Their plugin provides a way to hide the noisy part of the host language from the user.

2.5.3 Fluent API

A kind of domain-specific assistance is available in host IDEs by using fluent API. Fluent API is an API that the programmer uses in a method chaining. In a method chaining, methods are called one after another. Quoted example of a method chaining used in jOOQ[2] is as follows:

```
create.select(BOOK.TITLE)
.from(BOOK)
.where(BOOK.PUBLISHED_IN.eq(2011))
.orderBy(BOOK.TITLE)
```

This method chaining represents a SQL query for a relational database in an embedded DSL way. It is equivalent to the following SQL query:

```
SELECT TITLE
FROM BOOK
WHERE BOOK.PUBLISHED_IN = 2011
ORDER BY BOOK.TITLE
```

[2] https://www.jooq.org/

The method chining is regarded as a program in a mini-DSL embedded in the host language. The grammar of this mini-DSL is encoded in type definitions written in the host language. These definitions keep the user from writing an invalid program. For example, SQL allows `GROUP BY`, `HAVING`, `ORDER BY`, or `LIMIT` clauses after `WHERE` clause. Fluent API emulates this restriction with the type of value returned by the `where` method. The return type of the `where` method only implements `groupBy`, `having`, `orderBy`, `limit`. Therefore, if another method is specified by mistake, the error was detected before executing a program. If the user writes a method changing in an IDE for the host language, it can provide domain-specific code completion and error checking. For example, if the user types Ctrl and space after typing the call of the `where` method, the IDE will show the list of the methods which are allowed at the place. If the user happens to write an invalid method, the IDE will warn it immediately.

While fluent API is convenient for the user of the embedded DSL, implementing fluent API is not easy. The developer of fluent API has to write many complex type definitions to implement the fluent API. To mitigate this problem, Gil *et al.* [35, 36], Nakamaru *et al.* [69] and Yamazaki *et al.* [94] have proposed automated generation of a fluent API. These tools take the grammar of a fluent API as input and generate class definitions that implement the fluent API.

2.6 Motivation

Our motivation is to reduce the effort to implement both static and dynamic domain-specific assistance in embedded DSLs. Inadequate domain-specific assistance in embedded DSLs is because many additional efforts are needed to implement domain-specific assistance. While a benefit of embedded DSLs is their low implementation cost, these additional efforts reduce this benefit. Hence, reasonable ways to implement domain-specific assistance in embedded DSL are required.

2.6.1 Static Programming Assistance

The key to better static programming assistance in embedded DSLs is to improve the difficulty in writing island grammars for parsers. The development of a parser is the most laborious task in implementing syntax extension for programming assistance.

Island parsing is a promising technique to implement a parser for syntax extension. The island grammar does not contain the detailed parts of the language that are irrelevant to extended programming constructs. Therefore, the number of rules for parsing is relatively smaller than complete grammar for the language. For example, let us consider implementing the `unless` statement by syntax extension. An example of an unless statement handled by this syntax extension is as follows:

```
unless (x < 0) {
    doit();
}
```

This program is converted into the following program with the `if` statement:

```
if (!(x < 0)) {
    doit();
}
```

To implement this syntax extension, we are only interested in the leading part of the if statement, of which pattern is represented as the following pattern:

```
unless (<A>)
```

What we need is the `unless` keyword and the following condition part of the `unless` statement. In the above pattern, `<A>` represents the uninteresting part of the leading part of the unless statement. If we can extract the text matched by the above pattern, we can convert it into an equivalent code snippet as shown in the following pattern:

```
if (!(<A>))
```

An island grammar enables the DSL developer to write a grammar similar to the above pattern to extract the leading part of the `unless` statement. Island grammars consist of rules for *island* that is a programming construct of interest and *water* that is uninteresting parts of the language. Since the rules for *water* are not be fully specified, the number of the rule in island grammar is smaller than the complete grammar.

While the number of rules is small in an island grammar, it is not easy to define rules for water. Writing rules for water on the island is complicated, especially when the water appears inside islands. The language designers must write rules for water such that the parser does not consume a part of the island as water. For example, if we want to extract the leading part of the unless statement represented by the following pattern:

```
unless (<A>)
```

, we must define a rule for water to skip `<A>` such that it does not consume `)` at the end of the condition clause. The rule for water must specify that `)` should be excluded. Bounded seas partially reduce this difficulty in writing the rule for water by automatically calculating the $NEXT(e)$ of a bounded sea. In the case of the `unless` statement, `)` is automatically calculated as a member of $NEXT(e)$. Therefore, the user does not need to specify `)` by herself.

However, relying on $NEXT(e)$ limits the applicability of the sea operator. Let's assume we want to extract only C-like statements as shown in the language represented by the following PEG:

```
program <- stmt*
stmt <- block / expr_stmt
block <- '{' stmt* '}'
expr_stmt <- expr ';'
expr <- ...
```

While the details of `expr` is omitted in the above grammar, its definition requires many rules for subexpressions. Hence we do not want to specify the grammar fully. So we want to write the island grammar with the sea operator as follows.

```
program <- stmt*
stmt <- block / expr_stmt
block <- '{' stmt* '}'
expr_stmt <- ~~ ';'
expr <- ...
```

and it is equivalent to:

```
program <- stmt*
stmt <- block / expr_stmt
block <- '{' stmt* '}'
expr_stmt <- (! ';')* (! ';')* ';'
expr <- ...
```

The sea operators are converted into (! ';') in the above grammar. However, the above grammar cannot correctly parse the following program:

```
{ x = x + 1; } y = x;
```

After the parser recognizes `x = x + 1;` as `expr_stmt`, it recognizes the `} y = x;` as another `expr_stmt` by mistake. This is due to `(! ';' .)*` in the rule for `expr_stmt` consumes the `'}'` as water. An correct island grammar would be as follows:

```
program <- stmt*
stmt <- block / expr_stmt
block <- '{' stmt* '}'
expr_stmt <- (! ('}' / ';') .)* ';'
expr <- ...
```

The rules for `expr_stmt` contains not only `;` but also `}` as operands of the not predicate. This indicates that the sea operator cannot be used in the leading part of an ordered choice operator.

The problem of the bounded sea is that we cannot handle the leading part of the island as water. If the DSL developer is not interested in the leading part of the program construct, she must write a complicated rule for water to prevent the water from consuming a part of an island.

2.6.2 Dynamic Programming Assistance

The key to providing better dynamic domain-specific assistance in embedded DSLs is exploiting IDEs for the host language. If the host IDE can provide dynamic domain-specific assistance, the embedded DSL developer does not need to develop a specific IDE or plugin for an existing IDE for the host language. Developing these tools may not be easy for the embedded DSL developer since the host language is not the same as the one for developing these tools.

While the idea of exploiting the host IDE for the host language has been proposed, the idea is limited to method chaining. The embedded DSL developer can support dynamic-domain specific assistance in method chaining by implementing Fluet API. Fluent API is implemented in the host language. The DSL

Listing 2.1: MUL instruction in the reference manual (ARM)

```
1  if condition_passed()
2      d = UInt(Rd);
3      n = UInt(Rn);
4      m = UInt(Rm);
5      setflags = (S == '1')
6      operands1 = SInt(R[n])
7      operands2 = SInt(R[m])
8      result = operand1 * operand2
9      R[d] = result<31:0>
10     if setflags then
11        APSR.N = result<31>;
12        APSR.Z = IsZeroBit(result);
```

developer does not have to write a program in a language different from the host language.

Fluent API indicates that the design and implementation of the embedded DSL affect the availability of dynamic domain-specific assistance in the embedded DSL. Regardless of whether Fluent API is implemented or not, method chaining can be provided. By carefully designing types for method chaining as Fluent API, better dynamic programming assistance is available.

What we need to focus on is language design in terms of dynamic programming assistance. In general, the language design of the embedded DSL for better dynamic programming assistance has not been considered except Fluent API. Even with well-designed embedded DSL, there is room to improve its domain-specific assistance.

One successful application domain of embedded DSLs is processor description. DSLs for the processor description are called processor description language PDL. PDSL are designed for a task such as generating processor emulators or design space expropriation in developing a processor. Suppose that we describe the `MUL` instruction of the ARM processor in such an embedded DSL. The `MUL` instruction multiplies values in the source registers and stores the result in the destination register. Listing 2.1 is the pseudo-code expressing the semantics of the `MUL` instruction, which is quoted from the ARM reference manual [45]. In a Python-based embedded DSL, the Pydgin PDL [62], the same semantics is implemented by the `execute_mul` function shown in Listing 2.2. This function appears to be very similar to the pseudo-code in Listing 2.1. Thus, the user can describe the `execute_mul` function by mostly copying the pseudo-code in the reference manual. The user does not have to specify how to translate the `MUL` instruction into the host instructions when implementing the processor emulator in Pydgin.

Expressions `inst.cond`, `inst.rm`, and `inst.rn` in Listing 2.2 represent the values of the instruction fields of `MUL`, which are `cond`, `Rm`, and `Rn`, respectively. Figure 2.5 shows the excerpt of the bit encoding of the `MUL` instruction taken from the ARM reference manual. The `MUL` instruction has five instruction fields: `cond`, `S`, `Rd`, `Rm`, and `Rn`. Expressions `inst.cond`, `inst.rm`, and `inst.rn` are method calls on `inst`, which is a function parameter to `execute_mull`, and they return the values of those instruction fields.

Listing 2.2: MUL instruction in Pydgin (ARM)

```
1  def execute_mul( s, inst ):
2    if condition_passed( s, inst.cond ):
3      Rn, Rm = s.rf[ inst.rn ], s.rf[ inst.rm ]
4      result = trim_32(Rn * Rm)
5      s.rf[ inst.rd ] = result
6
7      if inst.S:
8        s.N = (result >> 31)&1
9        s.Z = result == 0
10
11     if inst.rd == 15:
12       return
13   s.rf[PC] = s.fetch_pc() + 4
```

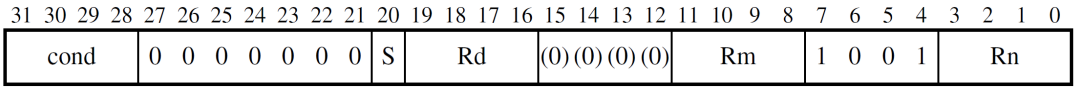

Figure 2.5: Encoding of the MUL instruction (ARM)

The type of `inst` is the `Instruction` class, which must be defined by the Pydgin user. Listing 2.3 is an example of the `Instruction` class for ARM, which is included in the source tree of Pydgin. We renamed the methods in the class such that the names are consistent with the field names in Figure 2.5. The methods in `Instruction` extract the value of an instruction field from the binary representation of the instruction. The binary representation is available from `self.bits`. For example, method `rd` extracts instruction field `Rd` in bits 16–19.

The Pydgin user can expect that the IDE for Python, which is the host language, reads the definition of the `Instruction` class and provides program-

Listing 2.3: Definition of the Instruction class

```
1  class Instruction( object ):
2    ...
3    @property
4    def rd( self ): return (self.bits >> 16) & 0xF
5
6    @property
7    def rm( self ): return (self.bits >> 8) & 0xF
8
9    @property
10   def rn( self ): return self.bits & 0xF
11   ...
12   @property
13   def imm24( self ): return self.bits & 0xFFFFFF
14   ...
```

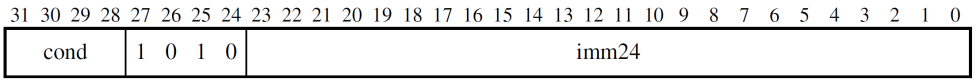

Figure 2.6: Encoding of the B instruction (ARM)

ming assistance to the user. For example, when the user writes `execute_mul`, auto-completion can be expected. When `inst.` is typed, a modern IDE would show the list of the word candidates that could follow `inst.`, and this list would include `cond`, `rm`, and `rn`.

However, this auto-completion lacks domain-specific assistance. Even if the IDE correctly infers that the type of `inst` is the `Instruction` class, the candidates for the auto-completion would include `imm24`, which is the method for extracting the `imm24` field of the `B` (branch) instruction shown in Figure 2.6. The reason is that this method is included in the `Instruction` class. In Pydgin, all of the methods for extracting an instruction field are included in the `Instruction` class. Although `imm24` is not available in the body of `execute_mul`, if the user selects `imm24` after typing `inst.`, the IDE would not warn the user of that incorrect selection. Note that the selection is valid from the host-language perspective of Python; it is invalid only from the domain-specific perspective of Pydgin.

To mitigate this problem, some readers might change the design of Pydgin to enable the user to define a different version of the `Instruction` class for a different instruction. For example, the type of `inst` passed to `execute_mul` could be `MulInstruction`, whereas that to `execute_b` could be `BInstruction`. Then, they could provide only the methods available for their instruction. However, this design requires the user to exert extra effort; numerous `Instruction` classes should be defined; thereby complicating the type inference by the IDE. If the host language were statically typed, that design would require the user to pay extra attention to the type of the `inst` parameter.

The dynamic domain-specific programming assistance in embedded DSLs is poorer than the assistance in external DSLs. If Pydgin were a standalone external DSL, it would report a compilation error when the user writes `inst.imm24` in the body of the `execute_mul` function. A dedicated IDE for Pydgin would provide better dynamic programming assistance to prevent writing `inst.imm24` mistakenly in the body of `execute_mul`.

Developing a dedicated IDE for an embedded DSL might be another option to mitigate the inadequate domain-specific assistance. Developing an external program-analysis tool might be another option. However, these options decrease the benefit of embedded DSLs; the user would not be able to use the DSL as a library for its host language. The user would be forced to write a program in a particular IDE, which may be unfamiliar.

Chapter 3

Lake Symbols

3.1 Introduction

Island parsing is a promising technique for the development of software engineering tools. Island parsers extract only programming constructs of interest as *islands* from an input string and they ignore the rest of the string as *water*. Such *incomplete* parsers are often easy to develop but still useful enough for a number of software engineering tools. These tools do not need the complete parse tree or the abstract syntax tree (AST) of the input program. They only need a parse tree or an AST for the interesting parts of the program. Suppose that we develop a tool for measuring the chromatics complexity [63] of programs. The tool only needs a parse tree for all the `if/else` statements and other control-flow statements. We can just ignore the rest of the program. An island parser fits this application; the statements like `if/else` are *islands* while the rest is *water*. An island parser is also suitable for source model extraction from incomplete program [67], multilingual parsing [87, 90, 2], extracting programming constructs from documentations [9, 81], and lightweight impact analysis [68].

The grammar for an island parser is described in a formal language such as PEG (Parsing Expression Grammar) [32], SDF (Syntax Definition Formalism) [44], or TXL [20, 19]. This grammar consists of the rules for islands and water. Describing this grammar is often easier than describing the corresponding full-featured grammar, which is for parsing a whole program. The rules for the water are usually simple, in an ideal case, just one wildcard character, and thus the grammar for island parsing consists of only a small number of rules. The island parser can be constructed from this grammar in the same way as ordinal parsers.

In practice, the grammar for island parsing consists of a small number of rules only when most parts of a program are recognized as water. To do so, some inner parts of the island must be also water, which is not fully parsed. For example, if an island is an `if/else` statement, the expressions included in that statement should be recognized as water. However, the rules for such water in the middle of an island are complicated and difficult to describe. Due to this difficulty, island parsing seems to be not widely used today despite its applicability to software engineering tools.

In this chapter, we propose the lake symbols to mitigate the difficulty in describing the rules for the water in the middle of an island. We call this water a *lake*. The lake symbols are newly added lexical elements in a PEG to represent

lakes in the PEG. A lake symbol can be used at the same place where nonterminal symbols can be specified in the PEG. A lake symbol is used as an operand of PEG's operators in the same way as nonterminal symbols. Lake symbols are surrounded with `<>` to be distinguished from nonterminal symbols. For example, `<expr>` is a lake symbol. A lake symbol can optionally have a rule whose left-hand side is the lake symbol in the same way as nonterminal symbols. Without an optional rule, the lake symbol can be regarded as a special wildcard that matches any single character except some parts of the island in/around that lake.

A lake symbol is typically used with a repeat operator such as `*` and `+`. We can use a lake symbol with a repat operator to find the *shortest* match of an arbitrary string between islands. It resembles `.+?` (one or more arbitrary characters with the lazy qualifier) in the regular expression. A lake symbol matches arbitrary character until the parser encounters a part or the whole of an island. For example, suppose that we want to extract statements terminated by a semicolon (;) as islands. The rule in PEG for the statement is described with a lake symbol as follows:

```
statement <- <e>+ ';'
```

In the above rule, `<e>` is a lake symbol. This rule describes that the `statement` non-terminal symbol is recognized if its right-hand side `<e>+ ';'` matches the input string. `<e>+ ';'` is one or more repetitions of the `<e>` lake symbol followed by `';'` and matches the shortest string that ends with a semicolon like `x = 1;`. A semicolon may alternatively appear in an input string instead of characters that `<e>` should match. `<e>` is not a naive wildcard like `.` in the regular expression. If we replace `<e>` with `.` in `<e>+`, the `.` happens to match `';'` since the `.` qualified with the one or more operator `+` matches a string greedy. For example, if the input string consists of multiple statements such as `x = 1; y = 1;`, `.+` matches `x = 1; y = 1`, excluding the last semicolon greedy according to the widely used semantics of regular expressions. As a result, `.+ ';'` matches two statements `x = 1; y = 1;` unintentionally. On the other hand, `<e>+ ';'` matches only one statement `x = 1;` or `y = 1;`. Therefore, `<e>+` is not equivalent to `.+`. Moreover, `<e>+` is not equivalent to the shortest match like `.+?` in the regular expression. `<e>+` does not match a part or the whole of an island. To explain this, we change the above rule as follows:

```
statement <- <e>+ ';' / block
```

where `/` is the ordered prioritized operator in the PEG. We assume that the `block` non-terminal symbol recognizes a string surrounded with `{}` such as `{ k = 1; }`. Now, `statement` recognizes `block` as well as the shortest string that ends with a semicolon. `<e>+` does not match the strings recognized by `block` since `block` is regarded as an island in the above rule. If we replace `<e>+` with `.+?`, it may match the strings recognized by `block` unintentionally. For example, `{ k = 1;` is recognized as `statement`, although it should have been recognized as a part of `block`. This results in failing to recognize `block` correctly. The parser needs to know what text patterns must not be matched by a lake symbol to exclude the parts of the island. These patterns are given by terminal or non-terminal symbols, which we call the *alternative symbols*. In the

above example, ';' and `block` are the *alternative symbols*. The lake symbol automates the enumeration of the alternative symbols so that it can be used as a wildcard-like symbol for the shortest match.

Although a lake symbol matches any character except alternative symbols by default, the user can explicitly specify the patterns that the lake symbol should match. The user can specify this by writing a rule whose left-hand side is the lake symbol and whose right-hand side represents the pattern to be matched. If there is a rule whose left-hand side is a lake symbol, the lake symbol behaves as a wildcard if the pattern on the right-hand side of the rule fails to match the input string. The pattern on the right-hand side is prioritized over alternative symbols. A lake symbol matches the input string if the pattern on the right-hand side matches the input string, even if one of the alternative symbols also matches the input string. We can define the rule for a lake symbol to create a lake containing its alternative symbols as (part of) its islands. For example, if the pattern for string literals is specified in the right-hand of the rule for a lake symbol, the lake symbol can match a string literal even if it contains an alternative symbol as part of it. The rule for a lake symbol also enables to handle nested islands inside the lake. We can put a pattern for nested islands on the right-hand side of the rule for the lake symbol.

A grammar in PEG with lake symbols can be translated into an equivalent one without lake symbols. We present an algorithm for this translation. The resulting grammar without lake symbols can be used as an input to an existing PEG-based parser or parser-generator. A grammar in PEG with lake symbols can be translated into an equivalent one without lake symbols. We present an algorithm for this translation. The resulting grammar without lake symbols can be used as an input to an existing PEG-based parser or parser-generator.

Furthermore, we implemented *PEGIsland*, an island-parser generator supporting our extended PEG with lake symbols. By using PEGIsland, we implemented 36 island parsers for Java programs and 20 ones for Python programs. We compared the number of their grammar rules with the number of the rules for the full-featured Java/Python parser. The comparison revealed that our extended PEG effectively reduces the number of the rules.

The contribution of this chapter is threefold:

- We propose the lake symbols and an extended PEG supporting the lake symbols. It mitigates the difficulty in describing the grammar rule for the water in the middle of an island. This contributes to reducing the total number of rules for an island parser and thereby improving the usefulness of the island parser.

- We introduce a novel semantics of the shortest match suitable for island parsing. Our shortest match with a lake symbol consumes an arbitrary input character until a part or the whole of an island appears.

- We implemented island parsers based on the extended PEG and revealed that the extended PEG effectively reduced the size of the grammar for island parsing.

In the rest of this chapter, we first present island parsing and our motivating problem. We then propose the lake symbols, an extension to PEG for supporting

Listing 3.1: The PEG for a simple language

```
1  program <- spacing (stmt / func_def)*
2  func_def <- FUNCTION ID LPAREN ID? RPAREN block
3  if_else_stmt <- if_stmt (ELSE stmt)?
4  if_stmt <- IF LPAREN expr RPAREN stmt
5  stmt <- block / if_else_stmt / exp_stmt
6  exp_stmt <- expr SEMICOLON
7  block <- LBRACE stmt* RBRACE
8  expr <- rel_expr (ASSIGN rel_expr)*
9  rel_expr <- add_expr ((EQ / GT / LT) add_expr)*
10 add_expr <- mul_expr ((PLUS / MINUS) mul_expr)*
11 mul_expr <- pri_expr ((MUL / DIV) pri_expr)*
12 pri_expr <- LPAREN expr RPAREN / lambda_expr / funcall /
      NUMBER / STRING / ID
13 lambda_expr <- LAMBDA LPAREN ID? RPAREN block
14 funcall <- ID LPAREN expr? RPAREN
15
16 spacing        <- [ \t\n]*
17 ID             <- [_a-z]+ spacing
18 LPAREN         <- '(' spacing
19 RPAREN         <- ')' spacing
20 LBRACE         <- '{' spacing
21 RBRACE         <- '}' spacing
22 SEMICOLON      <- ';' spacing
23 IF             <- 'if' spacing
24 ELSE           <- 'else' spacing
25 STRING         <-  '"' [^"]* '"' spacing
26 FUNCTION       <- 'function' spacing
27 LAMBDA         <- 'lambda' spacing
28 NUMBER         <- [0-9]+ spacing
29 ASSIGN         <- '=' spacing
30 EQ             <- '==' spacing
31 GT             <- '>' spacing
32 LT             <- '<' spacing
33 PLUS           <- '+' spacing
34 MINUS          <- '-' spacing
35 MUL            <- '*' spacing
36 DIV            <- '/' spacing
```

the lake symbols, and the algorithm for translating the extended PEG into the normal PEG. We also present our experiments and related work. Finally, we conclude this chapter.

3.2 Motivating Example

A parsing expression grammar (PEG) [32] is one of the formal grammars that are used for building a top-down parser (see Appendix A.1 for details on PEGs). Listing 3.1 shows an example of PEG. It specifies a simple language; its program consists of statements (`stmt`) and function definitions (`func_def`). A statement

Listing 3.2: A program in the language in Listing 3.1

```
1  function make_cmp_f(x)
2  {
3      if (x == 0)
4          func = lambda (y) {
5              if (y > x) {
6                  result = "positive";
7              }
8              else if (y < x) {
9                  result = "negative";
10             }
11             else {
12                 result = "zero";
13             }
14             result;
15         };
16     else
17         func = lambda (y) {
18             ...
19         };
20     func;
21 }
22
23 compare = make_cmp_f(0);
24 compare(1);
```

is either a block (`block`), an if/else statement (`if_else_stmt`), or an expression statement (`exp_stmt`). An expression statement is an expression (`expr`) with a semicolon at the end. An expression supports several binary operators such as `+` and `>` and its terms are parenthesized expressions, function calls, or number literals. Note that the terms may be a lambda expression (`lambda_expr`), which includes a block and the statements in that block.

The starting symbol of this grammar is `program`. The grammar is scannerless and its terminal symbols (or lexical tokens) are every character of its programs. `spacing` is a nonterminal symbol used for recognizing whitespace and skipping it. For example, the parsing expression for `program` starts with `spacing` for skipping the whitespace at the beginning of the program.

A program in Listing 3.2 is an example written in the language specified by Listing 3.1. Lines 1 to 21 are a function definition recognized by the nonterminal symbol `func_def`. Lines 3 to 19 are an if/else statement recognized by `if_else_stmt`. Lines 4 to 15 are a lambda expression that includes another if/else statement on lines 5 to 13. Lines 8 to 13 are also an if/else statement that is the *else* part of the if/else statement on lines 5 to 13.

Figure 3.1 is the parse tree obtained after the parser based on the grammar in Listing 3.1 parses the program in Listing 3.2. Each node of the tree is labeled with a nonterminal symbol. For example, the root of the tree is labeled with `program`. The details of the subtree labeled with `expr` is omitted and represented by a gray triangle in this figure.

Island parsing is a technique for recognizing only interesting program con-

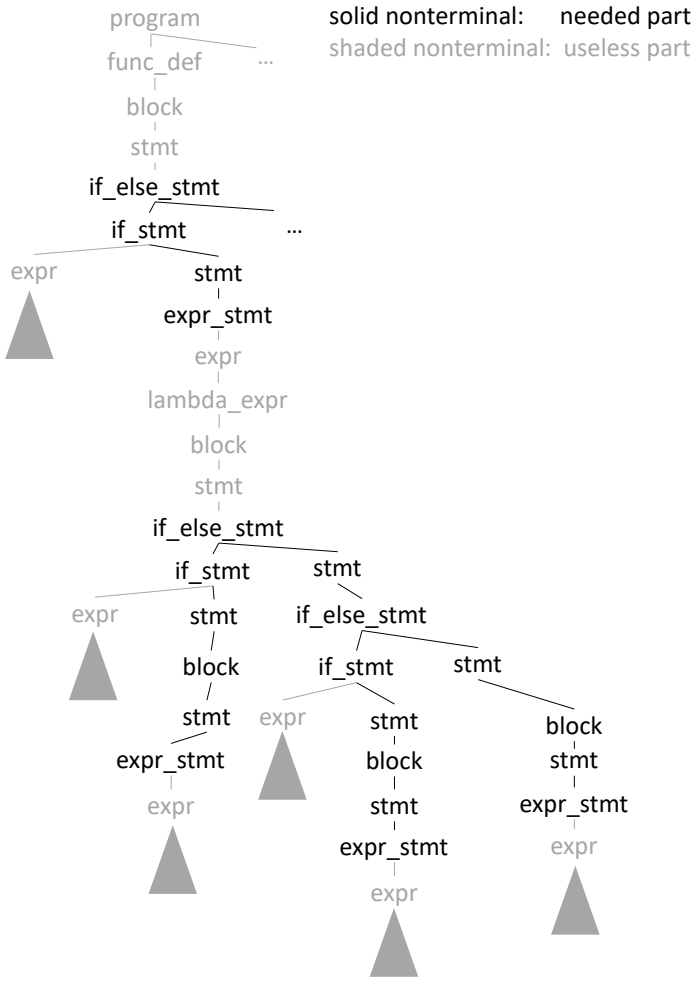

Figure 3.1: The parse tree for the program in Listing 3.2

structs such as an if/else statement in the given program. The benefit of island parsing is that the number of the rules in the grammar is reduced since we can eliminate the rules for the uninteresting constructs from the grammar. The program text of these uninteresting constructs is skipped by using a wildcard-like symbol. The interesting program constructs are called *islands* and recognized by the parser while the rest of the constructs are called *water* and skipped by the parser. Islands and water are also called *sea*.

Suppose that we are interested in only the if/else statements for the language in Listing 3.1 when we are developing a simple tool for that language. We assume that the tool rewrites if/else statements so that they will be written with curly braces. Coding conventions such as MISRA C [7] state that the control structures should be written with curly braces. In the case of the Listing 3.2, the tool rewrites the outermost if/else statement on lines 3 to 19 since it does not include braces for the if clause and the else clause. The parse tree that we would want to obtain does not need to contain the nodes or leaves that are irrelevant to the if/else statements. It would be the tree consisting of only the nodes labeled with a solid name in Figure 3.1. The gray nodes or the gray triangles would not be included in the tree.

An island parser that we need recognizes only an if/else statement as an

Listing 3.3: The grammar for the island parser

```
1  program <- spacing program_sea*
2  program_sea <- if_else_stmt / program_water
3  program_water <- STRING / .
4  if_else_stmt <- if_stmt (ELSE stmt)?
5  if_stmt <- IF LPAREN expr RPAREN stmt
6  stmt <- block / if_else_stmt / exp_stmt
7  exp_stmt <- expr SEMICOLON
8  block <- LBRACE stmt* RBRACE
9  expr <- expr_sea+
10 expr_sea <- if_else_stmt / expr_water
11 expr_water <- LPAREN expr_sea* RPAREN / block / STRING / !(
       SEMICOLON / RPAREN / RBRACE) .
12
13 spacing        <- [ \t\n]*
14 LPAREN         <- '(' spacing
15 RPAREN         <- ')' spacing
16 LBRACE         <- '{' spacing
17 RBRACE         <- '}' spacing
18 SEMICOLON      <- ';' spacing
19 IF             <- 'if' spacing
20 ELSE           <- 'else' spacing
21 STRING         <-  '"' [^"]* '"' spacing
```

island but skips the others as *water*. The grammar for that parser does not need to include the rule for `func_def`. It is obtained by eliminating the rule for `func_def` from Listing 3.1 and adding the following three rules:

```
program <- spacing program_sea*
program_sea <- if_else_stmt / program_water
program_water <- STRING / .
```

We introduce two new nonterminal symbols `program_sea` and `program_water`. A `program` is the repetition of `program_sea`, which is either `if_else_stmt` (*i.e.* an island) or `program_water`. `program_water` is a wildcard-like symbol. It recognizes either a string literal or any character matching a wildcard character ".". Note that the prioritized choice operator `/` gives a higher priority to `if_else_stmt` than `program_water`. Otherwise, `program_sea` would not recognize an `if_else_stmt` at all. The rule for `program_water` is not

```
program_water <- .
```

since this would let the parser recognize a string literal like `"if(x)y;"` wrongly as `if_else_stmt`.

Next, we remove `expr` from the grammar as well as `func_def` since the details of `expr` are not interesting. The grammar without `expr` and `func_def` is listed in Listing 3.3. The size of this grammar is approximately only two thirds of the original one in Listing 3.1. In this grammar, the original rule for `expr` is replaced with a simpler one. It is the repetition of `expr_sea`, and `expr_sea` is either `if_else_stmt` or `expr_water` as `program_sea` is.

Note that `expr_water` is not the same as `program_water` although both are *water* symbols, which are used as a wildcard matching any symbol.

```
expr_water <- LPAREN expr_sea* RPAREN / block / STRING / !(
   SEMICOLON / RPAREN / RBRACE) .
```

First, recall that an expression may be a lambda expression, which may include an if/else statement in its body. Thus, the parser must recognize an if/else statement as an island when it is included in the expression. `expr_water` must be aware of a parenthesized expression, a block, and a string literal since they may include an if/else statement. On the other hand, `program_water` must be aware of only a string literal.

Second, the last option of the prioritized choice `/` in `expr_water` is not only a wildcard character `.` but:

```
!(SEMICOLON / RPAREN / RBRACE) .
```

The wildcard character follows a *not* predicate (negative lookahead). This reads as matching any character except a semicolon, a right parenthesis, and a right brace. Since they are parts of an island, they should not be recognized as a part of water. They are sentinels for terminating the repetition of a wildcard. We call these symbols as *alternative symbols* in this dissertation. Note that the wildcard character in the rule for `program_water` does not follow a *not* predicate.

Since `expr` appears in the parsing expressions (the right-hand side of `<-` for the rule) for other nonterminals such as `if_stmt` and `exp_stmt`, its water symbol `expr_water` may not match the symbols following `expr` in those parsing expressions. These symbols are alternative symbols for `expr_water`.

For example, the parsing expression for `exp_stmt` is:

```
expr SEMICOLON
```

Since `expr` matches the repetition of `if_else_stmt` or `expr_water`, `expr_water` may not match a semicolon. Note that the PEG-based parser can be regarded as the parser always finding the longest match. Unless a semicolon is excluded from the symbols that `expr_water` matches, a semicolon would be absorbed into `expr` and thus not recognized as the last token of `expr_stmt`.

The alternative symbols for `expr_water` are not only the symbols following `expr`. They are also the symbols that may appear at the grammatical position where `expr_water` may appear. In other words, the alternative symbols include the symbol that the parser must consume as part of a different non-terminal when the parser can consume it as `expr_water`. Thus, the idea of the alternative symbols are different from the follow set used in the LALR(1) parsing.

An example of such an alternative symbol is `RBRACE` for `expr_water`. Note that `RBRACE` does not follow `expr`. However, `RBRACE` is included in the alternative symbols for `expr_water`. The parser must consume it as a part of block when the parser can consume it as `expr_water`. Suppose that the parser has read the following character/token sequence:

```
if (x < 0) { x = 1;
```

The parser recognizes this as

```
IF LPAREN expr RPAREN LBRACE stmt
```

The parser will next attempt to recognize another occurrence of `stmt`. It will try `block`, `if_else_stmt`, and finally `expr_stmt`. If the next token is `}`, that is, the parser reads:

```
if (x < 0) { x = 1; }
```

then the parser does not recognize `}` (`RBRACE`) as the first token for `block` or `if_else_stmt`. So the parser next attempts `expr_stmt` but this attempt must fail. The parser must fail to recognize `}` as `expr_water`. It must recognize `}` as the last token for `block` instead of `expr_water`. This is why `RBRACE` is included in the alternative symbols for `expr_water`.

Island parsing is a useful technique when we only need a parser that recognizes only interesting program constructs, such as an if/else statement, in the given program. The size of the grammar for island parsing is often smaller than that of the normal grammar. This fact reduces the development cost of the parser. Furthermore, an island parser can recognize program constructs even when a syntax error is included in the program but in the program constructs that the parser does not recognize but it skips.

However, implementing the grammar definition for island parsing is not easy. As we showed above, the rules for *water* symbols tend to be complicated and error-prone. For example, selecting a right set of the alternative symbols is not easy. This difficulty is a burden to the developers when implementing an island parser.

3.3 Lake Symbol

To reduce the development cost of island parsers, we propose an extension to Parsing Expression Grammar (PEG). It supports island parsing by a new kind of symbol named a *lake symbol*.

An island parser parses a program as the repetition of *sea*. *sea* is either an *island* or *water*.

```
program <- sea*
sea <- island / water
```

An island is a symbol representing the program constructs that we want to parse, while water is a symbol representing the other program constructs. Since we are not interested in the latter constructs or want to parse them, the naive definition of the water is a wildcard that anything matches.

As we have seen in the previous section, the definition of the water becomes complicated when we are not interested in inner parts of the island. For example, when we were not interested in the expressions included in an if/else statement, we had to define the water that the expressions match, and that definition was complicated.

We call such uninteresting inner parts *lakes* because they are *water* in an *island*. As the *sea* consists of islands and water, the *lake* consists of islands and water. Thus, there may be an island in the middle of the lake. The lake is a kind of the sea but it is an inner part of an island. Our lake symbol helps us define the lake. In particular, it automates the calculation of the alternate symbols.

Our *lake* symbol is a grammatical symbol that has the following properties:

1. It is always enclosed in angle brackets <>, unlike nonterminal symbols.
2. It can be used on the right-hand side of any rules in the same way as other grammatical symbols. It matches any character as a wildcard symbol for shortest matching.

3. It can optionally have a rule whose left-hand side is itself and whose right-hand side is any parsing expression, in the same way as nonterminal symbols. It matches the input string that the parsing expression on the right-hand side matches. It works as a wildcard matching any single character only when it has no rule or the right-hand side of its rule does not match the input string.

For example, `<STRING_lake>` is a lake symbol. It can be used on the right-hand side of any rule. Suppose that a string literal `STRING` is an island. `<STRING_lake>` is used as follows:

```
STRING <- '"' <STRING_lake>* '"' spacing
```

Here, `<STRING_lake>` is used as part of the parsing expression on the right-hand side of the rule. It represents a lake in the string literal. We do not have to write the rule for defining this lake symbol. The lake symbol can be regarded as a wildcard character that any character except " matches since the lake symbol is a wildcard for shortest matching. So the parsing expression `<STRING_lake>*` matches any text enclosed in double quotes ". The excluded character " is automatically detected; this is a difference from the wildcard for the regular expressions. " is an alternative symbol of this lake. A lake symbol with a repetition operator can be considered as a representation of the *shortest* match of an arbitrary string followed by an alternative symbol.

A lake symbol is not a simple wildcard. It can have a rule whose left-hand side is itself. If a lake symbol has a rule, a lake symbol matches the input string that the right-hand side of that rule matches in the same way as nonterminal symbols. The lake symbol works as a wildcard matching any single character only when the right-hand side does not match the input string. We can specify islands in the lake by defining a rule for the lake symbol. Suppose that a `block` is an island.

```
block <- '{' <block_lake>* '}'
<block_lake> <- block
```

Here, `<block_lake>` is a lake symbol. The second line specifies that a `block` enclosed in another `block` is an island. If there are multiple kinds of islands in the lake, they are enumerated by using the prioritized choice operator `/`.

The rule for a lake symbol can be recursive. For example,

```
block <- '{' <block_lake>* '}'
<block_lake> <- '{' <block_lake>* '}'
```

In the second rule, `<block_lake>` is recursively referred to in the parsing expression on the right-hand side of `->`. This reads as the parser considers the nested curly braces, but it does not parse an inner block as an island. It parses only the outermost block as an island. This rule is necessary to identify which closing brace is balanced to an opening brace, even if we are not interested in inner blocks.

In our extended PEG, `water` is a special symbol. The parsing expression for `water` is appended to the parsing expressions for all the *lake* symbols. For example, if the rule for `water` is:

```
water <- COMMENT / STRING
```

then the following rule for a lake:

Listing 3.4: The grammar written with lake symbols

```
1  program <- spacing program_sea*
2  program_sea <- if_else_stmt / water
3  water <- STRING / .
4  if_else_stmt <- if_stmt (ELSE stmt)?
5  if_stmt <- IF LPAREN expr RPAREN stmt
6  stmt <- block / if_else_stmt / exp_stmt
7  exp_stmt <- expr SEMICOLON
8  block <- LBRACE stmt* RBRACE
9  expr <- <expr_lake>*
10 <expr_lake> <- if_else_stmt / LPAREN <expr_lake>* RPAREN /
       block
11
12 spacing        <- [ \t\n]*
13 LPAREN         <- '(' spacing
14 RPAREN         <- ')' spacing
15 LBRACE         <- '{' spacing
16 RBRACE         <- '}' spacing
17 SEMICOLON      <- ';' spacing
18 IF             <- 'if' spacing
19 ELSE           <- 'else' spacing
20 STRING         <-  '"' <STRING_lake>* '"' spacing
```

```
<block_lake> <- block
```

is treated as an equivalent of this:

```
<block_lake> <- block / COMMENT / STRING
```

Note that `water` is also used for the top-level rule `program`. Listing 3.4 shows the grammar definition written with lake symbols `expr_lake` and `STRING_lake`. It is equivalent to Listing 3.3 except the use of lake symbols. `water` is used in line 2 and defined in line 3 in that grammar definition.

We can generate an island parser from our extended PEG. We first translate the extended PEG into a normal PEG and then use this normal PEG to generate a parser based on packrat parsing [31] or its variants [28]. In the following subsections, we present how our extended PEG is translated into a normal PEG.

3.3.1 Translation into a Normal PEG

Our extended PEG with lake symbols can be represented by a tuple $G = (V_N, V_L, V_T, R, e_s, e_w)$, where V_N is a set of nonterminal symbols, V_L is a set of lake symbols, V_T is a set of terminal symbols, R is a set of rules, e_s is a starting expression, e_w is a global water expression, $V_N \cap V_L = \emptyset$, $V_N \cap V_T = \emptyset$, $V_L \cap V_T = \emptyset$. The extended elements of the tuple is V_L and e_w. V_L is a set of symbols enclosed in angle brackets `<>` and e_w is given by the rule for the special symbol `water`. If `water` is not explicitly given in the grammar, e_w is the regular expression `!.` (*i.e.* nothing matches e_w).

We translate an extended PEG G into a normal PEG G''. Here, $G'' = (V'_N, V_T, R'', e_s)$, such that $V'_N = V_N \cup V_L$. The translation is divided into two steps. In the following, X, $<X>$ and e are meta-variables. X ranges over

Listing 3.5: The syntax of parsing expressions

```
parsing_expr ::= parsing_expr parsing_expr | parsing_expr
   '/' parsing_expr
      | parsing_expr '*' | parsing_expr '+' | parsing_expr
        '?'
      | '&' parsing_expr | '!' parsing_expr
      | grammar_symbol
```

nonterminal symbols V_N, $<X>$ ranges over lake symbols V_L, and e ranges over a parsing expression.

Step 1

We translate an extended PEG G into an intermediate PEG $G' = (V_N, V_L, V_T, R', e_s)$. In G', the global water expression e_w is appended to the parsing expression of the rule for every lake symbol in V_L.

For each rule in R, if the rule is $X \leftarrow e$, then R' includes the rule $X \leftarrow e$ as it is. If the rule is $<X> \leftarrow e$, then R' includes the following rule:

$$<X> \leftarrow e/e_w$$

For each lake symbol $<X>$ in V_L, if R does not include a rule for $<X>$, $<X> \leftarrow e$, then R' includes the following rule:

$$<X> \leftarrow e_w$$

Step 2

For each rule in R', if the rule is $X \leftarrow e$, then R'' includes the rule $X \leftarrow e$ as it is. If the rule is $<X> \leftarrow e$, then R'' includes the following rule:

$$<X> \leftarrow e/!(s_1/.../s_n)\ .$$

such that $s_i \in V_N \cup V_T$ and $s_1, s_2, ..., s_n$ are the elements of $ALT(e)$, the set of the alternative symbols for e, which is the definition of $<X>$ in R'. This rule reads as $<X>$ is e or any character that is not the first part of the text string recognized by $s_1, s_2, ..., s_n$. Note that the last period in the rule is a wildcard character.

3.3.2 Alternative Symbols

$ALT(e)$ is the set of the alternative symbols for e. The elements of $ALT(e)$ are grammar symbols, which are either a terminal, nonterminal, or lake symbol.

We below use meta-variables e_i, e_j, ... ranging over the parsing expressions in R'. e_i, e_j, ... are location-aware. Location-aware means that two lexically equivalent parsing expressions e_i and e_j are not identical when they belong to a different rule or they are different sub-expressions in the same expression. A parsing expression `parsing_expr` is defined as in Listing 3.5. We assume that the sequence operator and the prioritized choice operator have left associativity.

Suppose that we have a grammar written in our extended PEG:

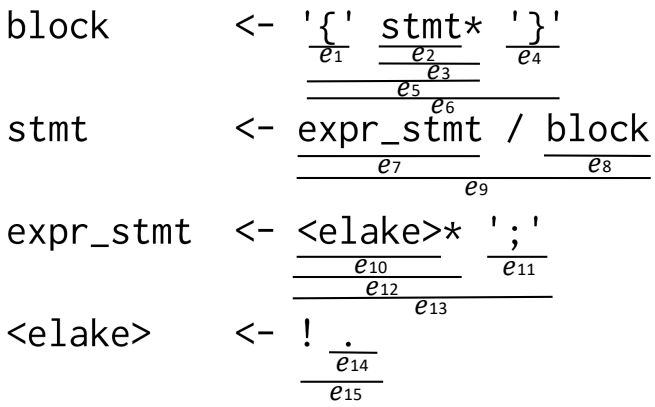

Figure 3.2: The parsing expressions in the grammar

```
block       <- '{' stmt* '}'
stmt        <- expr_stmt / block
expr_stmt   <- <elake>* ';'
```

The step 1 presented in Section 3.3.1 translates this PEG into the intermediate PEG shown in Figure 3.2. All the parsing expressions included in this intermediate PEG are also presented. There are 15 parsing expressions. We label 1 to 15 for each parsing expression. The domain of ALT for this grammar is these parsing expressions e_1 to e_{15}.

Definition of ALT

We define $ALT(e_i)$ as the set of the uppermost symbols that the parser may recognize immediately after it fails to recognize e_i. When the parser recognizes a non-terminal symbol, it also recognizes its lower-level symbols. We discard these lower-level symbols when calculating $ALT(e_i)$. We use the fixed-point iteration algorithm shown in Algorithm 1 to calculate $ALT(e_i)$ for all e_i in the intermediate PEG. The constraints satisfied by the fixed-point is shown in Appendix A.2.1. The input to Algorithm 1 is the rules R' of the intermediate PEG G' obtained at the step 1. Algorithm 1 also takes a set E as the input. E contains all the parsing expressions appearing in the rules R'.

The fixed-point iteration terminates after a finite number of iterations since the size of every $ALT(e_i)$ grows monotonically but it is smaller than $|V_N| + |V_L| + |V_T|$. In the worst case, every iteration adds only one element to only one instance of ALT. This case may cause the maximum number of iteration, which is still lower than $|E| \cdot (|V_N| + |V_L| + |V_T|)$.

Algorithm 1 uses two functions $BEGINNING(e_i)$ and $SUCCEED(e_i)$. $BEGINNING$ and $SUCCEED$ are similar to $FIRST$ and $FOLLOW$ for context-free grammars. The difference is that $BEGINNING$ and $SUCCEED$ contain not only terminal symbols but also non-terminal symbols. This difference is essential when a grammar is written in PEG, in which a terminal symbol is just a letter. $BEGINNING(e_i)$ is the set of grammar symbols that the parser may first recognize when it starts recognizing the parsing expression e_i. When e_i recognizes an empty input, $BEGINNING(e_i)$ includes ϵ. $SUCCEED(e_i)$ is a set of grammar symbols that the parser may next recognize after it recognizes e_i.

Algorithm 1 Calculation of $ALT(e_i)$

Input: a set of all the parsing expressions E included in the rules R'
Output: $ALT(e_i)$ includes the alternative symbols for the parsing expression e_i

```
 1: for all parsing expressions e_i in E do
 2:     ALT(e_i) ← ∅
 3: while ALT(e_i) is changing for some e_i in E do
 4:     for all parsing expressions e_i in E do
 5:         if e_i is a terminal symbol then
 6:             do nothing
 7:         else if e_i is a nonterminal or lake symbol 𝒮 and 𝒮 ← e_j ∈ R' then
 8:             ALT(e_j) ← ALT(e_j) ∪ ALT(e_i)
 9:         else if e_i is e_j*, e_j+, or e_j? then
10:             ALT(e_j) ← ALT(e_i) ∪ SUCCEED(e_i)
11:         else if e_i is !e_j then
12:             ALT(e_j) ← SUCCEED(e_i)
13:         else if e_i is &e_j then
14:             ALT(e_j) ← ALT(e_i)
15:         else if e_i is e_j/e_k then
16:             ALT(e_k) ← ALT(e_i)
17:             if ϵ ∈ BEGINNING(e_k) then
18:                 ALT(e_j) ← ALT(e_i) ∪ (BEGINNING(e_k) − {ϵ}) ∪ SUCCEED(e_k)
19:             else
20:                 ALT(e_j) ← ALT(e_i) ∪ BEGINNING(e_k)
21:         else if e_i is e_j e_k then
22:             ALT(e_j) ← ALT(e_i)
23:             if ϵ ∈ BEGINNING(e_j) then
24:                 ALT(e_k) ← ALT(e_i)
25:             else
26:                 ALT(e_k) ← ∅
```

The two functions are calculated by Algorithm 2 and Algorithm 3, respectively. They use fixed-point iteration. Each fixed-point satisfies the constraints shown in Appendix A.2.2 and A.2.3.

The algorithm calculates $ALT(e_i)$ for every parsing expression e_i in E, which is the set of all the parsing expressions included in the rules R'. For every e_i, the algorithm first sets $ALT(e_i)$ to an empty set. Then the algorithm incrementally updates $ALT(e_i)$ until $ALT(e_i)$ does not change for any e_i in E. The update is performed according to line 5 to 26 for every e_i in E. For example, when e_i is e_2 in Figure 3.2, $ALT(e_9)$ is updated to be $ALT(e_9) \cup ALT(e_2)$ according to line 8. Since e_2 is a nonterminal symbol `stmt` and R' includes the rule `stmt <-` e_9, $ALT(e_2)$ is added to $ALT(e_9)$. When e_i is e_9, $ALT(e_8)$ is updated to be $ALT(e_9)$ according to line 16. $ALT(e_7)$ is updated to be $ALT(e_7) \cup BEGINNING(e_8)$ according to line 20 because $BEGINNING(e_8)$ does not include ϵ; e_8 does not recognize an empty input.

3.3.3 Example

In this subsection, we show the calculation of ALT for the parsing expressions in Figure 3.2. Before doing so, we first show that `block`, '`;`', and '`}`' are the alternative symbols for `<elake>`, which is $ALT(e_{15})$ by using a railroad diagram. The railroad diagram is useful to visualize alternative symbols.

Figure 3.3 is the railroad diagram for the nonterminal symbol `block`. `stmt` in

Algorithm 2 Calculation of $BEGINNING(e_i)$

Input: a set of all the parsing expressions E included in the rules R'
Output: $BEGINNING(e_i)$ includes the grammar symbols that the parser may first recognize when it starts recognizing e_i.

```
1: for all parsing expressions e_i in E do
2:     BEGINNING(e_i) ← ∅
3: while BEGINNING(e_i) is changing for some e_i in E do
4:     for all parsing expressions e_i in E do
5:         if e_i is a terminal symbol α then
6:             BEGINNING(e_i) ← {α}
7:         else if e_i is a nonterminal or lake symbol 𝒮 and 𝒮 ← e_j ∈ R' then
8:             BEGINNING(e_i) ← {𝒮}
9:         else if e_i is e_j? or e_j* then
10:            BEGINNING(e_i) ← BEGINNING(e_j) ∪ {ϵ}
11:        else if e_i is e_j+ then
12:            BEGINNING(e_i) ← BEGINNING(e_j)
13:        else if e_i is !e_j or &e_j then
14:            BEGINNING(e_i) ← {ϵ}
15:        else if e_i is e_j/e_k then
16:            BEGINNING(e_i) ← BEGINNING(e_j) ∪ BEGINNING(e_k)
17:        else if e_i is e_j e_k then
18:            if ϵ ∈ BEGINNING(e_j) then
19:                BEGINNING(e_i) ← (BEGINNING(e_j) − {ϵ}) ∪ BEGINNING(e_k)
20:            else
21:                BEGINNING(e_i) ← BEGINNING(e_j)
```

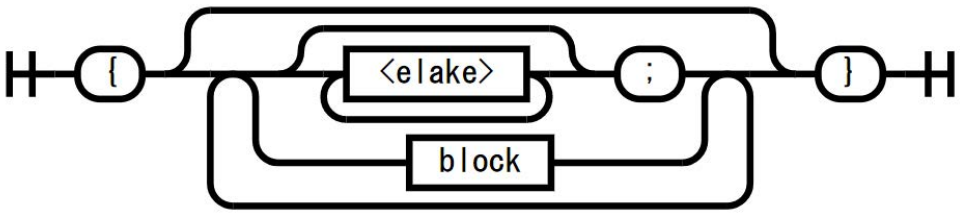

Figure 3.3: Railroad diagram

the parsing expression for `block` has been expanded into its definition. `expr_stmt` has been also expanded. In Figure 3.3, round boxes depict terminal symbols while square boxes depict nonterminal or lake symbols.

The alternative symbols for `<elake>` are what the parser may recognize when it can next recognize `<elake>`. These symbols are identified by looking at the railroad diagram in Figure 3.3. According to this diagram, the parser recognizes `<elake>` after the grammar symbols '`{`', '`;`', and `block`. In other words, when the parser recognizes these symbols, it can next recognize `<elake>`. We can see that the parser can next recognize `block`, '`;`', and '`}`' as well as `<elake>` when it recognizes '`{`', '`;`', or `block`. Therefore, the symbols `block`, '`;`', and '`}`' are the alternative symbols for `<elake>`. The alternative symbols are the destination symbols reached from the source symbols that we can also reach `<elake>` from.

Table 3.1 illustrates how Algorithm 1 calculates $ALT(e_i)$ for every parsing expression e_i in Figure 3.2. To calculate the fixed point, the algorithm iterates five times. Each row represents a parsing expression e_1 to e_{15}. $BEGINNING(e_i)$ and $SUCCEED(e_i)$ are also shown for each e_i. Each iteration examines a parsing

Algorithm 3 Calculation of $SUCCEED(e_i)$

Input: a set of all the parsing expressions E included in the rules R'
Output: $SUCCEED(e_i)$ includes the grammar symbols that the parser may next recognize after e_i.

```
1: for all parsing expressions e_i in E do
2:     SUCCEED(e_i) ← ∅
3: while SUCCEED(e_i) is changing for some e_i in E do
4:     for all parsing expressions e_i in E do
5:         if e_i is a terminal symbol then
6:             do nothing
7:         else if e_i is a nonterminal or lake symbol 𝒮 and 𝒮 ← e_j ∈ R' then
8:             SUCCEED(e_j) ← SUCCEED(e_i) ∪ SUCCEED(e_j)
9:         else if e_i is e_j? then
10:            SUCCEED(e_j) ← SUCCEED(e_i)
11:        else if e_i is e_j* or e_j+ then
12:            SUCCEED(e_j) ← SUCCEED(e_i) ∪ BEGINNING(e_i) − {ϵ}
13:        else if e_i is !e_j or &e_j then
14:            SUCCEED(e_j) ← ∅
15:        else if e_i is e_j/e_k then
16:            SUCCEED(e_j) ← SUCCEED(e_i)
17:            SUCCEED(e_k) ← SUCCEED(e_i)
18:        else if e_i is e_j e_k then
19:            SUCCEED(e_k) ← SUCCEED(e_i)
20:            if ϵ ∈ BEGINNING(e_k) then
21:                SUCCEED(e_j) ← (BEGINNING(e_k) − {ϵ}) ∪ SUCCEED(e_k)
22:            else
23:                SUCCEED(e_j) ← BEGINNING(e_k)
```

expression in the order from e_1 to e_{15}.

In the first iteration, for example, when the algorithm processes e_3, it updates $ALT(e_2)$ since e_3 is a zero-or-more expression e_2*. $ALT(e_3)$ is empty but $SUCCEED(e_3) = \{\text{'}\texttt{}\text{'}\}$. Thus, the updated $ALT(e_2)$ is $\{\text{'}\texttt{}\text{'}\}$. When the algorithm processes e_9, it updates $ALT(e_7)$ since e_9 is a prioritized-choice expression e_7/e_8. $ALT(e_9)$ is empty in this iteration and $BEGINNING(e_8) = \{block\}$. Thus, the updated $ALT(e_7)$ is $\{block\}$.

In the second iteration, when the algorithm processes e_2, it updates $ALT(e_9)$. Since e_2 is a nonterminal symbol `stmt` and its rule is `stmt` $\leftarrow e_9$, $ALT(e_2) = \{\text{'}\texttt{}\text{'}\}$ is added to $ALT(e_9)$. Thus, it updates $ALT(e_9)$ into $\{\text{'}\texttt{}\text{'}\}$. Then, when the algorithm processes e_9, it updates $ALT(e_7)$ again. Since $ALT(e_9)$ is already $\{\text{'}\texttt{}\text{'}\}$ at this time, $ALT(e_7)$ becomes $\{block, \text{'}\texttt{}\text{'}\}$.

In the final iteration, when the algorithm processes e_{10}, it updates e_{15} into $ALT(e_{15}) \cup ALT(e_{10}) = \{\text{';'}, \texttt{block}, \text{'}\texttt{}\text{'}\}$. This is what we need as the alternative symbols for `<elake>`.

Since we have obtained $ALT(e_{15})$, we can apply Step 2 in Section 3.3.1 to the intermediate grammar shown in Figure 3.2. The rule for `<elake>`:

```
<elake> <- !.
```

in the intermediate PEG is translated into the following rule:

```
<elake> <- !. / !(block / ';' / '}') .
```

in the normal PEG. This rule is semantically equivalent to the following rule:

```
<elake> <- !(block / ';' / '}') .
```

Table 3.1: $ALT(e_i)$ for each iteration

e_i	operator	$BEGINNING(e_i)$	$SUCCEED(e_i)$	$ALT(e_i)$ for each iteration				
				1	2	3	4	5
e_1	Terminal {	{'{'}	{stmt, '}'}	∅	∅	∅	∅	{'}'}
e_2	Nonterminal stmt	{stmt}	{stmt, '}'}	{'}'}	{'}'}	{'}'}	{'}'}	{'}'}
e_3	Zero-or-more e_2*	{ϵ, stmt}	{'}'}	∅	∅	∅	∅	∅
e_4	Sequence e_1e_3	{'{'}	{'}'}	∅	∅	∅	{'}'}	{'}'}
e_5	Terminal }	{'}'}	{stmt, '}'}	∅	∅	∅	∅	∅
e_6	Sequence e_4e_5	{'{'}	{stmt, '}'}	∅	∅	{'}'}	{'}'}	{'}'}
e_7	Nonterminal expr_stmt	{expr_stmt}	{stmt, '}'}	{block}	{block, '}'}	{block, '}'}	{block, '}'}	{block, '}'}
e_8	Nonterminal block	{block}	{stmt, '}'}	∅	{'}'}	{'}'}	{'}'}	{'}'}
e_9	Prioritized choice e_7/e_8	{expr_stmt, block}	{stmt, '}'}	∅	{'}'}	{'}'}	{'}'}	{'}'}
e_{10}	Lake <elake>	{<elake>, ϵ}	{<elake>, ';'}	{';'}	{';'}	{';', block}	{';', block, '}'}	{';', block, '}'}
e_{11}	Zero-or-more e_{10}*	{ϵ, <elake>}	{';'}	∅	{block}	{block, '}'}	{block, '}'}	{block, '}'}
e_{12}	Terminal ;	{';'}	{stmt, '}'}	∅	{block}	{block, '}'}	{block, '}'}	{block, '}'}
e_{13}	Sequence $e_{11}e_{12}$	{<elake>, ';'}	{stmt, '}'}	∅	{block}	{block, '}'}	{block, '}'}	{block, '}'}
e_{14}	Terminal .	{'.'}	∅	∅	∅	∅	∅	∅
e_{15}	Not !e_{14}	{ϵ}	{<elake>, ';'}	∅	{';'}	{';'}	{';', block}	{';', block, '}'}

since `!.` does not match any input. Note that `block`, ';', and '}' are the alternative symbols for `<elake>`, which are the elements of $ALT(e_{15})$.

3.3.4 Limitations

The island parser looks ahead into the remaining text and attempts to match alternative symbols to check if the next token is water. The number of lookahead tokens is not limited and depends on what the alternative symbols are. However, at least one or more lookahead token is needed. In other words, our algorithm does not support the case that the alternate symbol for a lake symbol recognizes an empty input ϵ. For example,

```
stmt <- expr ';'
expr <- <term> opt
opt <- '++'?
```

The `++` operator is optional. The alternate symbols for `<term>` are `opt` and ';' and the rule for `<term>` is:

```
<term> <- !(opt / ';') .
```

Since `opt` can recognize ϵ, this *not*-predicate always fails. Thus, the lake symbol `<term>` could not recognize any symbol; it could not work as a wildcard-like symbol.

Our prototype system of the island-parser generator detects whether a lake symbol has ϵ as its alternative symbol. If our prototype detects such a lake symbol, it prints a warning message so that the user can modify the grammar. Our system reports that ϵ is the alternative symbol for a lake symbol `<X>` if there exists $Y \in ALT(<X>)$ such that $Y \leftarrow e_i$ and $\epsilon \in BEGINNING(e_i)$.

3.4 Experiments

We have implemented our prototype system *PEGIsland* in Python. PEGIsland is an island parser. It reads a grammar definition written in our extended PEG and then parses a given source program to build a parse tree based on that grammar. It performs packrat parsing [31], based on the internally generated normal PEG from the given grammar. The generated normal PEG can be optionally written out to a file. The parse tree is written out in the JSON format. Their intermediate nodes correspond to nonterminal symbols or lake symbols in the grammar. Their leaves correspond to terminal symbols.

We conducted experiments using PEGIsland to answer our research question, to what extent the lake symbols reduce the number of grammar rules for parsers. We implemented island parsers for Java and Python and compared the size of their grammars with the size of the full-featured grammars.

We also compared the size of the grammars written for island parsers with lakes and without lakes. Although our lake symbols help the definition of lakes, an island parser can be implemented without lakes. As we showed in Section 3.2, writing a grammar without lakes is not difficult. It was easy when we only defined `program_sea` and `program_water`. It became difficult when we defined `expr_sea` and `expr_water` for a lake so that we could omit the definition of `expr` appearing in the middle of the island, `if_else_stmt`. We examined whether or not the use of the lakes effectively affected the number of grammar rules for island parsers. This was our second research question.

3.4.1 Java

For the full-featured Java grammar, we chose the grammar definition of an open-source PEG parser *MOUSE* [79]. It is based on the Java Language Specification for Java SE 8 Edition, with the corrections by the MOUSE developer to make it compatible with *javac*. The grammar consists of 279 PEG rules.

We implemented 36 island parsers on top of our PEGIsland parser. For each parser, we implemented two versions with lakes and without lakes. As an island, each of these parsers recognizes one of the nonterminal symbols in the full-featured grammar. According to the source-code comments, that grammar includes the nonterminal symbols corresponding to a program construct described in the dedicated section of the Java Language Specification [39]. We chose these nonterminal symbols as the islands. We implemented the island parsers so that the number of their grammar rules would be as small as possible. We used lake symbols to avoid defining irrelevant nonterminal symbols as far as the parser could correctly recognize an island. To verify that those island parsers correctly recognized their islands, we ran the island parsers to parse all the 7695 Java source files distributed as part of the Java 1.8 SDK. We then confirmed that these island parsers recognized the same number of islands as the number of the corresponding nonterminals that the full-featured Java parser recognized.

Table 3.2 lists the results. Each row represents the result of each island parser. For example, the second row represents that the parser recognizes `PackageDeclaration` as an island and it is described in Section 7.4 of the Java Language Specification (JLS). The number of the rules for its grammar is 22 and this grammar includes two lake symbols and three alternative symbols. #

Table 3.2: The island parsers for Java

Section	Nonterminal symbol	# of grammar rules without lakes	# of grammar rules with lakes	# of lakes	# of ALT
7.3	CompilationUnit	279	16	1	1
7.4	PackageDeclaration	277	22	2	3
7.5	ImportDeclaration	280	137	5	8
7.6	TypeDeclaration	280	109	6	8
8.1	ClassDeclaration	274	80	5	6
8.3	FieldDeclaration	274	206	3	3
8.4	MethodDeclaration	274	130	5	8
8.6	InstanceInitializer	274	274	0	0
8.7	StaticInitializer	274	21	1	1
8.8	ConstructorDeclaration	274	236	1	1
8.9	EnumDeclaration	274	94	4	7
9.1	InterfaceDeclaration	274	88	4	6
9.3	ConstantDeclaration	274	204	3	3
9.4	InterfaceMethodDeclaration	274	133	5	8
9.6	AnnotationTypeDeclaration	274	39	2	2
9.7	Annotation	274	26	1	1
10.6	ArrayInitializer	274	274	0	0
14.2	Block	274	274	0	0
14.4	LocalVariableDeclaration-Statement	274	274	0	0
14.5	Statement	274	274	0	0
14.8	StatementExpression	274	274	0	0
14.11	SwitchBlock	274	23	1	1
14.14	BasicForStatement	274	231	4	13
14.20	TryStatement	274	29	2	2
15.2	Expression	274	274	0	0
15.8	Primary	274	274	0	0
15.9	ClassCreator	274	72	3	6
15.10	ArrayCreator	274	73	2	6
15.12	Arguments	274	274	0	0
15.15	UnaryExpression	274	274	0	0
15.16	CastExpression	274	274	0	0
15.17-24	InfixExpression	274	274	0	0
15.25	ConditionalExpression	274	274	0	0
15.26	AssignmentExpression	274	274	0	0
15.27	LambdaExpression	274	82	2	8
15.28	ConstantExpression	274	174	1	3

of *ALT* is the sum of the alternative symbols for every lake. We counted duplicate symbols more than once. The grammar defined without lakes needs 277 rules. `PackageDeclaration` is described in JLS 7.4 as follows:

```
PackageDeclaration:
  {PackageModifier} package Identifier {. Identifier} ;
PackageModifier:
  Annotation
```

It consists of an optional `PackageModifier`, the `package` keyword, comma-separated identifiers, and a semicolon. `PackageModifier` is an annotation. When we used a lake symbol, we could substitute a lake symbol for the identifiers between `package` and a semicolon. The alternate symbol for this lake symbol was a semicolon. We could also redefine `PackageModifier` as a lake

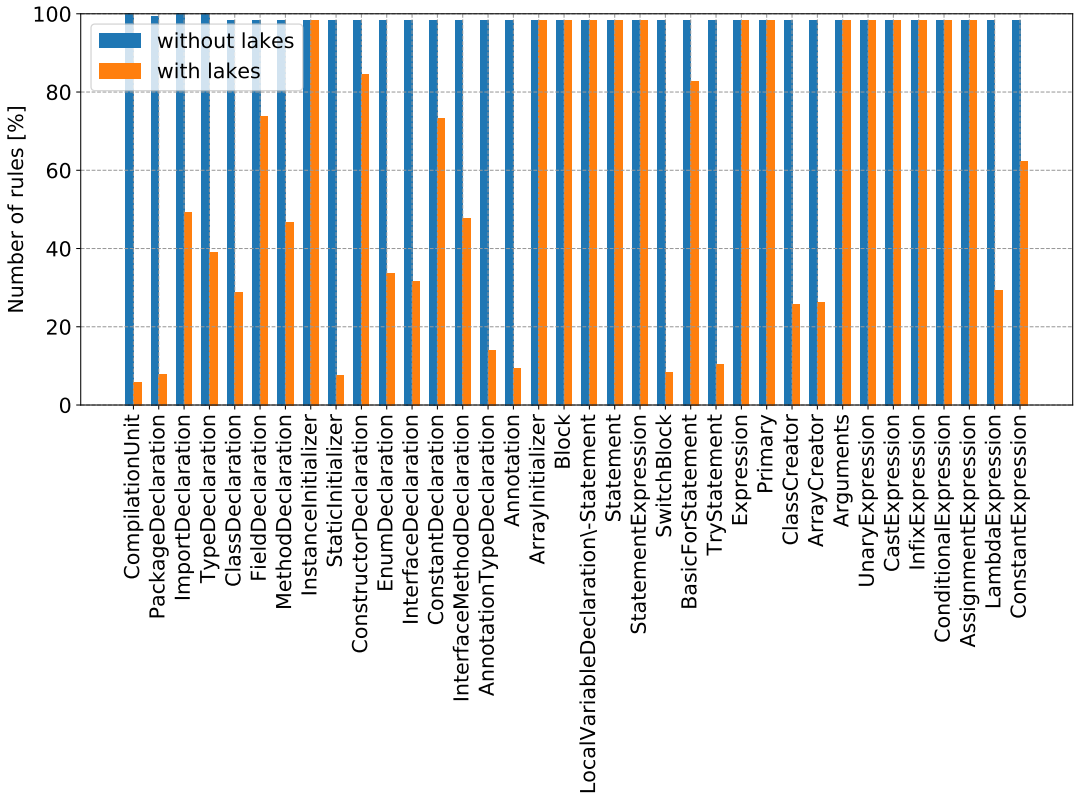

Figure 3.4: The number of the rules for each island parser for Java

symbol following `'@'`. However, when we did not use a lake symbol, we used only the `sea` symbol. Because we had to define the `Annotation` symbol for the `PackageModifier` symbol, we had to define almost all the other nonterminals included in the full-featured grammar. An annotation may include a lambda expression and its body may include almost all kinds of statements and declarations. The number of the grammar rules in this case was 277; only two rules could be omitted.

Figure 3.4 illustrates the comparison of the number of rules listed in Table 3.2. The height of bars indicates the number of rules in percentage. The number of the rules for the full-featured grammar is 100%. Each pair of bars corresponds to the island parsers for recognizing the same nonterminal symbol. The left bar indicates the island parser without lakes while the right bar indicates with lakes.

Without Lakes

The experiments revealed that an island parser for Java did not reduce the number of grammar rules unless it uses lakes. The best case was 274 rules whereas the full-featured Java parser needs 279 rules; only 2% reduction. In Java, most program constructs may include almost all kinds of program constructs. As shown above, a `PackageDeclaration` may include an annotation and an annotation may include a lambda expression. A lambda expression may include all kinds of statements and declarations in its body. A `PackageDeclaration`, therefore, may indirectly include all kinds of statements and declarations. Without a lake, the grammar for the island parser for `PackageDeclaration` must include the

rules for almost all nonterminal symbols in Java.

With Lakes

If lakes are used, 22 of 36 island parsers needed a smaller number of grammar rules than the full-featured Java parser and the island parser using no lakes. The 22 island parsers are 61% among all. They needed 16 to 231 rules. `CompilationUnit` needed 16 rules and it achieved 94% reduction. `BasicForStatement` needed 231 rules and it achieved 17% reduction. The rules included 1 to 6 lake symbols. The lake symbols automated the enumeration of 1 to 13 alternative symbols. They enumerated 1.7 alternative symbols per lake.

Some island parsers needed a number of grammar rules for correctly recognizing an island. For example, `MethodDeclaration` needed 130 rules despite 5 lake symbols included in the rules. It needed such a large number of rules because it had to be distinguished from `InterfaceMethodDeclaration`. The following code snippet:

```
abstract void foo(int x);
```

is recognized as a `MethodDeclaration` when it is enclosed in a class declaration. However, it is recognized as an `InterfaceMethodDeclaration` when it is enclosed in an interface declaration. To recognize only a `MethodDeclaration`, the island parser had to also recognize `ClassDeclaration` and `InterfaceDeclaration` so that it could distinguish `MethodDeclaration` from `InterfaceMethodDeclaration`.

`Block` needed 274 rules for the same reason. Its grammar is just slightly smaller than the grammar for the full-featured Java parser. A `Block` recognizes a token sequence that starts with `{` and ends with `}` but other nonterminal symbols also recognize such a sequence. `ClassBody`, `SwitchBlock`, `EnumBody`, and so on recognize the sequence. To distinguish them, we had to define the rules for recognizing all of them. If we did not distinguish them but we just wanted to recognize any kind of block-like structure, the grammar for this island parser would be much smaller.

The island parsers for expressions, such as `Expression`, `Primary`, and `UnaryExpression`, needed a large number of rules despite the use of lakes. Since Java supports infix operators, these nonterminal symbols do not start with a particular keyword. It is difficult to spot where these symbols start in the sea or a lake. Thus, we had to recognize all the program constructs that may enclose such an expression. This is why the grammars had to include almost all the nonterminal symbols that the full-featured grammar does. Although the island parser for a most kind of expression needed a large grammar, `LambdaExpression` was an exception. It only needed 82 rules since it always starts with a parenthesized parameter list or comma-separated identifiers, and an arrow `->`.

3.4.2 Python

For the full-featured Python grammar, we chose the grammar distributed with Python 3.7.4. Since it was a LL(1) grammar, we manually translated it into a PEG. The number of nonterminal symbols in the grammar was 187. The original grammar requires that the input text is preprocessed, so that its indentations will be converted into tokens `INDENT` and `DEDENT`. Our PEG translation of this grammar also takes similarly preprocessed text as its input.

Table 3.3: The island parsers for Python

Section	Nonterminal symbol	# of grammar rules without lakes	# of grammar rules with lakes	# of lakes	# of ALT
6.13.	lambdef	105	20	3	8
7.3.	assert_stmt	107	12	1	2
7.4.	pass_stmt	8	8	0	0
7.5.	del_stmt	107	12	1	2
7.6.	return_stmt	107	12	1	2
7.7.	yield_stmt	111	16	1	2
7.8.	raise_stmt	107	12	1	2
7.9.	break_stmt	8	8	0	0
7.10.	continue_stmt	8	8	0	0
7.11.	import_stmt	25	17	3	5
7.12.	global_stmt	14	12	1	2
7.13.	nonlocal_stmt	14	12	1	2
8.1.	if_stmt	188	34	5	13
8.2.	while_stmt	188	31	5	12
8.3.	for_stmt	188	36	6	15
8.4.	try_stmt	188	37	5	13
8.5.	with_stmt	188	33	6	15
8.6.	funcdef	188	36	6	13
8.7.	classdef	188	35	6	14
8.8.	async_funcdef	188	41	5	12

We implemented 20 island parsers on top of our PEGIsland parser. For each parser, we implemented two versions with lakes and without lakes. As an island, each of these parsers recognizes one of the nonterminal symbols in the full-featured grammar. We selected 20 nonterminal symbols from the full-featured grammar. These 20 symbols correspond to the program constructs described in a dedicated section 7 or 8 in the Python Language Reference [1]. We excluded expression statements and assignment statements but added lambda expressions from Section 6 because we knew that island parsing was not suitable for expressions after we had conducted the experiments for Java. To verify the correctness of our island parsers, we ran the island parsers to parse 1634 Python source files under the `lib` directory of the Python 3.7.4 distribution.

Table 3.3 lists the results. Each row represents the result of each island parser. Figure 3.5 illustrates the comparison of the number of rules listed in Table 3.3.

Without Lakes

As in Java, 8 island parsers could not reduce the number of grammar rules without lakes. For example, `if_stmt` and `while_stmt` could not reduce because `if` and `while` statements may enclose all other program constructs in its body. They rather slightly increased the number of rules due to the introduction of the *sea* symbol.

However, because the Python grammar is relatively simpler than Java's, the other 12 island parsers implemented without lakes could successfully reduce the number of grammar rules. For example, `pass_stmt`, `break_stmt`, and `continue_stmt` needed only 8 rules. They achieved 96% reduction. The 6 island parsers such as `lambdef` and `assert_stmt` achieved 41–44% reduction.

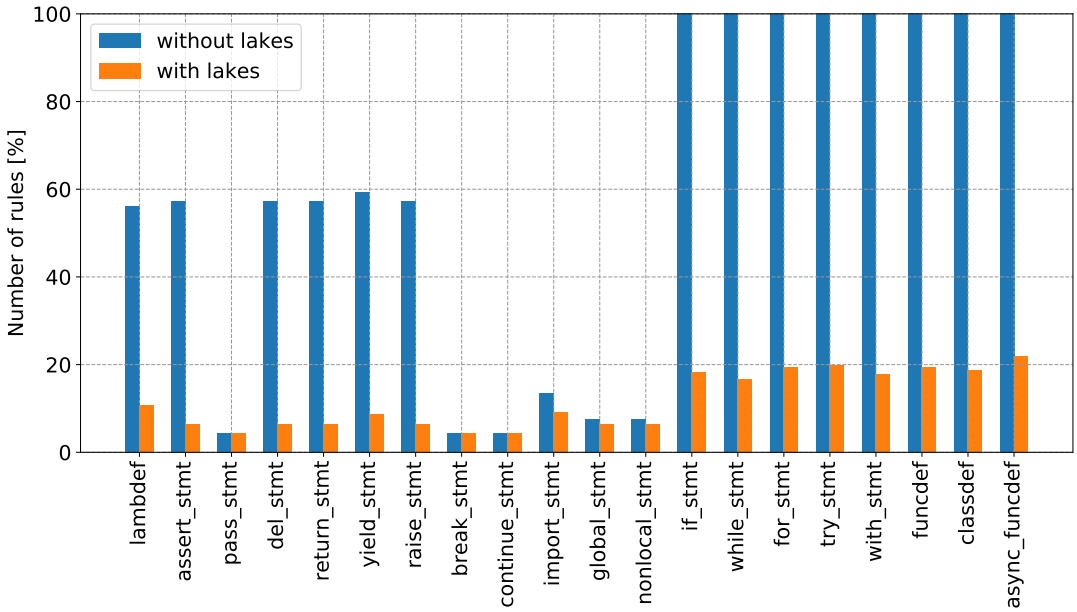

Figure 3.5: The number of the rules for each islands parsing for Python

With Lakes

All the island parsers implemented with lakes successfully reduced the number of grammar rules. They achieved more than 78% reduction. Unlike `Block` in Java, different nonterminal symbols in Python never recognize the same token sequence as their parts. This was a reason why lake symbols effectively reduced the number of grammar rules. Every lake symbol automated the enumeration of 2.4 alternative symbols in average.

Compared to the island parser without lakes, the island parser with lakes achieved larger reduction of the grammar rules. The lake symbols could reduce the number of grammar rules even when the island parsers without lakes could reduce it against the full-featured parser. For example, the island parser for `lambdef` without lakes reduced 44% against the full-featured parser but the parser with lakes reduced 89%. This was 45% improvement. The exceptions were the three parsers `pass_stmt`, `break_stmt`, and `continue_stmt`, which needed only 8 rules. They could not reduce the number of their grammar rules against the parser without lakes.

3.4.3 Summary of the Experiments

For our first research question, to what extent the lake symbols reduce the number of grammar rules for parsers, the results of our experiments revealed that the lake symbols effectively reduced the number of grammar rules for Java and Python except the nonterminal symbols for expressions and some others. The lake symbols worked better for Python. Excluding the nonterminal symbols for expressions (and `lambdef`), the average reduction rate for Python was 89% whereas it was 42% for Java. This would be related to the syntactical complexity of the two languages. Python is much simpler than Java. However, in both languages, the lake symbols were not effective when the start and the end of

the nonterminal symbol were not spot by a particular symbol. They were not effective either when more than one nonterminal symbols recognized a token sequence as their parts.

For our second research question, whether the use and disuse of lakes affect the number of grammar rules for island parsing, the results of the experiments revealed that the use of lakes was indispensable to reduce the number of grammar rules. However, in several cases, the use of lakes did not affect the number of grammar rules. The number of rules did not change no matter if lakes are used or not.

One of the threats of validity is that we used only one grammar as the full-featured grammar for each language, Java and Python. If we use a different full-featured grammar, we might see different results. Another threat is that we only examined Java and Python. The effects of the lakes may change depending on the syntactical complexity of the language.

3.5 Related Work

Koopa[1] is a parser generator with support for island grammars that has been used to implement an industrial-strength Cobol parser. Koopa provides the skip-to operator to support the description of *water*. The skip-to operator skips anything it encounters up to the given pattern. While the skip-to operator cannot handle nested islands in *water*, our lake symbol can handle them.

The semi-parsing approach used for Agile parsing [23] can be considered as a variant of island parsing. It proposed the use of the `not` operator in TXL [20, 19] so that the user can manually specify the alternative symbols, which our lake symbol automatically derives from the rest of the grammar.

Bounded seas [55] automatically calculate the alternative symbols and use them for the `not` predicate as our lake symbols do. However, as far as we understand, the bounded seas calculate only a subset of all the alternative symbols, which corresponds to our `SUCCEED`. Thus, its applicability is limited.

The generalized parsers, such as CYK [96], GLL [83], and GLR [88], can be used for island parsing although they deal with context-free grammars (CFGs) but our work deals with PEGs. The generalized parsers generate not a parse tree but parse forest, which consists of several candidates of parse tree for a given source program. This is useful to recognize *water* without specifying complex alternative symbols. However, the user must manually disambiguate the forest to obtain a single parse tree.

Afroozeh *et al.* [2] addressed this disambiguation for their GLL-based island parsing used for the Tom language. They proposed to use pattern matching for the disambiguation.

Noise skipping parsing [57] proposed the GLR* parser, which can parse any input sentence by ignoring unrecognizable parts of the sentence. The application domain of GLR* is speech recognition. GLR* supports the *sea* but not the *lakes*.

Goloveshkin *et al.* [38] proposed the special terminal symbol `Any` that matches zero or more tokens constituting uninteresting parts of a given input program. `Any` can be used to define the *sea* but not the *lakes*.

[1] `https://github.com/krisds/koopa`.

Fuzzy parsing is the parsing approach where only certain parts of a programming language is recognized. A hand-written fuzzy parser was presented for C++ in [11]. A framework for fuzzy parser was proposed by Koppler [54]. In the framework, the scanner searches an input source program to spot the anchors where the parser starts parsing. The user must manually implement that scanner.

Klusener *et al.* [52] proposed the method that constructs a skeleton grammar for tolerant parsing from a given grammar. The parsers derived from the skeleton grammar can be considered as an island parser. Their proposal does not support lakes but could be extended to support our lake symbols.

3.6 Summary

This chapter presented the lake symbols for island parsing and an extended PEG supporting lake symbols. The lake symbol is a special grammatical symbol like a wildcard. The user can use lake symbols to define rules for an island with lakes. The user can also define a rule for each lake symbol to specify inner islands or recursive structures inside the lake. This chapter proposed an algorithm that translates our extended PEG to the normal PEG that can be given to existing PEG parsers. We experimentally revealed that lake symbols effectively reduced the size of the grammars for island parsers for Java and Python, compared to their fully detailed grammars.

3.6.1 Future Work

We have defined the semantics of lake symbols by translating them into a specific grammar. Defining its semantics without the translation is our future work.

The implementation of lake symbols for another grammar class is also future work. The concept of lake symbols will be applicable not only PEGs but also context-free grammars (CFGs) and their subsets such as LR and LL grammars. An issue is the ambiguity of a grammar when introducing lake symbols into another grammar class than PEGs. Since a lake symbol works as a wildcard, the grammar written with lake symbols tends to be ambiguous. This ambiguity causes shift/reduce conflicts, for example, when using the LR parsing. We expect that the alternative symbols proposed in this chapter can be used for disambiguating the grammar. The algorithm for calculating the alternative symbols in CFGs would be similar to what we presented in this chapter. In this chapter, we selected PEGs since PEG's prioritized choice operator does not cause ambiguity even when lake symbols are used.

Other future work is to evaluate with more practical applications. Our experiment confirmed that the number of rules to extract a specific kind of programming construct decreased with lakes. Although we believe that reducing rules ease the construction of island parsing, the relation between the real workload and the number of rules has not been studied experimentally. Besides, while we assumed that the application is interested in only one kind of programming constructs in our experiment, a realistic application may need more than one kind of programming constructs.

Chapter 4

Interactive Grammar Editor

4.1 Introduction

Island parsing is a technique to extract programming construct of interest without specifying a complete grammar for a language. It extracts program constructs of interest as islands and skips the remaining parts of the program as water. An island grammar consists of rules for islands and water. Typically, an island grammar for island parsing contains fewer rules than the complete grammar since water can be handled by a wildcard character.

While island parsing reduces the effort to write a grammar for parsing, debugging island grammar is not easy. If we fail to write some grammar rules correctly, the parsing result is not what we expect. If the grammar were not an island grammar but a complete one, the debug would be easier. We could check that each rule in the grammar is correct by comparing it with the language specification. In the case of the island grammar, we cannot compare the rules in the island grammar with the language specification since most parts of the original grammar are omitted and modified in the island grammar.

To check whether the island grammar is correct or not, we repeat trial-and-errors. Therefore we need a way to confirm the parsing result interactively and effectively step by step while editing an island grammar. While interactive tools for generating parsers such as Parsify [61] and Parsimony [60] have been proposed, they do not support island grammars. They provide a way to edit a grammar interactively through GUI operations by using an example program. For example, the user selects expressions in an example program and labels them as expressions. The user can get an unambiguous grammar by repeatedly applying operations. However, the grammar is ambiguous until the user finishes labeling all the programming constructs in the language. The resulting grammar is not an island grammar but a complete grammar.

We propose an incremental description of island grammars with lake symbols and tool support for it. Our approaches enable the description of an island grammar in a step-by-step manner. In each step, the user can add a rule to the grammar and immediately test the rule. This prevents us from repeating inefficient trial-and-error to write an island grammar. We also propose GUI operations for the user to add a new rule to the grammar. These GUI operations take an example text as input and generate a new rule based on the user's selection. This prevents the user from making a specific type of mistake when editing the island grammar.

The contributions of this chapter are twofold:

- We propose an example-based GUI tool for editing grammar. This tool provides three GUI operations for the user to add a new rule to the grammar. These GUI operations take an example text as input and generate a new rule based on the user's selection. This prevents the user from making a specific type of mistake when editing the island grammar.
- We implemented our tool called PEGSEED that supports incremental editing and GUI operations for editing island grammars. We developed a framework for syntax extension based on PEGSEED. We confirmed that the possible mistake is detected soon in our PEGSEED. We also confirmed that parsers for syntax extensions could be constructed only with proposed GUI operations.

4.2 Motivation

We need tool support for writing an island grammar. While island grammars are suitable for extracting specific programming constructs of interest, it is difficult for non-experts to write an island grammar. In this section, we introduce island parsing and the difficulty of writing island grammars.

4.2.1 Use Case of Island Parsing

Island grammars are a best practice in developing parsers for software engineering tasks such as syntax extension, source code analysis, and so on. These tasks are only interested in specific programming constructs, and island grammars can be used to extract such specific programming constructs. Island parsers based on an island grammar can be seen as a pattern matcher like *grep* based on regular expressions. They extract only interesting programming constructs as *islands* and skip the remaining parts of the program as *water*.

One of the applications of island parsing is syntax extensions for the existing programming languages. Syntax extension is a promising technique to extend existing programming languages with additional programming constructs for application specific-domain. Syntax extension enables concrete syntax for domain-specific abstraction in host programming languages.

Let us suppose that we want to improve the readability of programs for a specific domain by implementing syntax extension for the programming language. For example, if we have to repeatedly write many programs that have a lot of nested `if` statements as shown in Listing 4.1, we want to extend the language such that it supports `onlyWhen` statements that enable the program can be rewritten as shown in Listing 4.2. At first glance, an `onlyWhen` statement may look like just a function call in the language. However, it is not a function call, but a part of special programming construct specifying that the statements following the `onlyWhen` statement until the end of the surrounding block must be executed only when the condition given to `onlyWhen` is satisfied.

This `onlyWhen` can be implemented by using a program transformer based on an island grammar. The program transformer converts `onlyWhen` constructs into native `if` statements. For example, the program transformer converts Listing 4.2

Listing 4.1: Program with nested if statements

```
1  {
2    doit();
3    if (fooIsNeeded()) {
4      foo();
5      if (barIsNeeded()) {
6        for (i = 0; i < 3; i++) {
7          bar();
8          if (bazIsNeeded()) {
9            for (j = 0; j < 3; j++) {
10             baz();
11           }
12         }
13       }
14     }
15   }
16 }
```

Listing 4.2: Example program with `onlyWhen`

```
1  {
2    doit();
3    onlyWhen(fooIsNeeded());
4    foo();
5    onlyWhen(barIsNeeded());
6    for (i = 0; i < 3; i++) {
7      bar();
8      onlyWhen(bazIsNeeded());
9      for (j = 0; j < 3; j++) {
10       baz();
11     }
12   }
13 }
```

into Listing 4.1. This program transformation can be done by pattern rewriting like *sed*. When the program transformer encounters the following pattern in the program,

```
{
   <A>
   onlyWhen(<B>);
   <C>
}
```

, where `<A>`, `<B>` and `<C>` indicates the ommision of parts of the program, it rewrites the program as shown below.

```
{
  <A>
  if (<B>) {
    <C>
  }
```

Listing 4.3: Island grammar for the `onlyWhen` constructs

```
1 program = sea*
2 sea> = island | water
3 island =
4    '{' sea* 'onlyWhen' '(' water* ')' ';' sea* '}'


}
```

Note that the `onlyWhen` construct starts with not `onlyWhen` keyword in line 3 in Listing 4.2 but the left curly brace in line 1. Since it must detect the end of the block surrounding the `onlyWhen` statement, the `onlyWhen` construct starts with the corresponding left curly brace.

While it may seem that a regular expression is suitable for this pattern matching, the extraction of the above example cannot be handled by a regular expression since it cannot handle recursive structures of the programming language. To handle nested structures of a programming language, we need a parser.

We can use an island grammar to obtain a parser that can extract extended programming constructs from a program. An island grammar is a specific grammar written for extracting only interesting programming constructs of a language. An island grammar does not mean a specific grammar or a grammar class. Moreover, island grammar is not bound to particular syntax formalism or parser generation tools. An island grammar can be described in any syntax formalism such as BNF, SDF, and PEG.

When we want to extract only `onlyWhen` constructs, we can write an island grammar as shown in Listing 4.3 written in EBNF. In line 1, `program` is defined as repetition of `sea` that is defined as `island` or `water`. `island` represents the `onlyWhen` construct, which we are interested in. On the other hand, `water` represents the remaining parts of the program. The right-hand side of the rule for `island` contains `sea`s and `water` at the place that we are not interested in. We use `sea` not `water` at the place where nested `onlyWhen` constructs may appear. `water` is like a wildcard symbol in regular expressions.

As shown in Listing 4.3, the island grammar is significantly smaller than the complete grammar for the language. If we had to write the rules for the uninteresting part of the program, we would have to write much more rules for the language. In general, programming languages need hundreds of rules for their grammars. Therefore, island grammars can reduce the effort to obtain a parser for extracting programming constructs dramatically.

4.2.2 Difficulty in Writing Island Grammar

It is not easy for non-experts to write a practical island grammar since island grammars tend to be ambiguous. The grammar is ambiguous when the resulting parse tree is not unique. In other words, ambiguous grammar generates not a single parse tree but a parse forest. In general, grammars must not be ambiguous for software engineering tasks.

For example, the grammar in Listing 4.3 may be ambiguous depending on the definition of `water`. Ideally, we want to define water as just a wildcard symbol. However, if we define the `water` nonterminal as just a naive wildcard,

Listing 4.4: Island grammar with lake symbls

```
1 program <- <sea>*
2 <sea> <-  island / water
3 island <-
4     '{' <sea>* 'onlyWhen' group ';' <other>* '}'
5 <other> <- island / water
6 water <- group / block
7 group <- '(' <sea>* ')'
8 block <- '{' <sea>* '}'
```

the resulting grammar will become ambiguous. Multiple parse trees are available when we parse a program in Listing 4.2 with a grammar in Listing 4.3. While one possible result is what we want, other results are not. For example, one possible result is produced when the whole program is skipped as water. However, it is obviously not the one the user expects. Without disambiguation, the parser cannot determine which parsing result is what the user expects.

Hence, to use an island grammar in practical applications such as syntax extension, the user must remove the ambiguity from the grammar. Some language tools such as a parser generator or syntax formalisms provide ways to disambiguate a grammar. For example, SDF supports the description of attributes for each rule in a grammar for this purpose. In SDF, the user can use the `prefer` attribute to specify that an island is prioritized to water. However, available disambiguation mechanisms are different among tools or syntax formalism. In the case of PEG, the same prioritizing can be done by using a prioritized choice operator. Moreover, prioritizing is not sufficient for disambiguating a grammar. The user must write the rules for water carefully to prevent water from being matching to the part of an island. Some strategies are proposed for a specific language [2], but some languages provide a more general way to ease the water.

We can use our lake symbols to disambiguate the grammar in Listing 4.3. It can be rewritten as Listing 4.4 in extended PEG with lake symbols. In Listing 4.4, `<sea>` is a lake symbol. A lake symbol works as just a special wildcard without its definition. In the case of Listing 4.4, `<sea>` has its definition in line 2. It can recognizes `island` and explicitly specified `water`. We use `<sea>` at the place where islands or water appear. If the `<sea>` does not recognize islands or explicitly specified water, it works as a wildcard character. For example, we use `<sea>` after '{' in line 4 in Listing 4.4. If the `<sea>` does not recognize `isand` or explicitly specified `water`, it consumes any character until `onlyWhen` appears in the input string. The lake symbol `<sea>` prevents itself from consuming `onlyWhen` as water. Hence a lake symbol is a context-aware wildcard symbol. The user does not need to explicitly specify excluding `onlyWhen` from the water. Similarly, `<sea>` does not consume '}' and ')', which are shore parts of islands.

Nevertheless, writing a correct island grammar is not easy. If we had forgotten to specify the `block` in the rule for water in line 6, the resulting grammar would have been incorrect. Without specifying `block` as water explicitly, the closing brace that is part of a block is treated as a part of an island. For example, when parsing the following `onlyWhen` construct:

```
{ onlyWhen(x);  { doit(); } }
```

, the incorrect parser extracts

```
{ onlyWhen(x);  { doit(); }
```

as a island by mistake. To correctly parse the above program, `{ doit(); }` must be skiped as a water.

To efficiently write a correct grammar, we need tool support for debugging the grammar. If the grammar were not an island grammar but a complete one, the debug would be easier. We could check that each rule in the grammar is correct by comparing it with the language specification. In the case of the island grammar, we cannot compare the rules in the island grammar with the language specification since most parts of original grammar are omitted and modified in the island grammar. To check whether the island grammar is correct or not, we repeat trial-and-errors. Therefore, we need an interactive tool that supports the efficient repetition of trial-and-errors.

What we want is to enable the user to confirm the parsing result interactively and effectively step by step while editing an island grammar. While interactive tools for generating parsers such as Parsify [61] and Parsimony [60] have been proposed, they do not support island grammars. They provide a way to editing grammar interactively through GUI operations by using an example program. For example, the user selects expressions in the example program and labels them as expressions. The user can get a grammar by repeatedly applying operations. However, the grammar is ambiguous until the user finishes labeling all the programming constructs in the language. The resulting grammar is not an island grammar but a complete grammar.

4.3 Interactive Grammar Editing

4.3.1 Editing Grammar Incrementally

Overview

While island grammar can reduce the number of rules for extracting programming constructs, obtaining a correct grammar is not easy. If water is not defined correctly, water may happen to consume a part of an island. Therefore, we need to check whether islands are extracted as intended and water does not consume a part of islands by mistake.

To address the difficulty in writing island grammars, we propose an interactive grammar editing system called *PEGSEED*. With PEGSEED, the user starts editing grammar from an initial island grammar, which treats the whole program as water. The user incrementally adds new islands into the grammar. Since the working parser is always available after each update of the island grammar, the user can immediately confirm that the island is correctly defined by parsing an example program. PEGSEED provides text highlighting functionality for this confirmation. The user can create a new island by adding a new token or concatenating existing small islands whose rules are already tested. Our system also provides interactive operations for this purpose. After multiple iterations of adding new islands into the grammar, the grammar finally becomes an expected one.

Example

We explain how the incremental grammar editing works by using `onlyWhen` constructs introduced in Section 2 in a step-by-step manner. Each step can be done with a GUI operation that our tool provides. We will explain these operations later in this section.

To develop an island grammar, we start editing from the initial grammar as follows:

```
1 program <- <lake>*
2 <lake> <- string
3 string <- r'"([^"]|\")*?"\s*'
```

This initial grammar extracts no island and just skips the whole program as water. The `program` nonterminal is defined as zero or more repetition of `<lake>`. `<lake>` is a lake symbol. In this thesis, we use a convention that the name of a lake symbol is surrounded by `<` and `>`. In line 2, the rule for `<lake>` is defined. The right-hand side of the rule for `<lake>` indicates that `string` may be recognized as `<lake>`. Even if `<lake>` does not match the parsing expression on the right-hand side of the rule for it, it consumes any character like a wildcard symbol. Since we are not interested in string literals, we can handle them as water. `string` consumes a string literal as a chunk. It prevents the parser from recognizing an island in the string literal by mistake. For example, even if the input contains `"{ onlyWhen(x==1); }"`, `water` recognizes it as the string literal. This prevents the parser from extracting the `onlyWhen` construct embedded in the string literal. For the same reason, the parsing expression that recognizes comments in the language must be specified in the right-hand side of the rule for `water` in a practical situation while we omit it in this thesis for simplicity. The right-hand side of the rule for `string` in line 3 is a regular expression surrounded by `r'` and `'` like Python's regular expressions.

From the initial grammar, the user refines the grammar incrementally. To enable the grammar extract `onlyWhen` constructs, the user starts with adding the rules for the tokens which build up `onlyWhen` constructs first. Such tokens includes the `onlyWhen` keyword, parentheses, braces, and semicolon. The user updates the grammar such that the parser can extract these tokens as islands. For example, after the user adds the `onlyWhen` keyword as an island, the grammar is updated as follows:

```
1 program <- <lake>*
2 <lake> <- onlyWhen / string
3 string <- r'"([^"]|\")*?"\s*'
4 onlyWhen <- r'(?<!\w)onlyWhen(?!\w)\s*'
```

In this grammar, the rules in lines 4 was added. The rule in line 2 was also updated such that the right-hand side of the rule contains `onlyWhen` nonterminal as the first operand of the ordered choice expression. It means that `only_when` was added as an island. The `onlyWhen` nonterminal was newly defined nonterminal in line 4. The `onlyWhen` nonterminal recognizes the `onlyWhen` token with additional white spaces. `r'(?<!\w)onlyWhen(?!\w)\s*'` is a regular expression that recognizes an onlyWhen token. Note that `(?<!\w)` matches if a look-behind character is not an alphanumeric character nor an underscore. Similarly, `(?!\w)` matches if a look-ahead character is not an alphanumeric character nor an un-

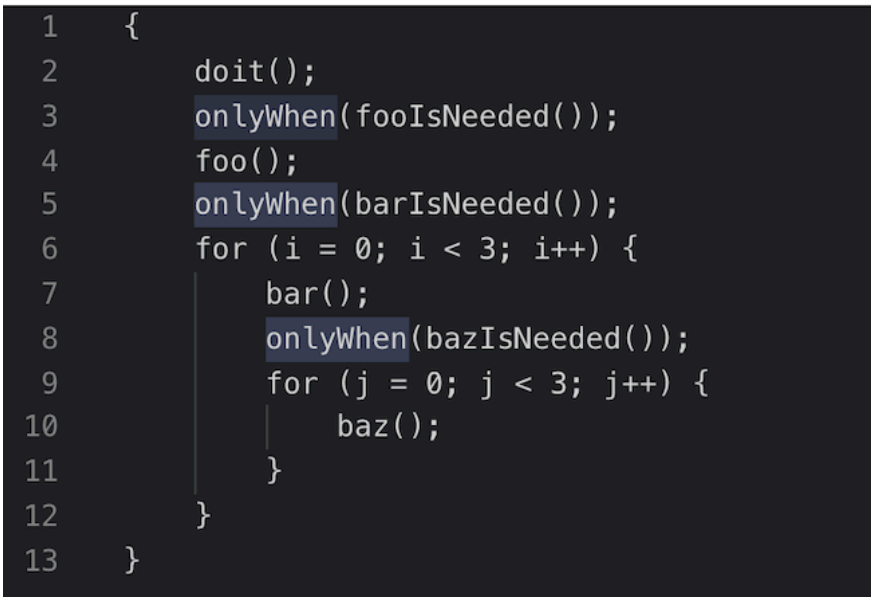

Figure 4.1: Text highlighting

derscore. This prevents the parser from unintentionally recognizing parts of another identifier such as `NotonlyWhen` or `onlyWhenA` as the `onlyWhen` token. Concatenating additional white spaces represented by `\s+` with a token is a typical description pattern in PEG for the scanner-less parser.

After updating the grammar, the user can check if this rule for the island is valid by highlighting the corresponding text areas in the example text. For example, after adding the rule for the `onlyWhen` keyword. The user selects the `onlyWhen` nonterminal for highlighting. Then, PEGSEED highlights the text areas that are recognized as `onlyWhen` as shown in Figure 4.1. In Figure 4.1, three text areas corresponding to `onlyWhen` is highlighted as expected since the rule for the `onlyWhen` nonterminal is correctly defined. If these areas were not highlighted as expected, the user could rewrite the rule for `onlyWhen`.

After adding all tokens as islands, the grammar becomes as follows:

```
1 program <- <lake>*
2 <lake> <- semi / rpar / lpar / rcub / lcub / onlyWhen /
     string
3 string <- r'"([^"]|\")*?"\s*'
4 onlyWhen <- r'(?<!\w)onlyWhen(?!\w)\s*'
5 lcub <- r'\{\s*'
6 rcub <- r'\}\s*'
7 lpar <- r'\(\s*'
8 rpar <- r'\)\s*'
9 semi <- r';\s*'
```

Lines 5-9 are rules for newly added tokens. At this point, the user has confirmed that all tokens are defined correctly with text highlighting.

Now, the user can create a larger island by using already defined tokens. Let us consider adding a rule that recognizes the condition part of an `onlyWhen` construct as a new island. A condition part starts with a left parenthesis and ends with a right parenthesis. Since we are not interested in the strings surrounded

by parentheses, we would like to treat them as water. This can be done by concatenating existing islands as follows:

```
1  program <- <lake>*
2  <lake> <- group/ semi / rcub / lcub / onlyWhen / string
3  string <- r'"([^"]|\")*?"\s*'
4  onlyWhen <- r'(?<!\w)onlyWhen(?!\w)\s*'
5  lcub <- r'\{\s*'
6  rcub <- r'\}\s*'
7  lpar <- r'\(\s*'
8  rpar <- r'\)\s*'
9  semi <- r';\s*'
10 group <- lpar <lake>* rpar
```

Now, `lpar` and `rpar`, which recognize a left and right parenthesis respectively, were removed from the rule for `<lake>` in line 2. Instead, the `group` nonterminal symbol is added on the right-hand side of the grammar in line 2. The rule for `group` is added in line 10. Note that the right-hand side of the rule for `group` contains `<lake>` recursively for skipping uninteresting parts of the program. Since the right-hand side of the rule for `<lake>` contains `group`, the grammar can handle the recursive structure of `group`. We reuse `<lake>` in line 10 while it is also used in line 1. If the reuse of `<lake>` caused a problem, we could create a new lake symbol and put the group on the right-hand side of the rule for the new lake symbol. We can also check if the definition of the `group` is valid with text highlighting.

Next, we continuously define a larger island by concatenating islands already defined and tested. By concatenating `onlyWhen`, `group`, and `semi`, we can define `onlyWhenStmt` as follows:

```
onlyWhenStmt <- onlyWhen group semi
```

This `onlyWhenStmt` is added to the right-hand side of the rule for `<lake>` as follows:

```
<lake> <- onlyWhenStmt / group / rcub / lcub / string
```

Note that `onlyWhen` and `semi` have been removed from the above rule. They do not need to be recognized as `<lake>` since they are now recognized as part of `onlyWhenStmt`. On the other hand `group` is still in the rule for `<lake>`. Since the right-hand side of the rule for `group` includes `<lake>`, `group` must be specified in the rule for `<lake>` to handle the recursive structure of `group`.

Now, all the small islands to build up an `onlyWhen` construct seem to be already defined and tested. We can define a rule for an `onlyWhen` construct as follows by concatenating these islands:

```
onlyWhenConstruct <- lcub <lake>* onlyWhenStmt <other>*
   rcub
```

We use lake symbols to skip inner parts of `onlyWhen` constructs since we are not interested in them. We need two different lake symbols, `<lake>` and `<other>`. They are different in how they consume water. `<lake>` must not consume `onlyWhenStmt` as water for the parser to recognize the following `onlyWhenStmt`. On the other hand, `<other>` must skip `onlyWhenStmt` such as the one in line 5 in Listing 4.2 since they are uninteresting parts of the surrouding `onlyWhenConstruct`.

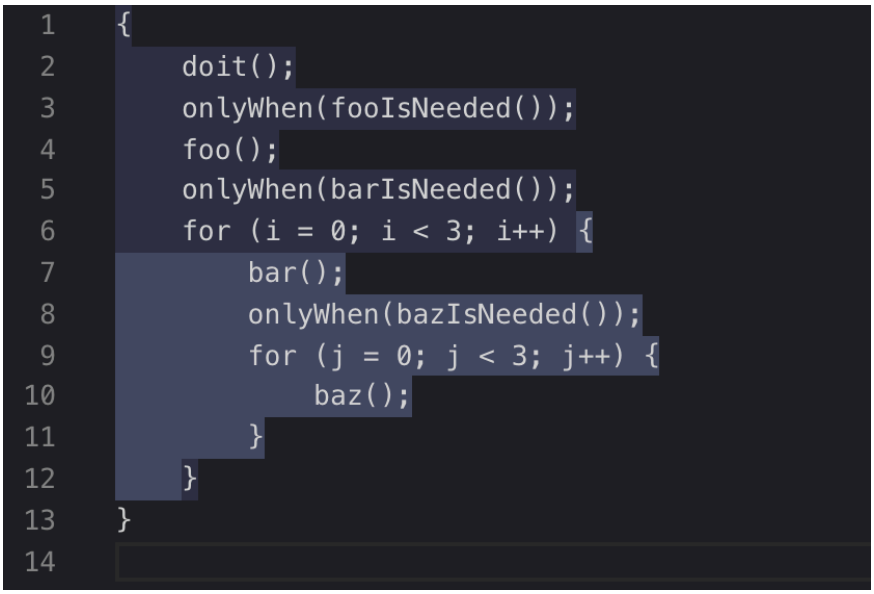

Figure 4.2: Text highlighting for invalid `onlyWhenConstruct`

A best practice for using lake symbols is to use different lake symbols for each different part in the grammar, since they are context aware. `onlyWhenConstruct` is added to the right-hand side of the rule for `<lake>` as follows:

```
<lake> <- onlyWhenConstruct / group / string
```

The nonterminals used in the rules for `onlyWhenConstruct` is removed, and `onlyWhenConstruct` is added. At this point, we can check if the above rule is valid by highlighting the areas recognized by the `onlyWhenConstruct` nonterminal. Unfortunately, the result is not what we expected, as shown in Figure 4.2. While the number of `onlyWhen` constructs is as we expected, the extracted area is not correct. We expected the right curly brace in line 13 is highlighted as part of the outer `onlyWhen` construct. And the right curly brace in line 12 is highlighted as part of the inner one. However, the outer one ends with the right curly brace in line 12. Similarly, the inner one ends with the right curly brace in line 11 and the following spaces, which are part of the `for` statement in lines 9–11. The result is due to that we did not define `block` that begins with a right curly brace and ends with a curly brace as water.

Hence, the user undoes the previous definition of `onlyWhenConstruct`, and defines the following rule before defining `onlyWhenConstruct` instead:

```
<block> <- lcub <lake>* rcub
```

The above `<block>` is added to the right-hand side of the rule as follows:

```
<lake> <- block / group / semi / onlyWhen / string
```

After updating the grammar, we can check if the rule for `<block>` is properly defined by text highlighting. The result is shown in Figure 4.3.

At this point, the `onlyWhen` constructs are recognized as `<block>`. We can differentiate them from `<block>` by adding the rule for `onlyWhenConstruct` as explained earlier in this section. Finally, we obtain the following grammar in Listing 4.5. Now, we can check if the `onlyWhenConstruct` nonterminal is valid with text highlighting.

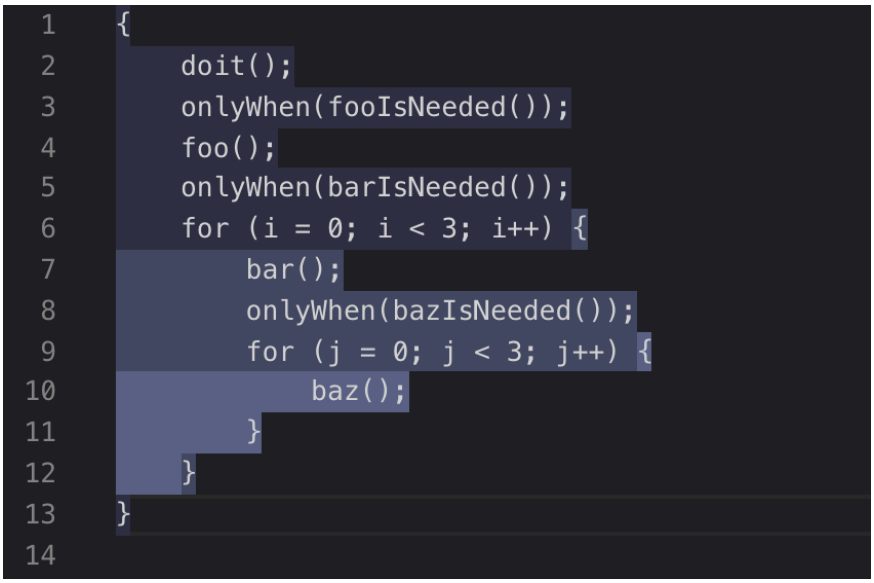

Figure 4.3: Text highlighting for `block`

Listing 4.5: Island grammar for the onlyWhen construct

```
1  program <- <lake>*
2  <lake> <- onlyWhenConstruct / block / group / water
3  string <- r'"([^"]|\")*?"\s*'
4  onlyWhen <- r'(?<!\w)onlyWhen(?!\w)\s*'
5  lcub <- r'\{\s*'
6  rcub <- r'\}\s*'
7  lpar <- r'\(\s*'
8  rpar <- r'\)\s*'
9  semi <- r';\s*'
10 group <- lpar <lake>* rpar
11 onlyWhenStmt <- onlyWhen group semi
12 block <- lcub <lake>* rcub
13 <other> <- onlyWhenConstruct / block / group / water
14 onlyWhenConstruct <- lcub <lake>* onlyWhenStmt <other>*
      rcub
```

As described above, we can get the correct island grammar quickly. Incremental definition of rules eases the grammar description. By defining a new island with already tested islands, we immediately found a mistake after defining a new rule if something is wrong. This benefit comes from incremental description and text highlighting feature of our PEGSEED.

4.3.2 Syntax of Generated Grammars

Our PEGSEED always produces a grammar, whose meta-syntax is described in EBNF in Listing 4.6. The grammar consists of one or more rules, as shown in line 1. Each rule is a `token_rule`, `sequence_rule`, or `choice_rule` (line 2). PEGSEED provides GUI operations for defining these rules.

`token_rule` represents a token rule that recognizes a specific kind of token

Listing 4.6: Meta-syntax of an generated PEG in EBNF

```
1 grammar = rule+
2 rule = token_rule | choice_rule | sequence_rule
3 token_rule    = nonterminal '<-' regex
4 sequence_rule = nonterminal '<-' (nonterminal '*'?| lake
     '*')+
5 choice_rule   = (nonterminal / lake)  '<-' nonterminal ('/'
      nonterminal)+
```

in a language. The left-hand side of a token rule is a nonterminal symbol. The right-hand side is a regular expression. This regular expression starts with r' and ends with ' like a regular expression literal in Python. For example, the following rule is a token rule:

```
onlyWhen <- r'(?<!\w)onlyWhen(?!\w)\s*'
```

`onlyWhen` is a nonterminal and `r'(?<!\w)onlyWhen(?!\w)\s*'` is a regular expression.

`squence_rule` represents a sequence rule that recognizes a specific sequence of nonterminal or lake symbols. The left-hand side of a sequence rule is a nonterminal symbol. The right-hand side is a parsing expression with PEG's sequence operator. Each operand of the sequence operator is a nonterminal or lake symbol with an optional postfix operator. For example, the following rule is a `sequence_rule`:

```
block <- lcub <lake>* rcub
```

Operands of PEG's sequence operator is `lcub`, `<lake>*`, and `rcub`. They are nonterminals or lake symbols with an optional postfix operator. For instance, `lcub` is just a nonterminal, and `<lake>*` is a lake symbol with the optional postfix operator. `*` is PEG's zero or more operator.

`choice_rule` represents a choice rule where a newly defined nonterminal can be added. Its left-hand side is a nonterminal or lake symbol. The right-hand side consists of one or more nonterminal symbols separated by PEG's choice operator. For example, the following rule is a choice rule:

```
<lake> <- string
```

This rule contains only one nonterminal symbol. When we define the `onlyWhen` nonterminal, we can add it to the right-hand side of the rule for `choice_rule` as follows:

```
<lake> <- onlyWhen / string
```

The right-hand side of the rule for `<lake>` contains `onlyWhen` and `string` separated by PEG's choice operator `/`.

4.3.3 Example-Based GUI Operations

Overview

Our PEGSEED always keeps the grammar under editing a correct island grammar. By utilizing this property, we can provide advance assistance for adding a new rule into the grammar. When the grammar is updated, PEGSEED parses

an example text that the user provided with the latest grammar and obtains the parse tree. This parse tree has information about which nonterminal symbol corresponds to each subpart of the example text. PEGSEED uses this information to realize assistance for updating the grammar. PEGSEED provides three GUI operations by which the user can add a new rule with assistance.

To apply one of the three GUI operations, the user takes the following three steps:

1. Select a text area in the example text
2. Select an operation to be executed
3. Fill out a dialog box corresponding to the selected operation

In Step 1, the user selects a text area that should be recognized by the rule to be added. This Step is in common regardless of which operation to be executed. In Step 2, the user clicks a button that corresponds to the operation to be executed. Then, the dialog box for the operation pops up. Step 3 depends on the operation that the user selected in Step 2.

Token Operation

The token operation adds a token rule corresponding to `token_rule` in line 3 in Listing 4.6. To apply this operation, the user selects a text area that should be recognized by the token rule to be added. Then, PEGSEED shows the dialog for the user to add the rule to the grammar as shown in Figure 4.4. This dialog consists of three parts: (1) a text field for the name of nonterminal to be added, (2) a text field of a regular expression, and (3) a list of nonterminals for choice rules.

The user specifies the name of the nonterminal for the rule to be updated in the text field (1). The default name of a nonterminal is automatically filled by PEGSEED based on the text which the user selected. The dialog in Figure 4.4 is shown after the user selects "onlyWhen" in the example text. PEGSEED automatically fills the text field for the name of the nonterminal symbol with "onlyWhen" as its default value. The user can change the name if needed. If the same name already exists in the current grammar, PEGSEED warns it.

With text field (2), the user specifies a regular expression that recognizes the token to be selected. The default regular expression for the token is automatically filled by PEGSEED based on the selected text. In Figure 4.4, `(?<!\w)onlyWhen(?!\w)\s*` is automatically fill by PEGSEED. The default regular expression is based on the text that the user selects. When the selected text starts with _ or an alphanumeric character, a negative look-behind and look-ahead prefixes are appended to the regular expression. It asserts that what immediately precedes or follows the token is not _ or alphanumeric character matched by regular expression `\w`. This prevents an unexpected character string such as "notonlyWhen" from being matched by the regular expression. `\s*` is also appended at the tail of the regular expression to skip the trailing white spaces. It is a practical technique to make the token rule consume trailing white spaces as part of it in scanner less parsing. The user can modify the pattern if the default regular expression is not what she expected.

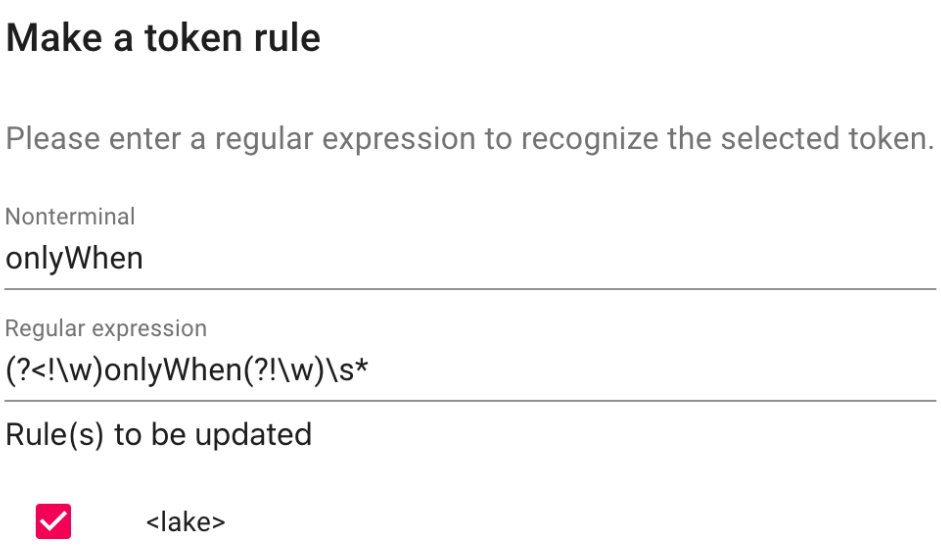

Figure 4.4: Dialog for the token operation

The user also specifies where the new rule is specified in the grammar. The new rule can be inserted in one or more choice rules in the grammar. List (3) shows all the candidate rules in the grammar. In the case of Figure 4.4, `<lake>` is only one element of the list. PEGSEED automatically selects an element of the list by default based on the text which the user selected. The default element is what the user selected is currently recognized. In the case of Figure 4.4, since each character of "onlyWhen" is recognized as `<lake>`, it is selected by default. When there are multiple candidate choice rules, the user can select ones explicitly.

Sequence Operation

The sequence operation adds a sequence rule, which corresponds to `sequence_rule` in line 4 in Listing 4.6. To apply the sequence operation, the user selects a text area that the sequence rule to be added should recognize. Then, the PEGSEED shows the dialog for the user to add the rule to the grammar as shown in Figure 4.5. This dialog consists of four parts: (1) a text field for the name of nonterminal to be added, (2) a list of list boxes, (3) a list of nonterminals for choice rules to be updated, and (4) a list of nonterminal that must be removed from choice rules selected in (3).

The user specifies the name of the nonterminal in the text field (1). In the case of Figure 4.5, the user specifies "onlyWhenStmt" as a name of the nonterminal.

The user specifies the rule to be added by using the list of list boxes (2). Each list box contains a list of nonterminal or lake symbols. Each list box corresponds to a substring of the text that the user selected. The user can create a rule by selecting a combination of nonterminal or lake symbols in each list box. For example, the dialog in Figure 4.5 was shown when the user selected "onlyWhen(fooIsNeeded()));" in line 3 in Listing 4.2. This character string consists of three islands: `onlyWhen`, `group`, and `semi`, that we have already

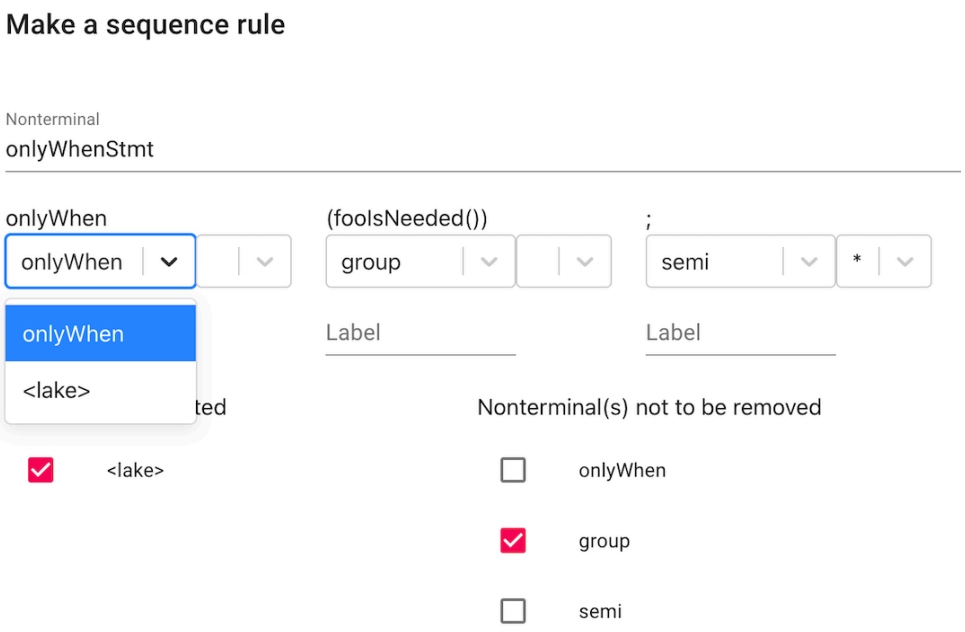

Figure 4.5: Dialog for the sequence operation

made the rule for recognizing them. Hence, PEGSEED suggests the sequence of these nonterminals as default. If the suggested combination of nonterminals is what the user expected, she does not need to change the selection of each list box. However, if it is not what she expected, she can change one or more list boxes. Each list box contains other possible nonterminal or lake that can recognize the corresponding character string. For example, the `onlyWhen` nonterminal is also recogzied as `<lake>` since `onlyWhen` is specified in the right-hand side of the rule for `<lake>`. Hence, the user can select `<lake>` in the list box as shwon in Figure 4.5. If that part does not need to be `onlyWhen`, the user can specify `<lake>` instead of the `onlyWhen` nonterminal. Moreover, the user can append one of the postfix operators to `<lake>` if needed.

The user can also create a new lake by editing the list box directly. In such a case, the right-hand side of the rule for the new lake is copied from the lake listed in the list box.

With the list of choice rules (3), the user specifies the choice rules where the new sequence rule should be added. As with the dialog for the token operation, all choice rules are listed. In the case of the sequence operation, the user specifies new lakes in the list of list boxes (2), the newly created lakes are also listed in the list (3). The user can create a recursion by selecting the nonterminal that is also specified in the list of list boxes (2).

With the list of nonterminals (4), the user can choose whether each nonterminal specified in the list of list boxes (2) should be removed from the choice rule selected in (3). Now, since these nonterminals are part of the new sequence rule, they may not need to be in the choice rules in many cases. An exception is when there is a recursion. For example, the right-hand side of `group` selected in the list of list boxes (2) contains `<lake>` selected in (3). In this case, `group` is recursively defined via `<lake>`. If the group nonterminal is removed from the

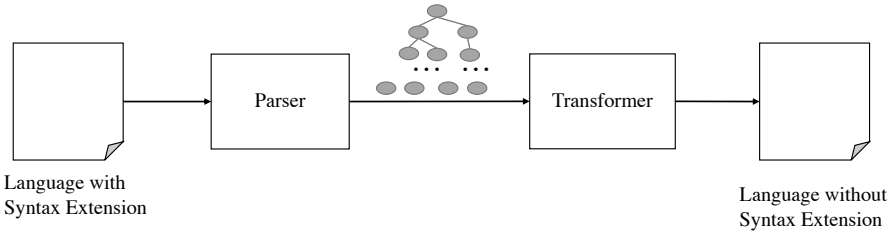

Figure 4.6: Transpiler for syntax extension

rule for `<lake>`, this recursion gets corrupt. Hence, PEGSEED automatically selects only these choice rules as nonterminal not to be removed by default. If the default selection is not what the user expects, the user can change the selection.

Choice Operation

The choice operation provides a way to add a new choice rule corresponding to line 5 in Listing 4.6. The user can apply the choice operation to create a new choice rule. The new rule contains only one nonterminal at first. To add a nonterminal to the rule's right-hand side, the user applies one of the three operations specifying the choice rule to be updated.

The user applies the choice operation by selecting a text area that corresponding to one of the islands and selecting the choice operation. Then, PEGSEED shows the GUI operation similar to Figure 4.5. In contrast to Figure 4.5, the list of list boxes contains only one list box. The user specifies the name of the nonterminal that is the left-hand side of the new rule. Then, the user selects a nonterminal from the list box. Finally, the user selects rules to be updated with the new rule and whether the nonterminal selected in the list box is removed from the rules to be updated.

4.4 Case Study

We applied our PEGSEED to implement a transpiler framework of syntax extension to confirm the effectiveness and usefulness of our proposed approach. In this section, we show the design and implementation of the framework for transpilers by using the `onlyWhen` construct as an example. Then we introduce two other syntax extensions, including `unless` statements and JSX-like extension.

4.4.1 Design and Implementation of Framework for Transpiler

Transpilers are tools for syntax extension. As shown in Figure 4.6, they take a program written in a language with syntax extension and convert it into a program written in the original language. A transpiler consists of two main software components: a parser and a transformer. The parser takes a program in extended language and generates a parse tree based on an input island grammar. The transformer takes the parse tree generated by the parser and traverses the

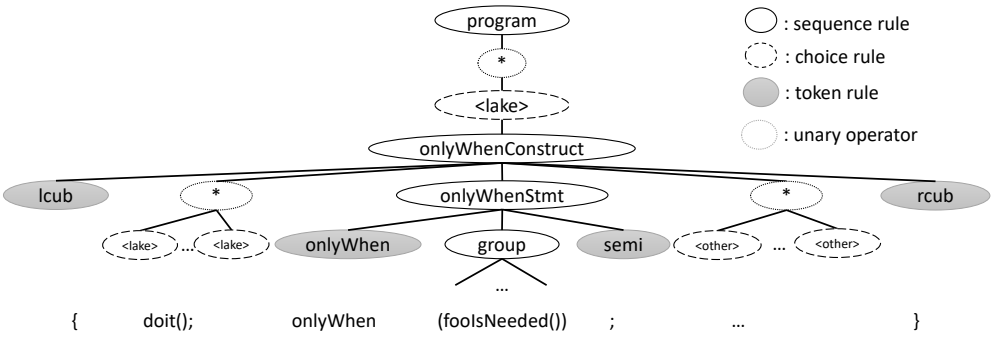

Figure 4.7: Parse tree for the onlyWhen construct

parse tree, and generates an equivalent program in the original language without the syntax extension.

Parse Tree

An example parse tree generated by the parser is shown in Figure 4.7. This parse tree is generated from the program in Listing 4.2. Rules in the grammar generated by PEGSEED are in one of three types: token rule, sequence rule, or choice rule, as shown in Listing 4.6. Each type of rule is encoded in a node or a leaf in a parse tree. The nonterminal symbol and lake symbol are used as a node label. For example, the node labeled with `onlyWhenConstruct` corresponds to the following sequence rule:

```
onlyWhenConstruct <- lcub <lake>* onlyWhenStmt <other>*
   rcub
```

Operands of sequence such as `lcub` or `<lake>*` are children of the node labeled with `onlyWhenConstruct`. The leftmost child of the node labeled with `onlyWhenConstruct` is the leaf labeled with `lcub`. The rule for `lcub` is a token rule, which is always encoded into a leaf. The next child of the `onlyWhenConstruct` node is the node labeled with `*`. PEG's unary operators: `*`, `+` and `?` are encoded into nodes labeled with them. These nodes have zero or more children corresponding to their operands. The node labeled with `*` has multiple children labeled with `<lake>`. The node labeled with `<lake>` corresponds to a choice rule. Nodes for choice rules have zero or only one child labeled with a nonterminal that recognizes the corresponding part of the input text. When the lake symbol works as just a wildcard character, the node corresponding to the lake symbol is a leaf with no child.

Transformation

The second part of the transpiler is a transformer. The transformer takes a parse tree generated by the parser and rewrites the input program by using the parse tree. The user specifies the rewriting rule for the extended constructs.

Our framework generates not only a parser but also most parts of a transformer written in TypeScript on behalf of the user. The user can implement the transformer just by writing a subclass that inherits the class generated by

Listing 4.7: Default transformer for the `onlyWhen` construct

```
1 class Transformer {
2   ...
3   onlyWhenConsturct(node: Node_onlyWhenConstruct): string {
4     return text(node.lcub) + text(node.lake) + text(node.
          onlyWhenStmt) + text(node.other) + text(node.lcub);
5   }
6   ...
7 }
```

Listing 4.8: Class definition for the node of the onlyWhen construct

```
1 class Node_onlyWhenConstruct extends Node{
2   lcub: Leaf;
3   lake: Node[];
4   onlyWhenStmt: Node_onlyWhenStmt;
5   other: Node[];
6   rcub: Leaf;
7 }
```

our framework. This superclass implements default methods for every sequence rule in an island grammar edited with PEGSEED. The generated transformer traverses a given parse tree in post-order and applies a method corresponding to the label of the visited node to convert the parse tree into a native program in plain text. Each method returns a text representation of the corresponding subtree. The default methods in the generated class do not modify the original input text at all. For example, the superclass generated by the grammar in Listing 4.5 is shown in Listing 4.7. The `Transformer` class implements default methods for each sequence rule in the grammar. For example, the `onlyWhenConstruct` method corresponds to the rule for the `onlyWhenConstruct` nonterminal. The `onlyWhenConstruct` method takes a parameter whose type is `Node_onlyWhenConstruct`. The definition of `Node_onlyWhenConstruct` is shown in Listing 4.8. This class is also generated by our framework from the grammar. Each operand of PEG's sequence operator becomes a property of the class. These properties are children of the node of the `Node_onlyWhenConstruct`. The default method for `onlyWhenConstructNode` concatenates all the transformation results for each child and returns it. The `text` method in Listing 4.7 returns the text representation of the given argument. The text returned by the `text` method is the original text or a text modified by the user-defined method.

To rewrite the `onlyWhen` construct, the user writes a class that inherits the `Transformer` class as shown in Listing 4.9. The `onlyWhenTransformer` class extends `Transformer` generated by PEGSEED and overrides only the `onlyhenConstruct` method. Lines 3–9 are a string interpolation that creates rewritten text. The `onlyWhenConstruct` method converts a tree of the `onlyWhen` construct into a block with an if statement. For example, when the input is the program in Listing 4.2, the method can generate the program written in Listing 4.1. In this case, `text(node.lake)` return `doit();` in Listing 4.2. Similary `text(node.onlyWhenStmt.group)` returns `(fooIsNeeded()))`.

Listing 4.9: User defined transformer for the `onlyWhen` construct

```
1  class onlyWhenTransformer extends Transformer {
2    onlyWhenConsturct(node: Node_onlyWhenConstruct): string {
3      return `
4          {
5              ${text(node.lake)};
6              if ${text(node.onlyWhenStmt.group)} {
7                  ${text(node.other)}
8              }
9          }
10         `;
11   }
12 }
```

4.4.2 Other Syntax Extensions

`unless` statement

Suppose that `unless` statements are essential for writing a natural program in a target domain when the programming language does not support `unless` statements. Unless statements execute its block statement only when a given condition is not satisfied. An example of an `unless` statement is as follows:

```
unless (status == FAILED) {
    doit();
}
```

In this case, `doit()` is called only when the value of the `status` variable status is `FAILED`. Syntax extension enables the embedded DSL support `unless` statements like the above one by converting a program in embedded DSL into a pure host language. In the case of the above example, the resulting program must be as follows:

```
if (!(status == FAILED)) {
    doit();
}
```

We created an island grammar for converting `unless` statements into `if` statements. In this case, we only need the leading part of the `unless` statement. For example, we need to extract `unless (status == FAILED)` as an island. We created the grammar in Listing 4.10 with only GUI operations provided by PEGSEED. The rules in lines 4–8 are created by GUI operations. We commented on each rule with the name of the corresponding operation. Five operations were required to create the grammar. The last rule in line 8 recognizes the leading part of an `unless` statement.

We also developed a transpiler that handles `unless` statements as shown in Listing 4.11. The `UnlessTransformer` implments the `Transformer` class which our framework generated from the grammar in Listing 4.10. It overrides only the `unless` method declared in its superclass. The `unless` method returns the leading part of an if statement whose condition is inversion of the condition in the `unless` statement.

Listing 4.10: Generated grammar for the `unless` construct

```
1 program <- <lake>*
2 <lake> <- unless_cond / group / string
3 string <- r'"([^"]|\")*?"\s*'
4 unless <- r'(?<!\w)unless(?!\w)\s*'
5 lpar <- r'\(\s*' // token
6 rpar <- r'\)\s*' // token
7 group <- lpar <lake>* rpar // sequence
8 unless_cond <- unless group // sequence
```

Listing 4.11: User defined transformer for the `unless` construct

```
1 class UnlessTransformer extends Transformer {
2   unless(node: Node_unless_cond): string {
3     return `if (!${node.group.lake})`;
4   }
5 }
```

JSX like statement

Embedding XML-like syntax into a host language is a successful application of syntax extension. For example, JSX is an extended language based on JavaScript and supports programming constructs similar to HTML tags. Listing 4.12 is an example of a JSX program quoted from the JSX web page [1]. The programming construct in lines 1–3 is an expression in JSX and converted to the program in Listring 4.13 by the JSX transpiler.

We implemented a transpiler for a subset of JSX with our framework such that it can convert the example program. We used the program in Listing 4.12 as an example text and created the grammar in Listing 4.14 with only GUI operations provided by PEGSEED. There are 16 rules, of which the last 13 rules are created by GUI operations. We commented on each line with the name of the corresponding operation. For example, the rule in line 4 was created by the token operation. We used seven token operations, four sequence operations, and two choice operations for the grammar in Listing 4.14. The last rule of which the left-hand side is the `element` nonterminal, which recognizes the extended program construct for JSX.

To implement a transpiler for the JSX construct, we wrote the program as shown in Listing 4.15. The `JSXTransformer` class only implements the `element` method declared in the `Transformer`, which is generated from the grammar in Listing 4.14. This method converts the JSX construct into the call of the `createElement` method on the `React` class.

Listing 4.12: Example of a JSX program

```
1 <MyButton color="blue" shadowSize={2}>
2   Click Me
3 </MyButton>
```

[1] `https://reactjs.org/docs/jsx-in-depth.html`

Listing 4.13: Generated JavaScript program from the JSX program

```
1  React.createElement(
2    MyButton,
3    {color: 'blue', shadowSize: 2},
4    'Click Me'
5  )
```

Listing 4.14: Generated Grammar for JSX extension

```
1  program <- <lake>*
2  <lake> <- element / block / string
3  string <- r'"([^"]|\")*?"\s*'
4  lcub <- r'\{\s*' // token
5  rcub <- r'\}\s*'  // token
6  block <- lcub <lake>* rcub // sequence
7  value <- string_value / block // choice
8  string_value <- string // choice
9  equals <- r'=\s*' // token
10 id <- r'(?<!\w)\w+(?!\w)\s*' // token
11 attr <- id equals value // sequence
12 attrs <- attr+ // sequence
13 lt <- r'<\s*' // token
14 ltsol <- r'<\/\s*' // token
15 gt <- r'>\s*' // token
16 element <- lt tag:id attrs* gt <lake>* ltsol tag:id gt  //
      sequence
```

Listing 4.15: User defined transformer for JSX extension

```
1  class JSXTransformer extends Transformer {
2    element(node: Node_element): string {
3      const args = node.attribute
4        .map((attr) => {
5          const value =
6            attr.value instanceof Node_block
7              ? text(attr.value.lake)
8              : text(attr.value);
9          return `${attr.SYMBOL}: ${value}`;
10       })
11       .join(', ');
12
13     return `
14     React.createElement(
15       ${text(node.tag)},
16       {${args}},
17       '${text(node.lake)}'
18     )
19     `;
20   }
21 }
```

4.5 Related Work

4.5.1 Parser Generators

A number of parser generators such as Yacc have been developed. They generate a parser from the given grammar definition. Their drawback is that the users need to learn several tricky techniques to write a grammar that the generator can process. For example, when the generator adopts the LR parsing, the users have to learn how to deal with shift-reduce and reduce-reduce conflicts. When the LL parsing or the packrat parsing is used, the users have to avoid left recursion.

ANTLRWorks [13] helps the users understand why the grammar did not appropriately parse the input source program as they intend. Although ANTLRWorks provides graphical user interface as PEGSEED does, the ANTLRWorks users have to directly edit a grammar definition. PEGSEED hides the details of the PEG definition from the users.

The generalized parsers such as Cocke-Younger-Kasami (CYK) [96], GLL [83], and GLR [88] can accept any context-free grammar even when the grammar is ambiguous. This property is similar to PEGSEED's but the user has to implement a post process when the generalized parsing is used. Since the generalized parsers produce several possible parse trees from one input source program, which are often called parse forest, the post process has to select the most appropriate one among them. This is necessary for resolving ambiguity.

4.5.2 Interactive Grammar Construction

Interactive grammar construction addresses the difficulty in writing a grammar definition. Crespi *et al.* [21] proposed an algorithm for generating a subset of operator precedence grammar from valid statements interactively given by the user as examples.

Parsify [61] and Parsimony [60] are parser generators with graphical user interface for non-expert users. The users can construct a grammar definition by interactively showing an example of how it is parsed. A difference from PEGSEED is that they require the users to define a fully detailed grammar even when the generated parser is used to recognize only a particularly kind of non-terminal symbols in the input source program.

4.5.3 Grammatical Inference

Grammatical inference is a technique for learning a grammar through examining the sentences in an unknown language. It has been studied for decades, for regular languages [5], reversible languages [4], reversible context-free languages [82], and ultimately context-free languages [59]. Those results have been adopted in the contexts of programming tools and software engineering, such as domain-specific languages (DSL), visual languages, execution traces [86].

The grammatical inference for a dialect of existing programming language has been proposed [27, 3, 25, 24]. These systems infer a grammar from not only example programs but also the grammar of the original language. They only learn differences from the original grammar. Unfortunately their approach is not effective for our aim; we cannot assume that we have the grammar definition of a language similar to the target language.

AUTOGRAM [46, 47] can infer a context free grammar by examining not only valid program code but also syntactically-incorrect code in the target language. Given a set of sample code, AUTOGRAM uses dynamic tainting to trace the data flow of each input character. It reflects the data flow on the grammar inference.

The grammatical inference for DSL has been studied. An evolutional algorithm to infer grammars for DSLs have been developed by Crepinsek *et al.* [91] and Javed *et al.* [48]. PAX [97] can infer a pattern language specification for generating the parser for it from example programs.

4.5.4 Programming by Examples

PEGSEED can be regarded as a system based on programming by example (PBE). The PBE technique has been studied broadly in software engineering [22, 40, 77]. The examples include the synthesis of string transformation [6, 40, 64], spreadsheet manipulation [10, 42], number transformation [85], and data extraction from unstructured or semi-structured data [58].

The LAPIS [65] system can highlight code fragments on the editor pane

Some systems are similar to PEGSEED in that they are GUI-based interactive systems. The STEPS [95] system supports text highlighting like PEGSEED, but toward the narrower aim of generating text transforms like those performed by short shell scripts.

4.6 Summary

In this chapter, we proposed an interactive approach for island grammars. With this approach, the user adds rules into grammar incrementally. In each step, the user adds a new rule that recognizes a small island. Rules added in each step recognize a new token or sequence of already defined islands and lakes. After adding a new rule, the user can immediately test the rule by highlighting text areas corresponding to islands recognized by the rule in an example text.

We also proposed GUI operations to edit island grammar. The user can add a new rule by selecting a text area and apply one of the GUI operations. Each GUI operation creates a new rule via a dialog with the user. In our case study on syntax extension, including the `onlyWhen` construct , the `unless` statement, and the JSX like construct. The grammars for these syntax extensions were created only with GUI operations.

Chapter 5

Dynamic Domain-Specific Assistance

5.1 Introduction

Domain-specific languages (DSLs) are widely used in many domains to develop software efficiently for a particular use. Because DSL users can concentrate on the domain-specific problem, the development in DSLs is more efficient than in general-purpose programming languages. One successful application of DSLs is processor specification to generate a processor emulator or to design a processor. The DSLs for this purpose are called processor description languages (PDLs) or architecture description languages [66]. Several PDLs, including nML [43], ISDL [41], LISA [75], EXPRESSION [80], Harmless [49], and HPADL [51], have been developed and used in academia and industry.

Some PDLs are implemented as embedded DSLs, which are libraries or frameworks in their host languages. Examples include Pydgin [62] hosted in Python and ArchC [8] hosted in SystemC [73]. An advantage of embedded DSLs is that they can borrow the host language's tools, including their integrated development environments (IDEs), thereby reducing the development cost of DSLs. However, the programming assistance by IDEs is not satisfactory; more domain-specific assistance for convenience and correctness should be provided.

In this chapter, we present a design approach to the embedded DSLs enabling domain-specific programming assistance by their host's IDEs. The DSLs are carefully designed to let the IDEs provide better auto-completion and error detection by domain-specific knowledge. To this end, the DSL compiler/runtime generates support programs while the users write a DSL program. The DSL program is split into multiple components for different concerns; the DSL compiler reads a component written earlier and generates a support program for later components. The domain-specific knowledge is encoded into the support program, and the IDE refers to this support program for providing domain-specific assistance when other components are written. An example of the support program is the superclass of the class written by the DSL user.

The contributions of this work are three-fold:

- We reveal that domain-specific programming assistance by IDEs is poor for embedded DSLs and present our approach to address this problem. Our approach exploits program generation such that programming assistance for the host language can be regarded as domain-specific.
- We conduct a case study by developing a practical embedded DSL called

MELTRANS and demonstrate the domain-specific assistance that is available when the user is writing a program in MELTRANS.

- We experimentally confirm that the domain-specific assistance provided by MELTRANS can effectively reduce the amount of code to be written by the user, the generated emulator can emulate a processor in over 1,000 MIPS, and MELTRANS is general enough to generate an emulator for several commercial instruction sets (ARM, MIPS64, SH, RH850, RISC-V, and RX).

In the rest of this chapter, we first reveal that domain-specific programming assistance by IDEs is inadequate for embedded DSL by using a processor description language as an example. We then propose a design approach for developing an embedded DSL with domain-specific programming assistance. We also present our experiments and related work. Finally, we conclude this chapter.

5.2 Motivating Problem

The growing adoption of cross-platform virtualization and the rise in instruction set architecture (ISA) diversity are resulting in a need for an efficient method to develop a fast processor emulator. One promising approach is to generate processor emulators using the program written in PDLs. In general, practical processor emulators need to implement dynamic binary translation (DBT) to execute the guest program rapidly. The implementation of emulators with DBT is complicated without a PDL. The emulator with DBT translates guest instructions into host native instructions at runtime. Without a PDL, the developer of emulators needs to describe how the emulator translates guest instructions into host instructions. The developer is required to have a deep understanding of not only guest instructions but also host instructions. When we use a PDL, the PDL compiler can generate code for the translation on behalf of the developer.

A PDL can be implemented as an embedded DSL. An advantage of this approach is that users can exploit an existing tool-chain for the host language of the embedded DSL. However, a drawback of this approach is that it provides poor domain-specific programming assistance.

Suppose that we describe the `MUL` instruction of the ARM processor in such an embedded DSL. The `MUL` instruction multiplies values in the source registers and stores the result in the destination register. Listing 5.1 is the pseudo-code expressing the semantics of the `MUL` instruction, which is quoted from the ARM reference manual [45]. In a Python-based embedded DSL, the Pydgin PDL [62], the same semantics is implemented by the `execute_mul` function shown in Listing 5.2. This function appears to be very similar to the pseudo-code in Listing 5.1. Thus, the user can describe the `execute_mul` function by mostly copying the pseudo-code in the reference manual. The user does not have to specify how to translate the `MUL` instruction into the host instructions when implementing the processor emulator in Pydgin.

Expressions `inst.cond`, `inst.rm`, and `inst.rn` in Listing 5.2 represent the values of the instruction fields of `MUL`, which are `cond`, `Rm`, and `Rn`, respectively. Figure 5.1 shows the excerpt of the bit encoding of the `MUL` instruction taken from the ARM reference manual. The `MUL` instruction has five instruction fields:

Listing 5.1: MUL instruction in the reference manual (ARM)

```
if condition_passed()
    d = UInt(Rd);
    n = UInt(Rn);
    m = UInt(Rm);
    setflags = (S == '1')
    operands1 = SInt(R[n])
    operands2 = SInt(R[m])
    result = operand1 * operand2
    R[d] = result<31:0>
    if setflags then
       APSR.N = result<31>;
       APSR.Z = IsZeroBit(result);
```

Listing 5.2: MUL instruction in Pydgin (ARM)

```
def execute_mul( s, inst ):
  if condition_passed( s, inst.cond ):
    Rn, Rm = s.rf[ inst.rn ], s.rf[ inst.rm ]
    result = trim_32(Rn * Rm)
    s.rf[ inst.rd ] = result

    if inst.S:
      s.N = (result >> 31)&1
      s.Z = result == 0

    if inst.rd == 15:
      return
  s.rf[PC] = s.fetch_pc() + 4
```

`cond`, `S`, `Rd`, `Rm`, and `Rn`. Expressions `inst.cond`, `inst.rm`, and `inst.rn` are method calls on `inst`, which is a function parameter to `execute_mull`, and they return the values of those instruction fields.

The type of `inst` is the `Instruction` class, which must be defined by the Pydgin user. Listing 5.3 is an example of the `Instruction` class for ARM, which is included in the source tree of Pydgin. We renamed the methods in the class such that the names are consistent with the field names in Figure 5.1. The methods in `Instruction` extract the value of an instruction field from the binary representation of the instruction. The binary representation is available from `self.bits`. For example, method `rd` extracts instruction field `Rd` in bits 16–19.

The Pydgin user can expect that the IDE for Python, which is the host language, reads the definition of the `Instruction` class and provides programming assistance to the user. For example, when the user writes `execute_mul`, auto-completion can be expected. When `inst.` is typed, a modern IDE would show the list of the word candidates that could follow `inst.`, and this list would include `cond`, `rm`, and `rn`.

However, this auto-completion lacks domain-specific assistance. Even if the IDE correctly infers that the type of `inst` is the `Instruction` class, the can-

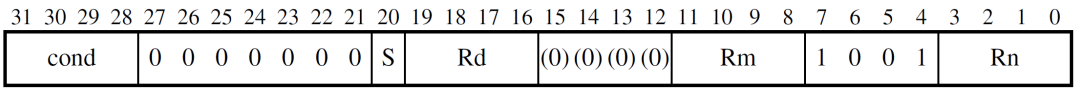

Figure 5.1: Encoding of the MUL instruction (ARM)

Listing 5.3: Definition of the Instruction class

```
1  class Instruction( object ):
2    ...
3    @property
4    def rd( self ): return (self.bits >> 16) & 0xF
5
6    @property
7    def rm( self ): return (self.bits >> 8) & 0xF
8
9    @property
10   def rn( self ): return self.bits & 0xF
11   ...
12   @property
13   def imm24( self ): return self.bits & 0xFFFFFF
14   ...
```

didates for the auto-completion would include `imm24`, which is the method for extracting the `imm24` field of the `B` (branch) instruction shown in Figure 5.2. The reason is that this method is included in the `Instruction` class. In Pydgin, all of the methods for extracting an instruction field are included in the `Instruction` class. Although `imm24` is not available in the body of `execute_mul`, if the user selects `imm24` after typing `inst.`, the IDE would not warn the user of that incorrect selection. Note that the selection is valid from the host-language perspective of Python; it is invalid only from the domain-specific perspective of Pydgin.

To mitigate this problem, some readers might change the design of Pydgin to enable the user to define a different version of the `Instruction` class for a different instruction. For example, the type of `inst` passed to `execute_mul` could be `MulInstruction`, whereas that to `execute_b` could be `BInstruction`. Then, they could provide only the methods available for their instruction. However, this design requires the user to exert extra effort; numerous `Instruction` classes should be defined; thereby complicating the type inference by the IDE. If the host language were statically typed, that design would require the user to pay extra attention to the type of the `inst` parameter.

The domain-specific programming assistance in embedded DSLs is poorer than the assistance in external DSLs. If Pydgin were a standalone external DSL, it would report a compilation error when the user writes `inst.imm24` in the body of the `execute_mul` function. A dedicated IDE for Pydgin would provide better programming assistance to prevent writing `inst.imm24` mistakenly in the body of `execute_mul`.

Developing a dedicated IDE for an embedded DSL might be another option to mitigate the poor domain-specific assistance. Developing an external program-analysis tool might be another option. However, these options decrease the

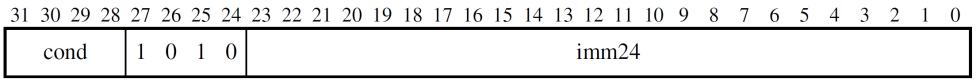

Figure 5.2: Encoding of the B instruction (ARM)

benefit of embedded DSLs; the user would not be able to use the DSL as a library for its host language. The user would be forced to write a program in a particular IDE, which may be unfamiliar. Moreover, developing IDEs for DSLs from scratch is too costly, according to the literature [16].

5.3 MELTRANS: A PDL with Domain-specific Programming Assistance

We present our design approach to an embedded DSL that enables domain-specific programming assistance. Our idea is to let the host-language IDE provide better programming assistance by domain-specific knowledge. To this end, a DSL program is split into multiple components for different concerns. We take advantage of the fact that some concerns contain useful domain-specific knowledge for the description of other concerns. The DSL programming is divided into multiple stages. When the user writes a DSL program in the early stage, the DSL compiler generates support code written in the host language from that user's DSL program. The domain-specific knowledge is encoded in that support code so that the IDE can provide domain-specific assistance for the later-stage DSL programming.

We have developed a PDL called MELTRANS with our design approach. MELTRANS is an embedded DSL hosted by Java for implementing a processor emulator with DBT. The emulator finally generated from a MELTRANS program is written in C++. We chose Java because it is a statically typed language, and its IDEs support rich programming assistance. Its environment-independent specification is also appropriate. For example, an integer value of type `int` is always 32-bit precision. The user of MELTRANS does not have to care about the execution environment to know the size of the integer.

A program in MELTRANS consists of six Java classes, each of which corresponds to a different concern about the description of a processor. There are dependencies among the concerns; some concerns contain domain-specific knowledge used when the user describes other concerns. The language runtime of MELTRANS utilizes these dependencies to provide domain-specific assistance by generating superclasses of the classes. Figure 5.3 shows the classes and concerns, as well as the relations among them. Each concern, except for the format concern, consists of a generated superclass and a user-defined subclass. The arrows are drawn from a concern to the superclass that is generated by the runtime of MELTRANS from the former. Each superclass serves as a canvas that provides auto-completion and limits the risk of error when the user defines the subclass for the concern.

The user defines classes in an order that is based on the dependencies among the concerns. The user starts by defining the format concern of a processor because it does not depend on any concern. For example, when the user develops a

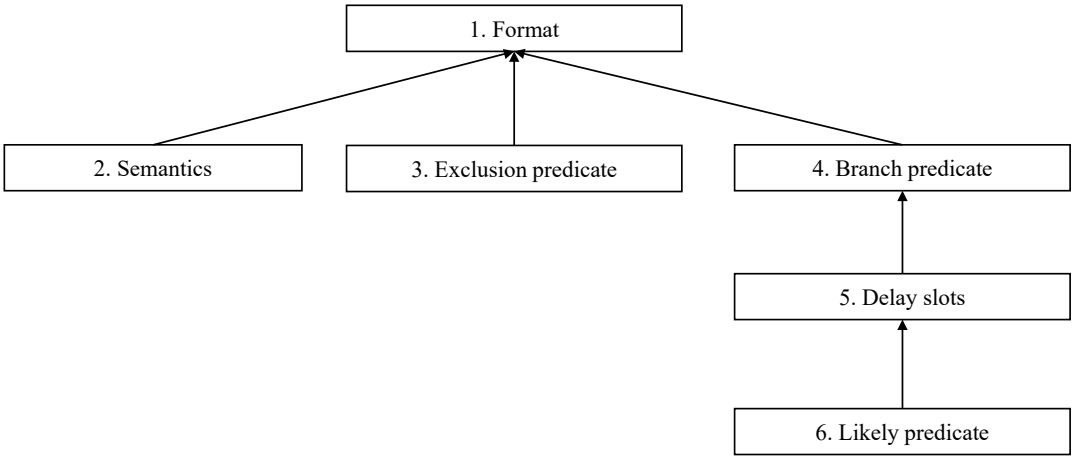

Figure 5.3: Structure of a program written in MELTRANS

processor emulator for the ARM instruction set, the user starts by defining the class named `ArmFormat`. In MELTRANS, there is a naming convention in which all the class names start with the ISA name, such as ARM, and the concern name follows the ISA name. The definition of the format concern enables the language runtime of MELTRANS to generate superclasses for a concern, of which subclass the user then writes. The generated superclasses are named `ArmSemanticsBase`, `ArmExclusionBase`, and `ArmBranchPredicateBase`. In MELTRANS, the name of a superclass starts with the name of its subclass and ends with `Base` according to the convention. The domain-specific knowledge about the instruction format is encoded in the methods in the generated superclasses so that the IDE can exploit them for programming assistance. Thus, the user can expect domain-specific assistance by the IDE when writing subclasses `ArmSementics`, `ArmExclusion`, and `ArmBranchPredicate`. After the user writes the class for the branch-predicate concern, named `ArmBranchPredicate`, MELTRANS generates a new Java class, `ArmDelaySlotsBase`, from that class. Then, the user writes `ArmDelaySlots`, and the same pattern follows.

5.3.1 Concerns

In this subsection, we present the six concerns in MELTRANS and demonstrate what domain-specific assistance the user can expect when writing each concern, except for the format concern.

Format

The language runtime of MELTRANS uses the domain-specific knowledge written in the class for the format concern to provide programming assistance for writing classes for other concerns. Because the class for the format concern is written first, domain-specific assistance is not available when the user writes this class. To ease the definition of the class with only general-purpose assistance, we designed a mini DSL in the string embedding style for specifying an instruction encoding.

Listing 5.4: Class for the format concern for ARM

```
1 class ArmFormat {
2   // instruction encodings
3   String Inst105_A1_MUL =
4     "31[cond]27[0b0000000]20[S]19[Rd]15[0b0000]11[Rm]7[0
        b1001]3[Rn]";
5   String Inst016_A1_B =
6     "31[cond]27[0b1010]23[imm24]";
7     ...
8 };
```

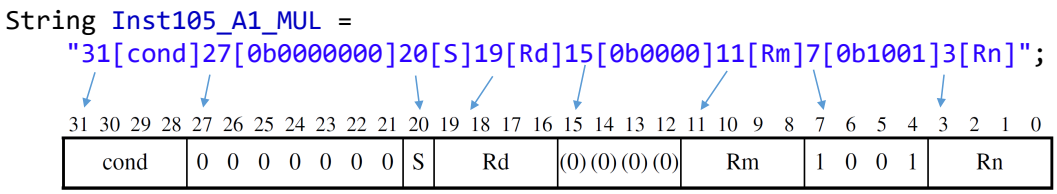

Figure 5.4: String literal for the MUL instruction (ARM)

The class for the format concern describes the bit encoding of every instruction. For each instruction, the class declares a field of `String` type with its initial value, representing the bit encoding written in the mini DSL. The name of this field is used as the identifier of that instruction. For example, the class for the ARM instruction set is shown in Listing 5.4, in which the `Inst105_A1_MUL` field represents the bit encoding of the `MUL` instruction. `Inst105` denotes its section number in the ISA manual, and `A1` indicates that the instruction is an ARM instruction and not a Thumb instruction.

The user can easily write the string literal in our mini DSL by almost copying the description of the instruction in the ISA manual. For example, Figure 5.4 shows the string literal for the `MUL` instruction and its format description found in the ISA manual for ARM. The string literal is read as follows. The bits from the 31st to (but excluding) the 27th are used for the instruction field named `cond`, bits from the 27th up to the 20th must be `0000000` (`0b` is the prefix for binary numbers), the 20th bit is instruction field `S`, and so on.

Semantics

The class for the semantics concern must declare the methods specifying the behavior of every instruction. The name of the method for each instruction must be identical to the name of the corresponding field in the class for the format concern. For example, Listing 5.5 shows the class for the semantics concern for ARM. In Listing 5.5, the method name for `MUL` is `Inst105_A1_MUL`, whereas that for `B` is `Inst016_A1_B`. These names are found in the `ArmFormat` class that the user wrote for the format concern. The parameters to the methods are the instruction fields of each instruction. They are also defined in the class for the format concern. For example, the parameters to the `Inst105_A1_MUL` method are `cond`, `S`, `Rd`, `Rm`, and `Rn`, and they are defined in the string literal given to

Listing 5.5: Class for the semantics concern for ARM

```
1  class ArmSemantics extends ArmSemanticsBase {
2
3    @Override
4    void Inst105_A1_MUL(int cond, int S, int Rd, int Rm, int
        Rn) {
5      if (isConditionPassed(cond)) {
6        int result = reg[Rn] * reg[Rm];
7        reg[Rd] = result;
8        if (S == 1) {
9          cpsr.N = extract(result, 31);
10         cpsr.Z = (result == 0) ? 1 : 0;
11       }
12     }
13   }
14
15   @Override
16   void Inst016_A1_B(int cond, int imm24) {
17     if (isConditionPassed(cond)) {
18       int imm32 = signExtend(imm24 << 2, 25);
19       setNextPc(getPc() + 8 + imm32);
20     }
21   }
22       ...
23 }
```

the `Inst105_A1_MUL` field in the `ArmFormat` class.

The bodies of those methods can be written by almost copying the corresponding description in the ISA manual. For example, the body of `Inst105_A1_MUL` is fairly identical to the description of the behavior of `MUL` in the ISA manual shown in Listing 5.1. It multiplies the `Rn` register by the `Rm` register and stores the result into the `Rd` register. Then it updates the current program status register.

The user can expect domain-specific programming assistance by the IDE when writing the class for the semantics concern. First, all of the methods that the user must write when specifying the instruction's behavior are already declared in the superclass generated by MELTRANS, although their bodies are empty. For example, superclass `ArmSemanticsBase` declares the `Inst105_A1_MUL` and `Inst016_A1_B` methods. The `ArmSemantics` class, which the user must write, overrides them. Thus, when the user types the first few letters of a method name, the IDE shows the candidate list of methods for auto-completion, as illustrated in Figure 5.5. The user can simply choose one of them to obtain a skeleton of a method declaration. In Listing 5.5, lines 3, 4, and 13 can be automatically completed when the user writes the `Inst105_A1_MUL` method.

Second, the instruction fields are encoded into the parameters to the method specifying the instruction's behavior. This also improves the domain-specific assistance by the IDE. For example, the IDE reports an error when the user writes the name of an instruction field unavailable for the instruction that is being written by the user. Figure 5.6 shows an error message that is reported when the user

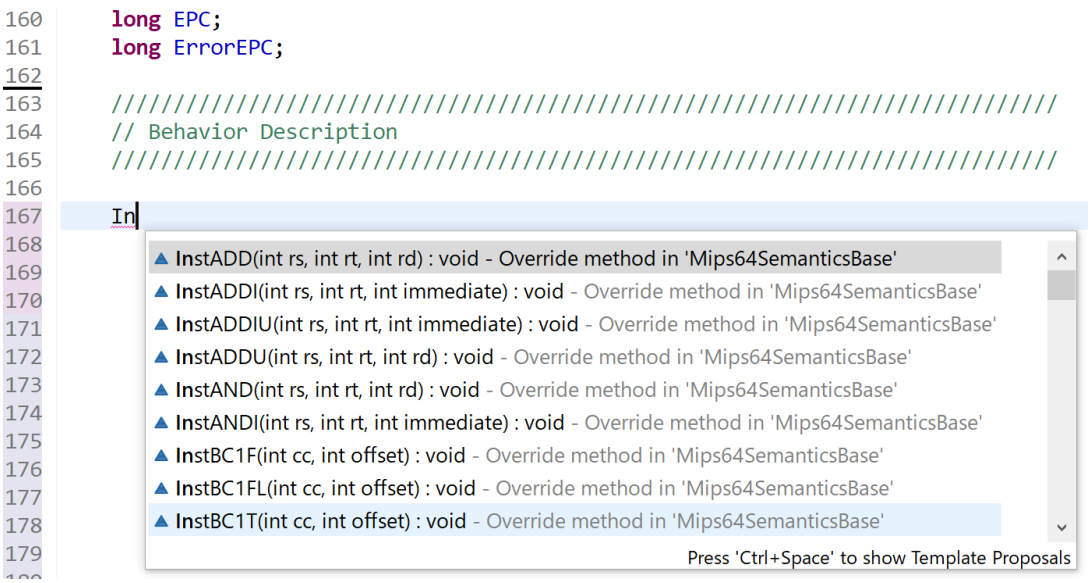

Figure 5.5: Auto-completion of instruction names

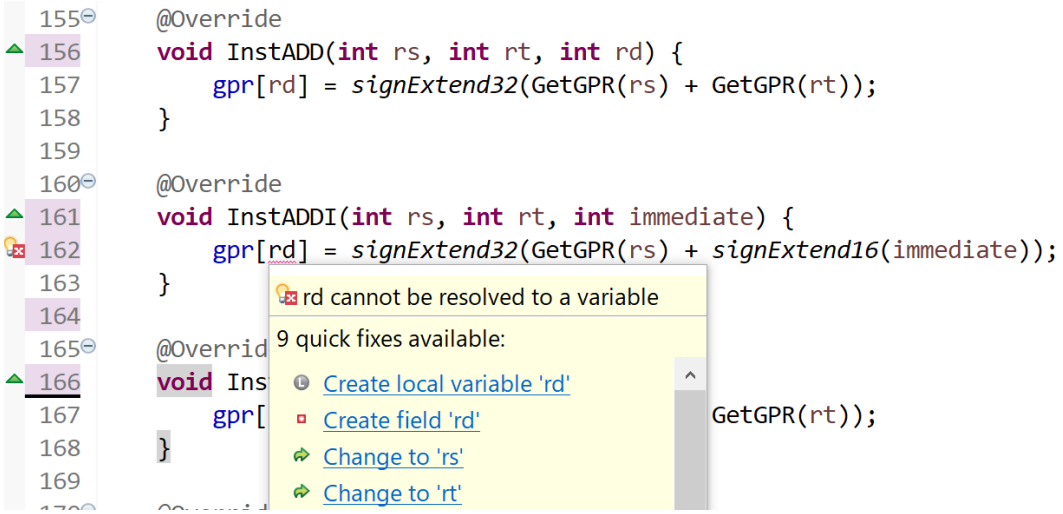

Figure 5.6: Instruction field `rd` is not available for `ADDI`

attempts to store a value into the `rd` register (the register specified by instruction field `rd`), which is not available for the `ADDI` instruction in MIPS64. This is a typical mistake unless the user carefully reads the ISA manual because most instructions in MIPS64, such as `ADD`, store the arithmetic results into the `rd` register. `ADDI` exceptionally stores the result into the `rt` register. In MELTRANS, this mistake is detected as an error of using an undeclared parameter or variable. The domain-specific knowledge regarding which instruction fields are available is encoded into the method parameters so that the domain-specific assistance will be provided as normal programming assistance in the host language.

This design of MELTRANS also helps the IDE to detect another typical mistake. Although the result of `ADD` in MIPS64 is stored in the `rd` register, the user might misinterpret that `ADD` is a double-operand instruction and store the result into the `rt` register (the second operand). The IDE may detect this

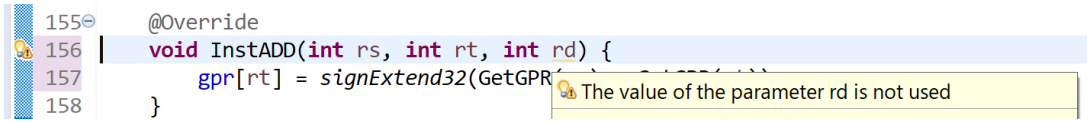

Figure 5.7: Result of `ADD` is stored in an incorrect register

mistake as an error because the third method parameter, `rd`, is never used in the method body of `InstADD` (Figure 5.7). When the instruction is executed, all of the instruction fields should be used. MELTRANS exploits this fact and thus passes them through the method parameters to the method implementing the instruction's behavior, such as `InstADD`.

Exclusion Predicate

The class for the exclusion-predicate concern describes how to disambiguate the instructions that share the same opcode. This is necessary for certain ISAs such as ARM and RH850.

Suppose that we implement an emulator for the RH850 instruction set. The class for the format concern would include the following field declarations for the `DIVH` and `RIE` instructions:

```
String Inst028_DIVH = "15[r]10[0b000010]4[R]";
String Inst086_RIE = "15[0b0000000001000000]";
```

When both the instruction fields, `r` and `R`, in `DIVH` are `00000`, we cannot distinguish `DIVH` and `RIE` because their bit patterns are identical. For disambiguation, the ISA manual for RH850 specifies that instruction field `r` or `R` in the `DIVH` instruction must not be zero. We call this the exclusion predicate of the `DIVH` instruction. In the class for the exclusion-predicate concern for RH850, the exclusion predicate of the `DIVH` is written as follows:

```
@Override
boolean Inst028_DIVH(int r, int R) {
    return r == 0 | R == 0;
}
```

The `Inst028_DIVH` method returns true when the given instruction field, `r` or `R`, is zero; hence, it is not valid. If it returns true, then the current instruction word is not that of `DIVH`, but that of `RIE`.

The user can expect domain-specific programming assistance when writing this class. This user-defined class must inherit from the superclass generated from the class for the format concern. The superclass declares all of the methods that the user may declare in its subclass. These methods return false. Therefore, the user can expect domain-specific assistance similar to the assistance that can be expected for the semantics concern.

Branch Predicate

The class for the branch-predicate concern describes which instructions are branch instructions. It declares a method for every instruction, and the method returns true if the instruction is a branch.

For example, the `B` instruction in ARM is a branch instruction. The user can specify this by defining a method in the class for the branch-predicate concern as follows:

```
@Override
boolean Inst016_A1_B() {
    return true;
}
```

The return value indicates that the `B` instruction is a branch instruction.

For some instructions, its instruction fields determine whether the instruction is a branch. For example, the `ADD` instruction in ARM is a branch instruction when instruction field `Rd` (destination register) is the program counter. Otherwise, `ADD` is not a branch instruction. To support this, the class for the branch-predicate concern can declare a method taking all the instruction fields. Hence, the branch predicate for the `ADD` is written as follows:

```
@Override
boolean Inst005_A1_ADD(int cond, int S, int Rn, int Rd, int
    imm12) {
    return Rd == 15;
}
```

The expression in the return statement indicates that the `ADD` instruction is a branch instruction when `Rd` is the program counter (register 15).

The user can expect domain-specific programming assistance when writing the class for the branch-predicate concern. This class must inherit from the superclass generated by the language runtime of MELTRANS from the class written for the format concern. The generated class declares two methods for every instruction. One takes no parameter while the other takes all the instruction fields as parameters. Because the methods in the superclass return false, the subclass does not need to declare the methods for instructions that are always non-branch instructions. When the subclass needs to declare the method, the user can use auto-completion by the IDE to select one method for each branch instruction.

Delay Slots

The emulators for some ISAs must consider delay slots for each branch instruction. In MELTRANS, the user can describe the delay slots by writing a class for the delay-slots concern. This class declares a method for the branch instruction with delay slots; the method returns the number of delay slots.

For example, the `BFS` instruction in the SH instruction set is a branch instruction with a delay slot. The user can specify this by writing a method in the class for the delay-slots concern as follows:

```
@Override
int InstBFS() {
    return 1;
}
```

The return value indicates that the `BFS` instruction has one delay slot.

The user can also expect domain-specific programming assistance when writing a class for the delay-slots concern. The user-defined class inherits from the

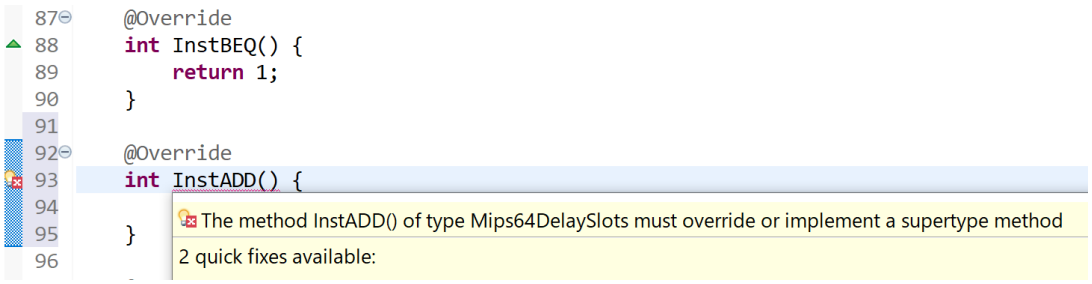

Figure 5.8: No overridden method in the superclass because the method is not for a branch instruction

superclass generated by MELTRANS from the class for the branch-predicate concern. The superclass declares a method for every branch instruction. Because the method in the superclass returns 0, its subclass can declare only the methods for the branch instructions with more than zero delay slots.

Note that the superclass declares only the methods for branch instructions and not all instructions. Therefore, the method list for auto-completion is more accurate. Even if the user declares a method for a non-branch instruction, the IDE will report a warning message because the method does not override any method in the superclass (Figure 5.8). To determine which instruction is a branch one, MELTRANS investigates the bodies of the methods in the class for the branch-predicate concern. If the method may return true, MELTRANS considers the corresponding instruction as a branch one.

Likely Predicate

The class for the branch-likely concern declares a method for every branch-likely instruction. The method must return true.

Some ISAs have branch-likely instructions. They skip the execution of the following instructions in the delay slots when the branch is not taken. From the class for the delay-slots concern, MELTRANS generates the superclass of the class for the likely-predicate concern. The superclass declares a method for every branch instruction with delay slots. This method returns false because MELTRANS assumes that all branch instructions are not branch-likely instructions by default. Hence, the subclass needs to declare only a method for branch-likely instructions. The generated superclass enables domain-specific assistance similar to the assistance for the delay-slots concern.

5.3.2 Generated Processor Emulator

To explain how the six concerns contribute to the DBT, we show our ISA-independent skeleton of DBT in Algorithm 4. The name of the concern denotes that the operation in its source line depends on that concern. Algorithm 4 dynamically translates the guest instructions in a basic block (BB) into the LLVM [56] intermediate representation (IR). Once the LLVM IR is available, the generated emulator uses the LLVM JIT engine to generate the host native code. Algorithm 4 is based on the strategy pattern [34] win which ISA-dependent parts are implemented in the strategy objects. The language runtime of MELTRANS

Algorithm 4 Translation of a basic block

Input: memory model `memory`, start address of BB `addr`, strategy object `isaStrategy`
Output: llvm IR corresponding to BB

```
1: repeat
2:     iword ← fetchInstruction(memory, addr)
3:     (inst, fields) ← isaStrategy.decode(iword)
4:     inst.generateIr(fields)                          ▷ semantics
5:     addr ← addr + inst.length                        ▷ format
6: until inst.isBranch(fields)                          ▷ branch predicate
7: if inst.delaySlots ¿ 0 then                          ▷ delay slots
8:     generateDelaySlotsIr(memory, addr, inst)         ▷ delay slots and likely predicate
```

Table 5.1: Comparison of code metrics among PDLs (ARM)

PDL	Concern	Instructions	LOC	%	LOC per instruction	Completed LOC	Completed LOC (%)	Methods	Complexity
MELTRANS	Format	173	180	5.87	1.04	0	0.00	1	1.00
	Semantics	173	1,953	63.64	11.29	519	26.57	191	2.65
	Exclusion predicate	173	695	22.65	4.02	519	74.68	173	1.00
	Branch predicate	173	235	7.66	1.36	174	74.04	58	1.00
	All	173	3,069	100.00	17.74	1212	39.49	423	1.75
Pydgin	-	62	1003	100.00	16.18	0	0.00	116	3.1

generates ISA-dependent parts from the six concerns written in MELTRANS.

Algorithm 4 takes the memory model `memory`, the start address of a BB `addr`, and a strategy object `isaStrategy` as parameters, and it generates the LLVM IR that emulates the behavior of the instructions in the BB. In lines 1–6, the emulator translates a guest BB into the LLVM IR. For each iteration in lines 2–5, a single guest instruction is translated into the LLVM IR instructions, which are appended to the resulting LLVM IR. In line 3, the `decode` method is called with arguments `isaStrategy` and `iword` to decode instruction `iword` in the ISA-specific way. The `decode` method returns the identified instruction and its instruction fields as `inst` and `fields`, respectively. The implementation of `decode` can be generated from the format and exclusion-predicate concerns using the algorithm proposed in [71]. In line 4, the emulator generates the LLVM IR code according to the identified instruction and its instruction fields. The LLVM IR for the instruction is generated by the AST of the method for the instruction in the semantic concern. We use the deep reification approach proposed in [17] to obtain ASTs from a program written in MELTRANS. In line 5, the emulator increments address `addr` such that it points to the next instruction. The length of the instruction is retrieved from the format concern. In line 6, the emulator checks whether the decoded instruction is a branch instruction. If it is a branch instruction, the emulator terminates the iteration; otherwise, the emulator goes back to line 2. In lines 7–8, the emulator translates the instructions in the delay slots immediately after the branch, if any. The number of delay slots is obtained from the delay-slots concern.

5.4 Experimental Results

To validate our design, we implemented several processor emulators with MELTRANS and conducted experiments using them.

5.4.1 Amount of Code to Be Written

To determine whether the domain-specific assistance is achieved without increasing the amount of code to be written by the user, we compared MELTRANS with Pydgin in terms of the code metrics of their programs for ARM. We used Eclipse as an IDE in this experiment.

The results are summarized in Table 5.1. For example, the first row in Table 5.1 shows that the format concern implements 173 instructions, and its code accounts for 5.87% of the entire code; the lines of code (LOC) excluding comments and the blank lines is 180, LOC per instruction is 1.04, and LOC automatically completed by the IDE is zero and accounts for 0% of the format concern. The format concern includes one method, and its cyclomatic complexity [37] is 1.0 on average.

The total LOC per instruction for ARM in MELTRANS is 17.74. This result is comparable with the LOC per instruction of 16.18 for ARM in Pydgin. MELTRANS requires an additional 1–2 LOC per instruction compared with Pydgin for ARM. This difference appears to be caused by the difference between the host languages. In general, Java is more verbose than Python.

In MELTRANS, 39.49% of the code could be automatically completed by the IDE. Although auto-completion may be available in each line of the program, we counted only lines for the method templates completed by the IDE, which consisted of method signatures with the `@Override` annotation and a pair of opening and closing braces. If we remove the automatically completed lines from the total amount of code, the remaining LOC becomes 1,857, and the LOC per instruction becomes 10.73. The result shows that our domain-specific assistance can effectively reduce the amount of code to be written by the user.

5.4.2 Performance

To determine whether the generated processor emulators run at practical speeds, we compared the emulator for ARM generated by MELTRANS with state-of-the-art QEMU and the emulator generated by Pydgin in terms of their simulation speed. We used benchmark programs from the EEMBC [78] Autobench benchmark suite, which is one of the de-facto industrial standard benchmarks for comparing embedded processors. We performed all of the measurements presented in this chapter on a Linux-based desktop machine with a 64-bit Core i7 7700T at 2.9 GHz, disabling Turbo Boost, and a 16-GB main memory. We built gcc 6.1.0 and used it with the -O2 flag to cross-compile the benchmark programs. We also used gcc 7.5.0, which is the default compiler in the host operating system, to build the processor emulators. The generated emulators used the JIT engine of LLVM 10.0 to translate LLVM IR into the host AMD64 instructions. Our emulator uses a superblock as a translation unit. A superblock consists of all BBs that are traceable through direct branch instructions, except call instructions when the emulator needs translation.

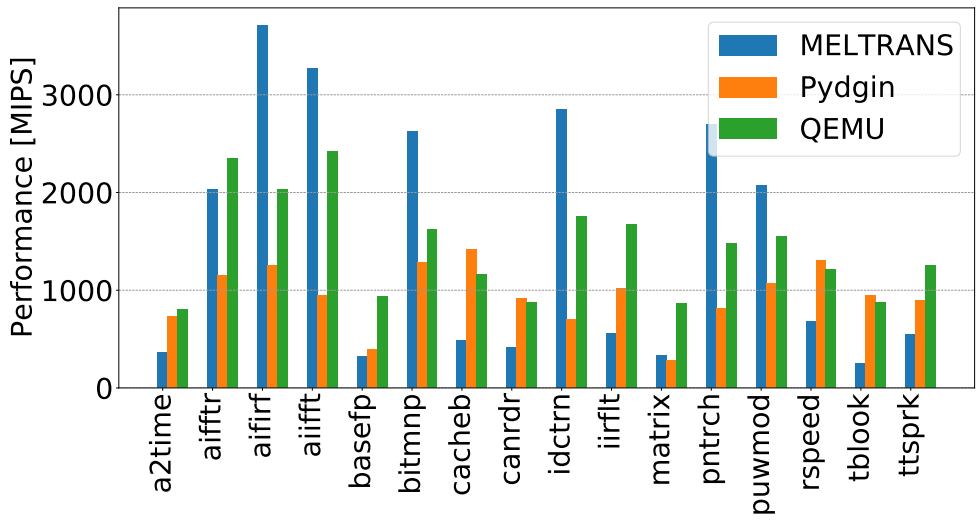

Figure 5.9: Performance of the emulators (ARM)

Figure 5.9 depicts the results. Our emulator achieved 1,449 MIPS on average. A minimum of 253 MIPS was observed when the emulator ran tblook, and the maximum was 3,704 MIPS when it ran aifirf. It appeared that the emulator runs fast when highly executed superblocks contain loops. In such a case, the superblock was well optimized by LLVM.

Our emulator outperformed the other two emulators in seven of the 16 programs. On average, QEMU and Pydgin executed the programs in 1,430 and 945 MIPS, respectively. Because all of these emulators use different translation strategies, the performance tendencies in the benchmark programs appeared to be different among the emulators.

5.4.3 Generality

To determine whether MELTRANS is general enough to generate emulators for multiple ISAs, we implemented MIPS64, RH850, SH, RISC-V, and RX, which are widely used in industry in addition to ARM, and compared the code metrics of the programs. Six concerns were used to write these programs. The results are listed in Table 5.2. For example, the first row in Table 5.2 shows that the program for ARM consists of concerns 1–4 and implements 173 instructions, its LOC is 3,069, the LOC per instruction is 17.74, and the IDE automatically completes 39.49% of the LOC. Each number of a concern corresponds to the number shown in Figure 5.3. The table shows that the IDE can automatically fill 24.17%–49.02% of the LOC. Although the percentage of the code automatically completed by the IDE depends on the complexity of the ISA, all six concerns are sufficient to describe these ISAs. Because other commercial ISAs such as PowerPC and TriCore are similar to these ISAs, it appears that MELTRANS is general enough to describe many practical ISAs.

Table 5.2: Comparison of code metrics among ISAs

ISA	Concerns	Instructions	LOC	LOC per instruction	Completed LOC (%)
ARM	1, 2, 3, 4	173	3,069	17.74	39.49
MIPS64	1, 2, 3, 4, 5, 6	250	2,185	8.74	49.02
RH850	1, 2, 3, 4	223	3,389	15.20	24.17
SH	1, 2, 4, 5	154	1,334	8.66	40.48
RISC-V	1, 2, 4	54	397	7.35	46.85
RX	1, 2, 4	473	3,619	7.65	44.43

5.5 Divide-and-Generate Pattern

We generalized our design approach as the *divide-and-generate* pattern for providing a way to implement code completion and error checking for an embedded DSL. The remaining part of this section explains the pattern in a similar way to the description of "Design Patterns" [34].

5.5.1 Intent

The *divide-and-generate* pattern provides a way to implement code completion and error checking for an embedded DSL by exploiting an existing IDE for the host language.

5.5.2 Motivation

Dynamic programming assistance such as code completion and error checking in editing time is essential for efficient program development. Usually, the programming assistance is provided by an IDE for the language. In the case of embedded DSLs, an IDE for the host language provides them. However, available assistance is limited to general-purpose one, and domain-specific assistance is inadequate for the embedded DSLs.

The solution to this problem is the *divide-and-generate* pattern. Figure 5.10 dipicts the *divide-and-generate* pattern. The DSL developer divides a DSL program into multiple files such that each file addresses a separate concern, and names used for describing a concern is also used for describing another concern. We define that concern A dpends on concern B if names used to describe concern A is provided by B. The DSL runtime generates support code for writing a concern from another concern. In the case of Figure 5.10, while the first concern does not depend on any concern, the second concern depends on the first concern. With this dependency, the language runtime can generate support code for the second concern from the first concern. Similarly, the language runtime can generate support code for the third concern from the second concern.

The available names in a subsequent concern are declared as identifiers in the support code for the concern, which is stored in a separate file from a program written by the user. The names for describing a concern can be mapped to any identifiers in the host language. We can use identifiers such as method names, parameter names, macro names, member names, and so on to encode names

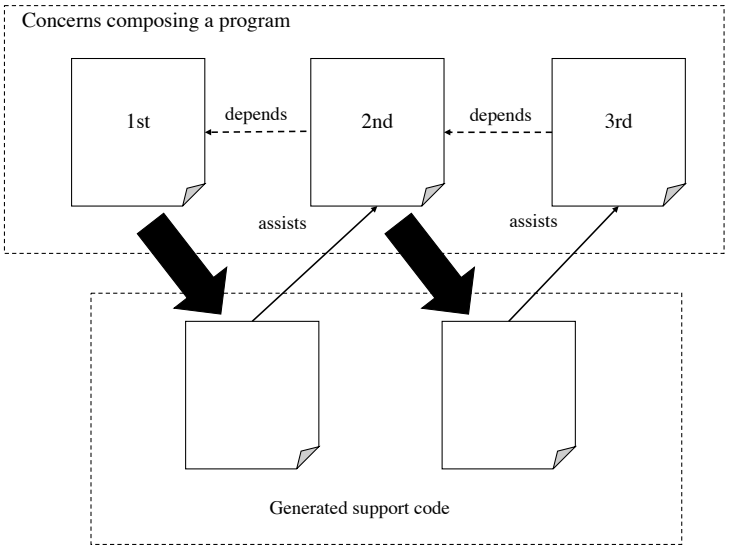

Figure 5.10: Divide-and-generate pattern

to identifiers. Which kind of identifiers that should be used to encode a name depends on where the name should be suggested with code completion. For example, if the name should be suggested as a method name in a class, the name should be declared as a method name in the superclass declared in the support code.

The availability of each name is passed to the user program by importing the declarations in the support code into the name scope. The IDEs for the host language use a name scope at the cursor position where the user is editing to provide auto-completion. Although this auto-completion is generic assistance for the host language, we can regard it as a domain-specific one. Similarly, the IDE warns the user if an unavailable name is used in the scope.

5.5.3 Applicability

The divide-and-generate pattern can be applied when all the following conditions are satisfied.

- There is an IDE for the host language, which provides code completion and error checking.
- Program is divided into two or more files such that each file addresses a separate concern, and all names needed for describing a concern are provided by another concern.
- The names are naturally encoded to an identifier in the host language.

In the case of MELTRANS, the divide-and-generate pattern can be applied for the following reason. Since the host language is Java, IDEs such as Eclipse or NetBeans are available. MELTRANS program is divided into six class files. Each class file addresses a separate concern. For example, the format concern addresses the format of each instruction. Similarly, the branch predicate concern

addresses which instructions are branch instructions. The delay slots concern addresses the number of delay slots of each branch instruction. In branch predicate concern, instruction names are used to specifies which instructions are branch instructions. These instruction names are given by the format concern, which specifies the name and the format of each instruction. Similarly, names of instruction fields used in the branch predicate concern are provided by the format concern. The branch predicate concern uses the names of instruction fields to check if some instructions are branch instructions or not. Since the name of instructions and instruction fields consist of alphanumeric characters, they are naturally encoded to an identifier in Java.

5.5.4 Consequences

Some of the benefits and liabilities of the divide-and-generate pattern are as follows:

1. Code completion is available when the user of the DSL types one of the names also declared in another concern.
2. The names suggested by code completion are limited to valid ones.
3. An error is detected immediately after the user types an invalid name that is not declared in other concerns.
4. Inconsistency derived from updating a concern is detected by error checking. If a name is removed or changed in a concern, the IDE warns against the use of the name in another concern.
5. Programs for each concern may be slightly noisy, while noisy parts may be automatically filled with code completion.

5.5.5 Implementation

We used Java's superclasses as support code in our case study on MELTRANS. In this section, we show other options for support code.

C Macros as Support Code

If we use C as a host language of an embedded DSL for processor description, we can use C macros as support code. For example, we can generate C macro from the description of the format concern instead of generating a superclass as follows.

```
#define DEF_ADD void add(int rs, int rt, int rd)
#define DEF_ADDI void addi(int rs, int rt, int immediate)
```

We can define macro variables for describing the semantics concern. When the user writes a program, domain-specific assistance can be provided as shown in Figure 5.11. The user does not need to write the signature of a function directly. The user can use a macro defined in support code instead of writing the signature of a function directly. In addition, the user can use auto-completion when writing the name of a macro as shown in Figure 5.11. When writing the body of the function, the user can benefit from domain-specific assistance similar to the ones presented in 5.3.1.

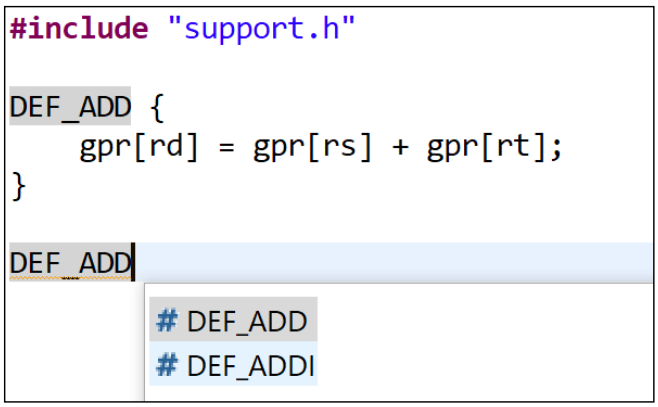

Figure 5.11: Domain-specific assistance with C macros

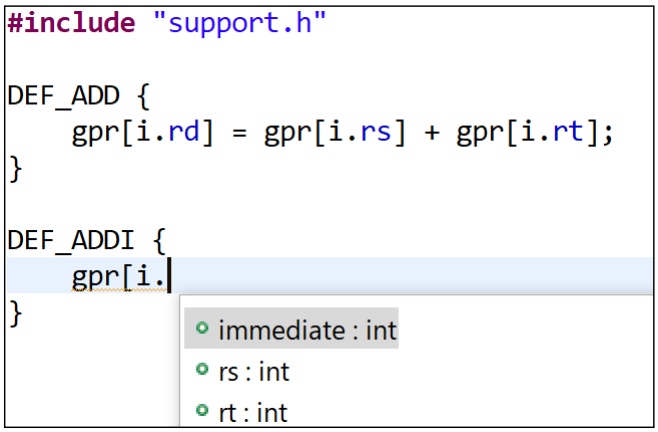

Figure 5.12: Domain-specific assistance with C macros and structs

C Macros and C Structs as Support Code

Another option for the implementation of support code is using *struct* when the host language is C, Instead of generating support code containing only macros, we can generate the following support code containing the definitions of structures.

```
struct ADD { int rs; int rt; int rd; };
struct ADDI { int rs; int rt; int immediate; };

#define DEF_ADD void add(struct ADD i)
#define DEF_ADDI void addi(struct ADDI i)
```

There is a C struct for each instruction. Each member of a C struct corresponds to each field of the instruction described in the format concern. When writing a code for the semantics concern, dynamic domain-specific assistance is available as shown in Figure 5.12. In this case, the name of each instruction field is encoded into a member of a C struct. The user can know which instruction field is available with auto-completion for the names of the members in that C struct.

5.6 Related Work

Programming Assistance in Embedded DSLs

One of the advantages of embedded DSLs is its low implementation cost. However, domain-specific programming assistance in embedded DSLs is poor. Researchers have tackled this problem, and their solutions appear to be able to be combined with our approach.

Dinkelaker [26] proposed the Eclipse plug-in called TigersEye, which enables the use of the domain's established syntax in programs in an embedded DSL. Nosal *et al.* [70] proposed techniques for customizing host IDEs for embedded DSLs, including the prevention of inexpert editing, code completion, and error reporting.

Embedded DSLs for Emulator Generation

Several embedded DSL approaches have been proposed for generating processor emulators; however, they do not provide domain-specific assistance. Pydgin [62] uses a Python-based embedded DSL for generating processor emulators. It uses PyPy's [12] meta-tracing JIT compiler for DBT. The resulting processor simulator runs the guest program with tracing JIT compilation. ArchC [8] is a SystemC [73]-based PDL. SystemC is also an embedded DSL hosted by C++ for system simulation. A processor description in ArchC can be compiled as a C++ program and runs as an interpretive processor emulator. Wagstaff *et al.* [93] proposed a method to generate processor emulators with DBT using the description written in ArchC.

Engel *et al.* [29] and Okuda et al. [72] proposed frameworks in C/C++ to generate static binary translators and dynamic binary translators, respectively. In these systems, the user can develop a processor emulator with binary translation as if an interpretive one is developed.

Using Ihneritance

The combination of separation of concerns and generating superclass have been used in parser generation, while the aim is not to provide dynamic domain-specific assistance. ANTLR [74] and SablcCC [33] are parser generator. They divide description for a parser into a grammar and semantic actions. They generate not only a parser from a grammar but also classes for traversing a parse tree generated by the parser. The user can write semantic actions in the class, inheriting the superclass generated by the parser generator. While we used superclass as support code in MELTRANS, we also introduced that other language mechanisms could be used to provide domain-specific assistance in 5.5.

5.7 Summary

This chapter presented our design approach for developing an embedded DSL with domain-specific programming assistance. The proposed approach divides DSL programming into multiple stages, and the language processor of that DSL generates a program from the program written by the user in an earlier stage.

The generated program exploits the inheritance mechanism to provide domain-specific assistance to the user. To demonstrate our approach, we explain the design of our PDL named MELTRANS, which is an embedded DSL hosted by Java. We implemented several emulators in MELTRANS and experimentally confirmed that our domain-specific programming assistance effectively reduces the amount of code that needs to be written by the user.

Chapter 6

Conclusion

In this dissertation, we proposed methods and design approaches that minimize the effort to provide domain-specific programming assistance in embedded DSLs. We categorized programming assistance into static one and dynamic one and studied them, respectively.

In chapter 3, we proposed lake symbols for island parsing. The lake symbol is a novel grammatical symbol that the DSL developer can use in the grammar for syntax extension. The user can use lake symbols to skip the uninteresting parts of the input program instead of describing the detailed rules for the uninteresting parts. When the application is syntax extension, only extended programming constructs are interesting. Therefore, the lake symbol can reduce the number of rules in a grammar to be written by the DSL developer.

Chapter 4 proposed an interactive editing method for an island grammar and tool called PEGSEED that helps the DSL developer write an island grammar with lake symbols. With PEGSEED, the language designer can write a functional island grammar in a step-by-step manner. In each step, she adds a rule for a new island. After adding a new rule, she can test the grammar on an example text by highlighting the text area recognized by the latest rule. A rule for a new island can be added by concatenating already tested islands. The DSL designer can efficiently get the expected island grammar by incrementally refining the island grammar tested in each step. PEGSEED also provides GUI operations to add a new rule by using an example text. By selecting a text area and applying one of the GUI operations, the user can add a new rule without writing it by hand. Our case study shows that the parsers for syntax extension can be developed only with the GUI operations provided by PEGSEED.

In chapter 5, we introduced the importance of domain-specific programming assistance for embedded DSLs. We define auto-completion and error checking provided by IDEs as dynamic programming assistance. Careful language design enables dynamic domain-specific programming assistance via an IDE for the host language. We demonstrated this with our practical processor description language called MELTRANS. Our case study shows that domain-specific assistance can be available by exploiting an IDE for the host programming language. Moreover, because our design approach does not need to customize the IDE or develop a specialized IDE, it does not sacrifice the benefit of embedded DSLs that the construction cost is low.

Future Work

In chapter 3, we proposed lake symbols for island parsing. Our lake symbols have been implemented only in the extended PEG. The implementation of lake symbols for another grammar class is future work. The concept of lake symbols will be applicable to not only PEGs but also to context-free grammars (CFGs) and their subsets, such as LR and LL grammars.

Another future work related to lake symbols is to evaluate with more practical applications. Our experiment confirmed that the number of rules to extract a specific kind of programming construct decreased with lakes. Although we believe that reducing the number of rules eases the construction of an island parser, the relation between the real workload and the number of rules has not been studied experimentally.

In chapter 5, we introduced the *divide-and-generate* pattern for dynamic programming assistance for embedded DSLs. We think there are other patterns that enable domain-specific assistance. Finding these patterns is future work. In the case study, we developed our processor description language MELTRANS to confirm the effectiveness of our *divide-and-generate* pattern. The results showed that domain-specific assistance could be achieved without sacrificing the performance of generated emulators. Moreover, the amount of code to be written in MELTRANS is smaller due to available auto-completion in MELTRANS. Evaluation of the *divide-and-generate* pattern on other domains is also another future work.

Appendix A

Additional Information Related to Lake Symbols

A.1 Parsing Expression Grammars

This appendix gives a brief explanation on PEGs used in this dissertation. According to the literature [32], a parsing expression grammar (PEG) is a 4-tuple $G = (V_N, V_T, R, e_s)$, where V_N is a finite set of non-terminal symbols, V_T is a finite set of terminal symbols, R is a finite set of rules, e_s is a start expression, and $V_N \cap V_T = \emptyset$. Each rule $r \in R$ is represented as $A \leftarrow e$, where A is a non-terminal symbol and e is a parsing expression.

A parsing expression is similar to a regular expression in that it presents a pattern of strings to be recognized. A parsing expression consists of a sequence of terminal/non-terminal symbols and operators. The operators are summarized in Table A.1. The semantics of operators except /, !, and & are the same as those in regular expressions. / and ! have important roles in island parsing.

/ operator is the prioritized choice operator. When both the left operand e_1 and right operand e_2 match an input string, e_1 is always prioritized. Hence, ambiguity does not exist in a parsing expression. In island parsing, we use a wildcard to skip water. If we use a wildcard as one of the operands of a ordinal choice operator, the grammar becomes ambiguous. In PEG, the user can disambiguate this situation by putting the wildcard on the right hand of / operator.

! is the lookahead not-predicate. The parser is expected to lookahead the input to check if the operand of ! recognize the head of the input without consuming any character.

A.2 Fixed-Point Constraints

This appendix shows the constraints satisfied by the fixed-points computed by Algorithms 1–3. We use meta-variables e_i, e_j, ... ranging over all the parsing expressions, including sub-expressions in a grammar. e_i, e_j, ... are location-aware. Location-aware means that two lexically equivalent parsing expressions e_i and e_j are not identical when they belong to a different rule or they are different sub-expressions in the same expression.

Table A.1: Operators for parsing expressions

Operator	Precedence	Description
.	5	Any character
(e)	5	Grouping
$e?$	4	Optional
$e*$	4	Zero-or-more
$!e$	3	Not-predicate (Negative lookahead)
$\&e$	3	And-predicate (Positive lookahead)
e_1e_2	2	Sequence
e_1/e_2	1	Prioritized choice

A.2.1 ALT

The ALT sets can be computed as the fixed-point over the following constraints:

1. If e_i is a nonterminal or lake symbol $\mathscr{S}$ and $\mathscr{S} \leftarrow e_j \subseteq R'$, then $ALT(e_i) \in ALT(e_j)$
2. If e_i is e_j*, e_j+, or $e_j?$, then $ALT(e_j) = ALT(e_i) \cup SUCCEED(e_i)$
3. If e_i is $!e_j$, then $ALT(e_j) = SUCCEED(e_i)$
4. If e_i is $\&e_j$, then $ALT(e_j) = ALT(e_i)$
5. If e_i is e_j/e_k, then $ALT(e_k) = ALT(e_i)$ and
   (a) If $\epsilon \in BEGINNING(e_k)$, then $ALT(e_j) = ALT(e_i) \cup (BEGINNING(e_k) - \{\epsilon\}) \cup SUCCEED(e_k)$
   (b) Else $ALT(e_j) = ALT(e_i) \cup BEGINNING(e_k)$
6. If e_i is e_je_k, then $ALT(e_j) = ALT(e_i)$ and
   (a) If $\epsilon \in BEGINNING(e_j)$, then $ALT(e_k) = ALT(e_i)$
   (b) Else $ALT(e_k) = \emptyset$

A.2.2 BEGINNING

The $BEGINNING$ sets can be computed as the fixed-point over the following constraints:

1. If e_i is a terminal symbol α, then $BEGINNING(e_i) = \{\alpha\}$
2. If e_i is a nonterminal or lake symbol $\mathscr{S}$ and $\mathscr{S} \leftarrow e_j \in R'$, then $BEGINNING(e_i) = \{\mathscr{S}\}$
3. If e_i is $e_j?$ or e_j*, then $BEGINNING(e_i) = BEGINNING(e_j) \cup \{\epsilon\}$
4. If e_i is e_j+, then $BEGINNING(e_i) = BEGINNING(e_j)$
5. If e_i is $!e_j$ or $\&e_j$, then $BEGINNING(e_i) = \{\epsilon\}$

6. If e_i is e_j / e_k, then $BEGINNING(e_i) = BEGINNING(e_j) \cup BEGINNING(e_k)$
7. If e_i is $e_j e_k$
   (a) If $\epsilon \in BEGINNING(e_j)$, then $BEGINNING(e_i) = (BEGINNING(e_j) - \{\epsilon\}) \cup BEGINNING(e_k)$
   (b) Else $BEGINNING(e_i) = BEGINNING(e_j)$

A.2.3 SUCCEED

The *SUCCEED* sets can be computed as the fixed-point over the following constraints:

1. If e_i is a nonterminal or lake symbol $\mathscr{S}$ and $\mathscr{S} \leftarrow e_j \in R'$, then $SUCCEED(e_i) \subseteq SUCCEED(e_j)$
2. If e_i is $e_j?$, then $SUCCEED(e_j) = SUCCEED(e_i)$
3. If e_i is e_j* or e_j+, then $SUCCEED(e_j) = SUCCEED(e_i) \cup BEGINNING(e_i) - \{\epsilon\}$
4. If e_i is $!e_j$ or $\&e_j$, then $SUCCEED(e_j) = \emptyset$
5. If e_i is e_j / e_k, then $SUCCEED(e_j) = SUCCEED(e_i)$ and $SUCCEED(e_k) = SUCCEED(e_i)$
6. If e_i is $e_j e_k$, then $SUCCEED(e_k) = SUCCEED(e_i)$
   (a) If $\epsilon \in BEGINNING(e_k)$, then $SUCCEED(e_j) = (BEGINNING(e_k) - \{\epsilon\}) \cup SUCCEED(e_k)$
   (b) Else $SUCCEED(e_j) = BEGINNING(e_k)$

References

[1] Python Software Foundation. Python Language Reference, version 2.7. `http://www.python.org`.

[2] Ali Afroozeh, Jean-Christophe Bach, Mark Van den Brand, Adrian Johnstone, Maarten Manders, Pierre-Etienne Moreau, and Elizabeth Scott. Island grammar-based parsing using gll and tom. In *Proceedings of the 5th International Conference on Software Language Engineering (SLE'12)*, pages 224–243, 09 2012.

[3] M. Di Penta and. Towards the automatic evolution of reengineering tools. In *Ninth European Conference on Software Maintenance and Reengineering*, pages 241–244, March 2005.

[4] Dana Angluin. Inference of reversible languages. *J. ACM*, 29(3):741–765, July 1982.

[5] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, November 1987.

[6] Arvind Arasu, Surajit Chaudhuri, and Raghav Kaushik. Learning string transformations from examples. *Proc. VLDB Endow.*, 2(1):514–525, August 2009.

[7] Motor Industry Software Reliability Association and Motor Industry Software Reliability Association Staff. *MISRA C:2012: Guidelines for the Use of the C Language in Critical Systems*. Motor Industry Research Association, 2013.

[8] Rodolfo Azevedo, Sandro Rigo, Marcus Bartholomeu, Guido Araujo, Cristiano Araujo, and Edna Barros. The archc architecture description language and tools. *Int. J. Parallel Program.*, 33(5):453—484, October 2005.

[9] Alberto Bacchelli, Anthony Cleve, Michele Lanza, and Andrea Mocci. Extracting structured data from natural language documents with island parsing. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 476–479, Nov 2011.

[10] Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin Zorn. Flashrelate: Extracting relational data from semi-structured spreadsheets using examples. *SIGPLAN Not.*, 50(6):218–228, June 2015.

[11] Walter R. Bischofberger. Sniff a pragmatic approach to a c++ programming environment. In *IN USENIX C++ CONFERENCE*, pages 67–82, 1992.

[12] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18—-25. ACM, 2009.

[13] Jean Bovet and Terence Parr. Antlrworks: An antlr grammar development environment. *Softw. Pract. Exper.*, 38(12):1305–1332, October 2008.

[14] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1):52–70, 2008. Special Issue on Second issue of experimental software and toolkits (EST).

[15] Martin Bravenboer and Eelco Visser. Concrete syntax for objects: Domain-specific language embedding and assimilation without restrictions. *SIGPLAN Not.*, 39(10):365–383, October 2004.

[16] Barrett R. Bryant, Jeff Gray, and Marjan Mernik. Domain-specific software engineering. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 65—-68. ACM, 2010.

[17] Shigeru Chiba, YungYu Zhuang, and Maximilian Scherr. Deeply reifying running code for constructing a domain-specific language. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '16. ACM, 2016.

[18] James R. Cordy. The txl source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006. Special Issue on The Fourth Workshop on Language Descriptions, Tools, and Applications (LDTA ' 04).

[19] James R Cordy, Ian H Carmichael, and Russel Halliday. The txl programming language-version 10. *Kingston: Queen's University at Kingston and Legasys Corporation*, 2000.

[20] James R. Cordy, Charles D. Halpern-Hamu, and Eric Promislow. Txl: A rapid prototyping system for programming language dialects. *Comput. Lang.*, 16(1):97–107, January 1991.

[21] S. Crespi-Reghizzi, M. A. Melkanoff, and L. Lichten. The use of grammatical inference for designing programming languages. *Commun. ACM*, 16(2):83–90, February 1973.

[22] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky, editors. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA, USA, 1993.

[23] Thomas R. Dean, James R. Cordy, Andrew J. Malton, and Kevin A. Schneider. Agile parsing in txl. *Automated Software Engg.*, 10(4):311–336, October 2003.

[24] Massimiliano Di Penta, Pierpaolo Lombardi, Kunal Taneja, and Luigi Troiano. Search-based inference of dialect grammars. *Soft Computing*, 12(1):51–66, Jan 2008.

[25] Massimiliano Di Penta and Kunal Taneja. Towards the automatic evolution of reengineering tools. In *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, CSMR '05, pages 241–244, Washington, DC, USA, 2005. IEEE Computer Society.

[26] Tom Dinkelaker, Michael Eichberg, and Mira Mezini. Incremental concrete syntax for embedded languages. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 1309—-1316. ACM, 2011.

[27] A. Dubey, P. Jalote, and S. K. Aggarwal. Learning context-free grammar rules from a set of program. *IET Software*, 2(3):223–240, June 2008.

[28] Patrick Dubroy and Alessandro Warth. Incremental packrat parsing. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, page 14–25, New York, NY, USA, 2017. Association for Computing Machinery.

[29] Frank Engel, Johannes Nührenberg, and Gerhard P. Fettweis. A generic tool set for application specific processor architectures. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign*, CODES '00, pages 126—-130. ACM, 2000.

[30] Moritz Eysholdt and Heiko Behrens. Xtext: Implement your language faster than the quick and dirty way. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, page 307–309, New York, NY, USA, 2010. Association for Computing Machinery.

[31] Bryan Ford. Packrat parsing:: Simple, powerful, lazy, linear time, functional pearl. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 36–47, New York, NY, USA, 2002. ACM.

[32] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 111–122, New York, NY, USA, 2004. ACM.

[33] Etienne M Gagnon and Laurie J Hendren. *SableCC, an object-oriented compiler framework*. IEEE, 1998.

[34] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, ECOOP '93, pages 406—-431. Springer-Verlag, 1993.

[35] Yossi Gil and Tomer Levy. Formal Language Recognition with the Java Type Checker. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP*

2016), volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:27, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[36] Yossi Gil and Ori Roth. Fling - A Fluent API Generator. In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:25, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[37] Geoffrey K. Gill and Chris F. Kemerer. Cyclomatic complexity density and software maintenance productivity. *IEEE transactions on software engineering*, 17(12):1284–1288, 1991.

[38] Alexey Goloveshkin and Stanislav Mikhalkovich. Tolerant parsing with a special kind of <<any>>symbol: the algorithm and practical application. 30:7–28, 10 2018.

[39] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

[40] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM.

[41] George Hadjiyiannis, Pietro Russo, and Srinivas Devadas. A methodology for accurate performance evaluation in architecture exploration. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, DAC '99, pages 927–932. ACM, 1999.

[42] William R. Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 317–328, New York, NY, USA, 2011. ACM.

[43] Mark R. Hartoog, James A. Rowson, Prakash D. Reddy, Soumya Desai, Douglas D. Dunlop, Edwin A. Harcourt, and Neeti Khullar. Generation of software tools from processor descriptions for hardware/software codesign. In *Proceedings of the 34th Annual Design Automation Conference*, DAC '97, pages 303–306. ACM, 1997.

[44] Jan Heering, Paul Robert Hendrik Hendriks, Paul Klint, and Jan Rekers. The syntax definition formalism sdf—reference manual—. *SIGPLAN Not.*, 24(11):43–75, November 1989.

[45] ARM Holdings. Arm architecture reference manual, armv7-a and armv7-r edition. *Arm Holdings*, 2014.

[46] Matthias Höschele and Andreas Zeller. Mining input grammars from dynamic taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 720–725, New York, NY, USA, 2016. ACM.

[47] Matthias Höschele and Andreas Zeller. Mining input grammars with autogram. In *Proceedings of the 39th International Conference on Software Engineering Companion*, ICSE-C '17, pages 31–34, Piscataway, NJ, USA, 2017. IEEE Press.

[48] Faizan Javed, Marjan Mernik, Alan Sprague, and Barrett Bryant. Incrementally inferring context-free grammars for domain-specific languages. pages 363–368, 01 2006.

[49] Rola Kassem, MikaëL Briday, Jean-Luc BéChennec, Guillaume Savaton, and Yvon Trinquet. Harmless, a hardware architecture description language dedicated to real-time embedded system simulation. *J. Syst. Archit.*, 58(8):318–337, September 2012.

[50] Lennart C.L. Kats and Eelco Visser. The spoofax language workbench: Rules for declarative specification of languages and ides. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, page 444–463, New York, NY, USA, 2010. Association for Computing Machinery.

[51] Marco Kaufmann, Matthias Häsing, Thomas Preußer, and Rainer Spallek. The java virtual machine in retargetable, high-performance instruction set simulation. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 21—-30. ACM, 2011.

[52] Steven Klusener and Ralf Lammel. Deriving tolerant grammars from a base-line grammar. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, pages 179–188, Sep. 2003.

[53] Donald E Knuth. Backus normal form vs. backus naur form. *Communications of the ACM*, 7(12):735–736, 1964.

[54] Rainer Koppler. A systematic approach to fuzzy parsing. *Softw. Pract. Exper.*, 27(6):637–649, June 1997.

[55] Jan Kurs, Mircea Lungu, Rathesan Iyadurai, and Oscar Nierstrasz. Bounded seas. *Computer Languages, Systems & Structures*, 44:114 – 140, 2015. Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014).

[56] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, pages 75–86. IEEE, 2004.

[57] Alon Lavie and Masaru Tomita. Glr* - an efficient noise-skipping parsing algorithm for context free grammars. In *Proceedings of the 3rd International Workshop on Parsing Technologies 1993, IWPT 1993, Tilburg, The Netherlands and Durby, Belgium, August 10-13,*, 10 1993.

[58] Vu Le and Sumit Gulwani. Flashextract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 542–553, New York, NY, USA, 2014. ACM.

[59] Lillian Lee. Learning of context-free languages: A survey of the literature. Technical Report TR-12-96, Harvard University, 1996. Available via ftp, ftp://deas-ftp.harvard.edu/techreports/tr-12-96.ps.gz.

[60] Alan Leung and Sorin Lerner. Parsimony: An ide for example-guided synthesis of lexers and parsers. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 815–825, Piscataway, NJ, USA, 2017. IEEE Press.

[61] Alan Leung, John Sarracino, and Sorin Lerner. Interactive parser synthesis by example. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 565–574, New York, NY, USA, 2015. ACM.

[62] Derek Lockhart, Berkin Ilbeyi, and Christopher Batten. Pydgin: generating fast instruction set simulators from simple architecture descriptions with meta-tracing jit compilers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 256–267, 2015.

[63] Thomas J McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976.

[64] Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani, and Adam Tauman Kalai. A machine learning framework for programming by example. In *In ICML*, 2013.

[65] Robert C. Miller and Brad A. Myers. Lightweight structured text processing. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, pages 10–10, Berkeley, CA, USA, 1999. USENIX Association.

[66] Prabhat Mishra and Nikil Dutt. *Processor Description Languages*. Morgan Kaufmann Publishers Inc., 2008.

[67] Leon Moonen. Generating robust parsers using island grammars. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, pages 13–22, Washington, DC, USA, 2001. IEEE Computer Society.

[68] Leon Moonen. Lightweight impact analysis using island grammars. In *Proceedings 10th International Workshop on Program Comprehension*, pages 219–228, June 2002.

[69] Tomoki Nakamaru, Kazuhiro Ichikawa, Tetsuro Yamazaki, and Shigeru Chiba. Silverchain: A fluent api generator. *SIGPLAN Not.*, 52(12):199–211, October 2017.

[70] Milan Nosál, Jaroslav Porubän, and Matúš Sulír. Customizing host ide for non-programming users of pure embedded dsls: A case study. *Computer Languages, Systems & Structures*, 49:101–118, 2017.

[71] Katsumi Okuda and Haruhiko Takeyama. Decision tree generation for mdecoding irregular instructions. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, DATE '16, pages 1592—-1597. EDA Consortium, 2016.

[72] Katsumi Okuda, Minoru Yoshida, Haruhiko Takeyama, and Minoru Nakamura. Automated generation of dynamic binary translators for instruction set simulation. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 214–219. IEEE, 2017.

[73] Preeti Ranjan Panda. Systemc: A modeling platform supporting multiple design abstractions. In *Proceedings of the 14th International Symposium on Systems Synthesis*, ISSS '01, pages 75—-80. ACM, 2001.

[74] Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[75] Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic, and Heinrich Meyr. Lisa machine description language for cycle-accurate models of programmable dsp architectures. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, DAC '99, pages 933–938. ACM, 1999.

[76] Moreau Pierre-Etienne, Christophe Ringeissen, and Marian Vittek. A pattern matching compiler for multiple target languages. In Görel Hedin, editor, *Compiler Construction*, pages 61–76, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[77] Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 107–126, New York, NY, USA, 2015. ACM.

[78] Jason A. Poovey, Thomas M. Conte, Markus Levy, and Shay Gal-On. A benchmark characterization of the eembc benchmark suite. *IEEE Micro*, 29(5):18—-29, September 2009.

[79] Roman Redziejowski. Mouse: From parsing expressions to a practical parser. *Proceedings of the CS&P 2009 Workshop*, 01 2009.

[80] Mehrdad Reshadi, Nikil Dutt, and Prabhat Mishra. A retargetable framework for instruction-set architecture simulation. *ACM Trans. Embed. Comput. Syst.*, 5(2):431–452, May 2006.

[81] Peter C Rigby and Martin P Robillard. Discovering essential code elements in informal documentation. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 832–841, May 2013.

[82] Yasubumi Sakakibara. Efficient learning of context-free grammars from positive structural examples. *Inf. Comput.*, 97(1):23–60, March 1992.

[83] Elizabeth Scott and Adrian Johnstone. Gll parsing. *Electron. Notes Theor. Comput. Sci.*, 253(7):177–189, September 2010.

[84] Elizabeth Scott and Adrian Johnstone. Gll parsing. *Electronic Notes in Theoretical Computer Science*, 253(7):177–189, 2010. Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009).

[85] Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 634–651, Berlin, Heidelberg, 2012. Springer-Verlag.

[86] Andrew Stevenson and James R. Cordy. A survey of grammatical inference in software engineering. *Sci. Comput. Program.*, 96(P4):444–459, December 2014.

[87] Nikita Synytskyy, James R. Cordy, and Thomas R. Dean. Robust multilingual parsing using island grammars. In *Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '03, pages 266–278. IBM Press, 2003.

[88] Masaru Tomita. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1985.

[89] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The asf+sdf meta-environment: A component-based language development environment. *Electronic Notes in Theoretical Computer Science*, 44(2):3–8, 2001. LDTA'01, First Workshop on Language Descriptions, Tools and Applications (a Satellite Event of ETAPS 2001).

[90] Arie Van Deursen and Tobias Kuipers. Building documentation generators. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, pages 40–49, Aug 1999.

[91] Matej Črepinšek, Marjan Mernik, and Viljem Žumer. Extracting grammar from programs: Brute force approach. *SIGPLAN Not.*, 40(4):29–38, April 2005.

[92] Eelco Visser et al. *Scannerless generalized-LR parsing*. Universiteit van Amsterdam. Programming Research Group, 1997.

[93] Harry Wagstaff, Miles Gould, Björn Franke, and Nigel Topham. Early partial evaluation in a jit-compiled, retargetable instruction set simulator generated from a high-level architecture description. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13. ACM, 2013.

[94] Tetsuro Yamazaki, Tomoki Nakamaru, Kazuhiro Ichikawa, and Shigeru Chiba. Generating a fluent api with syntax checking from an lr grammar. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[95] Kuat Yessenov, Shubham Tulsiani, Aditya Menon, Robert C. Miller, Sumit Gulwani, Butler Lampson, and Adam Kalai. A colorful approach to text processing by example. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13, pages 495–504, New York, NY, USA, 2013. ACM.

[96] Daniel H Younger. Recognition and parsing of context-free languages in time n3. *Information and Control*, 10(2):189 – 208, 1967.

[97] Vadim Zaytsev. Parser generation by example for legacy pattern languages. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2017, pages 212–218, New York, NY, USA, 2017. ACM.