The University of Tokyo

Graduate School of Information Science and Technology

Department of Creative Informatics

博士論文

A Platform for Composable and Statically-Checkable Domain-Specific Languages

(合成可能かつ静的検査可能なドメイン専用言語のためのプラットフォーム)

Doctoral Dissertation of:

Kazuhiro Ichikawa

市川　和央

Academic Advisor:

Shigeru Chiba

千葉　滋

Abstract

This dissertation proposes a platform of embedded domain specific languages (embedded DSLs) with high expressiveness and composability. A DSL is a programming language specialized to a specific domain. It makes a program concise and easy to maintain since a DSL program contains only code that directly expresses domain logic. An embedded DSL is a DSL implemented as a library of a general-purpose programming language (host language). It is technically not a programming language but it looks like a language. The main advantage of embedded DSLs is composability; composability means that programmers can use multiple embedded DSLs at the same time even if they are designed without considering each other. The main disadvantage of embedded DSLs is expressiveness; expressiveness represents what DSL can be implemented. In other words, it means what kinds of functionalities of a DSL compiler programmers can implement. Since an embedded DSL is just a library, programmers cannot implement all components of a compiler for embedded DSLs as they like. This dissertation proposes a system that supports powerful syntax extensions and user-defined name binding without breaking composability. The syntax extensions enable programmers to implement the parser of a DSL and the user-defined name binding enables programmers to implement the name resolver, a part of the type checker, of a DSL.

This dissertation introduces ProteaJ, which is a programming language supporting composable and powerful user-defined operators. The user-defined operators, named protean operators, are procedures with their own syntax. Protean operators can overload the operators that have the same syntax but have different return type or operand types. An operator is available only in the expression where its return type is expected. Since programmers can regard return types and operand types as non-terminal symbols, programmers can define various syntax as user-defined operators by exploiting types as non-terminals. This helps programmers to implement syntax of embedded DSLs. DSL users can safely use multiple embedded DSLs that different programmers developed even if the embedded DSLs have operators that have similar syntax. Programmers do not have to care about conflicts of syntax since protean operators are distinguished by types. Therefore, protean operators are powerful syntax extensions that do not break composability.

ProteaJ also supports context-sensitive expressions, which are a variant of lambda expressions, for expressing name binding and scope rules. Context-

sensitive expressions take parameters but the parameters are not explicitly written. Since the parameter names are not given, the parameters cannot be accessed via the parameter names. Instead, the members of such a parameter are available without receivers in context-sensitive expressions. In ProteaJ, protean operators can be instance members. This means that we can express syntax that is available only at context-sensitive expressions. Hence, we can express name binding and scope rules by using context-sensitive expressions. ProteaJ supports turnstile types, which express the types of context-sensitive expressions. ProteaJ also supports generic names to recognize arbitrary names given by end-users. User-defined language constructs implemented by using context-sensitive expressions can be safely composed since the scope of protean operators is restricted by turnstile types.

Furthermore, ProteaJ allows programmers to define declarations with their own scope rules. A declaration declares a name and enables to use it after the declaration. ProteaJ provides an `activate` statement that takes an object as its argument and it evaluates the rest of the program as a context-sensitive expression taking the given object. It enables users to use the syntax that is expressed by the protean operators that are the instance members of the given object after the `activate` statement. Therefore, the `activate` statement works as a variable declaration. Similarly, the operators using `activate` statements within their body also work as declarations. To express the range where the syntax introduced by an `activate` statement is available, programmers can use `scope for` clauses. A `scope for` clause is attached to an operand of an operator (or a parameter of a method) and it takes a type. If an `activate` statement with an object of the given type is used in the operand, the syntax introduced by the `activate` statement is available only after the statement but not available outside the operand. The `scope for` clause works as a scope of the syntax. Programmers can implement different scope rules for each embedded DSLs. To statically check which members are activated, ProteaJ also provides `activates` clauses. Since the compiler can check which members are available by checking the signatures of methods/operators, user-defined declarations and their scope rules can be safely composed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

There are huge number of programming languages today and programmers can select a language for their purpose. When they develop a program that runs on a supercomputer, they often select C, C++, Fortran, and so on. Scripting languages such as JavaScript, Ruby, Python, and PHP are often used for developing web pages. If programmers want to write more robust programs, they can select statically typed programming languages such as Java, Scala, OCaml, and Haskell.

Language oriented programming (LOP) is a paradigm of software development in which programmers develop *domain specific languages* (DSLs) at first rather than directly write a program in a general-purpose language [80]. A DSL is a programming language that is a programming language for a particular problem domain. Programmers can write more concise and readable code for the particular domain. In LOP, programmers develop most part of software as DSL programs. It improves maintainability since DSL programs only describe business logic. LOP is particularly effective for large-scale development. In large-scale development, most members might not be an expert of programming. In LOP, most programmers do not have to write a general-purpose language; their tasks are to simply write business logic in DSLs. Only a few experts write complex logic in a general-purpose language. Therefore, we need a platform for DSLs with that domain experts can easily implement powerful DSLs while non-experts easily use these multiple DSLs at the same time. It is desirable that the DSLs can interact with each other. It is also desirable that the DSLs have compile-time checking to find several bugs before run-time since most users of the DSLs are non-experts.

Unfortunately, the cost of developing a DSL from scratch is extremely high. To reduce this cost, various tools to help implementing a compiler or an interpreter have been developed. Such the tools are called as *compiler-generators* or *compiler-compilers*. We can regard them as DSLs for developing a programming language; therefore, we can develop a DSL without considering implementation details such as a parsing algorithm. However, developing a new DSL is still difficult. A compiler or interpreter of a programming language, even a small DSL,

is a large software product, so it is difficult to apply LOP for usual software development.

Embedded DSL (or internal DSL) is an idea for developing a DSL in lower cost by "inheriting the infrastructure of some other language" [32]. An embedded DSL is not a programming language. An embedded DSL is just a library of the *host language* but it looks like a DSL. It exploits language features of the host language such as macros for emulating the DSL. In the rest of this dissertation, we call normal DSLs *external DSLs*. If we simply write DSLs, they mean either or both of embedded DSLs and external DSLs.

An advantage of embedded DSLs is that a DSL can cooperate with other DSLs and its host language. Since an embedded DSL is just a library of the host language, programmers can use it only by importing the library. Unlike external DSLs, programmers can use multiple embedded DSLs at the same time as if they are parts of a single integrated DSL. This property is called *composability* [55]. In LOP, composability is important property since programmers develop a product by using multiple DSLs that are suitable for each component of the product.

A disadvantage of embedded DSLs is that *expressiveness* of DSLs is restricted by the host language. Expressiveness means what DSLs the platform can implement. Several features of DSLs are difficult to implement to embedded DSLs since an embedded DSL is just a library of the host language. Especially, *static checking* is an important feature that is difficult to implement for embedded DSLs. Static checking is a compiler task that consists of parsing and type checking. An embedded DSL can have only the syntax that can be parsed in the host language since syntax of an embedded DSL is implemented by exploiting the host language syntax. For example, *fluent interfaces* [22] are a technique for implementing embedded DSLs in object-oriented programming language. Fluent interfaces exploit a method call chain to express DSL syntax. Therefore, they cannot use keywords, such as `if`, since they cannot be used as a method name. Type checking of embedded DSLs is also a difficult issue. If an embedded DSL has different type rules from those of the host language, how does the compiler compose these rules?

This dissertation aims to give a platform of DSLs with good expressiveness and composability. We adopt an approach based on embedded DSLs for achieving composability. To achieve expressiveness, we support powerful syntax extensions and allow programmers to define name binding rules. Powerful syntax extensions allow embedded DSLs to have natural syntax. Name binding is a part of type checking that resolves the association of a definition with an identifier. These help embedded DSLs to be more natural, domain-specific, and safe. They are very useful for LOP since LOP is a paradigm of software development aiming at making business logic clear and safe.

We first show *protean operators*, which are user-defined operators that can express user-defined literals. Protean operators enable programmers to implement any DSL syntax that is expressed by parsing expression grammar (PEG) [20]. Then we show *context-sensitive expressions* and *turnstile types*, which enhance protean operators to cover DSL syntax involving name binding. They

enable programmers to implement lambda expressions, let expressions, for expressions, and so on. Then we show an extension of our system for supporting declaration-style name binding. A declaration is an expression or statement that declares a name such as a variable or function name and makes it available after the declaration. This extension is useful for implementing practical DSLs since most programming languages have language constructs involving declaration-style name binding rather than name binding like let expressions.

1.1 Motivation

Our ultimate goal is to let programmers add any kind of new language constructs to a host language. This means, programmers can use their favorite language constructs by importing libraries. Programmers can also use different syntax of their favorite language. Their programs can be easily composed since they are developed in the same language. For example, a programmer uses Lisp syntax and another programmer uses Java-like syntax in the same language.

To do this, programmers should be able to define new syntax within the host language. They can package language constructs involving new syntax as a library. A number of programming languages have a system that allows programmers to define new syntax, however, such the system has restrictions of the syntax that can be defined or it makes composability worse. For example, *SugarJ* [18] can define new syntax sugar as a library. In SugarJ, programmers can declare new syntax of arbitrary context free grammar and implement its semantics as a transformation of the abstract syntax tree (AST). Libraries of SugarJ are usually composable, however, they are not composable in case syntax is *ambiguous*. Here, being ambiguous means there are two or more abstract syntax trees representing the same code. In SugarJ, if a program is ambiguous, the compiler fails to compile and reports an error even if the correct AST is obvious from the context. If programmers can define their own syntax and programmers can compose multiple libraries, syntax tends to be ambiguous. Especially, *user-defined literals* make syntax highly ambiguous. We aim to develop a new language extension system with less syntax restrictions and more composability.

Another problem that programmers encounter when adding a new kind of language constructs is how to implement *name binding*. Name binding is one of the most important features of a programming language. It is the association of names with values, functions, classes, or types. For example, variable declarations bind a name with a variable. This name can be used within a *scope*, for example, a curly braced code block involving the declaration. A large part of language constructs introduces their new name binding. To allow programmers to define their own language constructs, a programming language should have a facility for defining new syntax having new name binding. However, there are few languages with a system that allows programmers to define new name binding. Anaphoric macros [29] are one of such systems but they are known as being unsafe. In this dissertation, we aim to develop a new language extension system that enables programmers to safely extend name binding rules.

1.2 Approach

This dissertation proposes three language features: *protean operators*, *context-sensitive expressions*, and *context activation within a user-defined scope*. Protean operators are a language extension system that can express user-defined literals. They have less syntax restrictions and more composability. Context-sensitive expressions are a language construct that helps protean operators. They enable protean operators to safely extend name binding rules. Context activation within a user-defined scope enables programmers to implement user-defined declarations. Context activation is expressed by `activate` statements and `activates` clauses and a user-defined scope is defined by `scope for` clauses. We have developed ProteaJ, which is an extension language of Java. ProteaJ implements our proposals.

Protean operators are procedures with their own syntax. They are invoked by an operator call and the call follows the syntax of the operator. The syntax of protean operators is written as a sequence of *name parts* and *operands*. A name part is a keyword for identifying the protean operator. An operand takes an expression as its argument. The characteristics of protean operators are that operators are overloaded by operators that have the same syntax but have a different return type or operand types. An operator is available only at an expression where its return type is expected. Hence, we can regard that the return type of an operator denotes where the operator is available and the operand types restrict the syntax of the operands. Programmers can use types as if they are *non-terminal symbols* of *Backus-Naur form* (BNF) [42] for expressing various syntax. In ProteaJ, syntax of literals is also extensible in the same way to normal syntax. Programmers do not have to care about conflicts of syntax since protean operators are distinguished by types.

Context-sensitive expressions are a variant of lambda expressions. They take parameters but the parameters are not explicitly written. Since the parameter names are not given, the parameters cannot be accessed via the parameter names. Instead, the members of a parameter are available without receivers in context-sensitive expressions. In ProteaJ, protean operators can be instance members. This means that we can express the syntax that is available only at context-sensitive expressions. It enables programmers to define name binding rules by defining an operator that take a context-sensitive expression as its operand. ProteaJ supports *turnstile types*, which express the types of context-sensitive expressions. ProteaJ also supports *generic names* to recognize arbitrary names given by end-users. User-defined language constructs implemented by using context-sensitive expressions can be safely composed because the scope of protean operators is expressed by turnstile types.

An `activate` statement takes an object as its argument and enables its members after the statement without receivers. The scope of activated members does not follow the scope of local variables in the host language. If the `activate` statement is used in the body of an operator/method, the visible members are also available at the call-site of the operator/method. Hence, we can implement an operator that simulates a variable declaration. The scope of

activated members is given by `scope for` clauses. A `scope for` clause takes several types and it expresses the scope of activated members of given objects. Programmers can implement different scope rules for each embedded DSL. To statically check which members are activated, ProteaJ also provides `activates` clauses. Since the compiler can check what members are available by checking signatures of methods/operators, user-defined declarations and their scope rules can be safely composed.

1.3 Structure of This Dissertation

In the rest of this dissertation, we first describe existing approaches for a platform of DSLs and discuss advantages and disadvantages of each approach. Then we propose protean operators, which are user-defined operators that allow programmers to define any PEG grammar, in Chapter 3. In Chapter 4, we propose context-sensitive expressions and turnstile types for supporting user-defined language constructs involving name binding. Chapter 5 extends the idea of Chapter 4 to support user-defined declarations by introducing `activate` statements, `activates` clauses, and `scope for` clauses. In Chapter 6, we conclude this dissertation.

Chapter 2

Domain Specific Languages

A *domain specific language* (DSL) is a programming language specialized to a specific domain. For example, SQL is a DSL for manipulating database and LaTeX is a DSL for writing documents. Advantage of DSLs is readability and maintainability. Since a DSL is specialized to the specified domain, a DSL program contains only code that directly expresses domain logic. This makes a program concise and easy to maintain. *Language oriented programming* (LOP) is a paradigm of software development in which programmers develop DSLs at first rather than directly write a program in a general-purpose language [80]. Large parts of software component are implemented in the developed DSLs. LOP improves maintainability of software since DSL programs have better maintainability than programs in a general-purpose language. In large software development, even though developing DSLs require extra development cost, advantage of LOP exceeds the cost.

If the cost of developing DSLs were smaller, LOP would be more effective. There are a lot of tools and techniques for helping to develop DSLs. In this chapter, we describe existing tools and techniques for developing DSLs and discuss advantages and disadvantages of each technique.

2.1 Parser Generators and Parser Combinators

To implement a programming language, a programmer should define an *interpreter* or *compiler*. The interpreter reads a source code of the language and executes it. The compiler translates a source code of the DSL into a code written in another language. Both of them first read a string from a text file and they try to analyze the structure of the program. This phase is called *parsing*. A *parser* is a component of an interpreter or compiler for parsing. A parser translates a string into a tree structure that is called an *abstract syntax tree (AST)*. A parser recognizes a string according to the *syntax* (*grammar*) of the language; syntax is a set of rules that express what input string is accepted or rejected as a program of the language.

To be easy to implement an interpreter or compiler, *parser generators* are developed since long time ago. A parser generator takes a program that includes syntax of a language and AST construction rules and generates a parser that recognizes the given syntax. The syntax is usually written in *Backus-Naur form* (BNF) [42] or its variant [82]. AST construction rules (semantics of the parser) are often written in a general-purpose language such as C. For example, yacc [38] is one of the most popular parser generator. In yacc, programmers use BNF for expressing syntax of a language and they use C for expressing AST construction rules. yacc generates a parser based on *LALR(1) parsing* [14].

A *parser combinator* [79, 26, 33] is a similar tool to a parser generator but it is just a library of a general-purpose language. Such the library provides functions for construction of a small primitive parser such as an identifier parser and functions for composition of several parsers. Users build a large parser by combining a number of small parsers. A parser combinator can be regarded as an embedded DSL of a parser generator. A parser combinator is getting popular in a statically typed functional language such as Haskell [52] and Scala [53]. In such the languages, a parser combinator is designed as a *monad* [78, 34]. The parsec [47] in Haskell is a famous implementation of a monadic parser combinator.

2.1.1 Parsing Algorithms

A parser generator or parser combinator cannot handle arbitrary syntax. The range of syntax that they can handle is quantified as *language class* and *grammar class*. A set of inputs that a parser can parse is called a *language*. A language class expresses a set of languages of a parser that a parser generator or parser combinator can generate. A grammar class expresses a set of syntax (grammar) that a parser generator or parser combinator can take. The language class and grammar class of a parser generator or parser combinator depend on their parsing algorithm. For example, a parser generator based on LALR(1) parsing such as yacc can handle *LALR(1) grammar* [14]. In general, a parsing algorithm with stronger grammar class or language class takes a longer time to parse.

LALR(1) parsing is a sort of LR parsing [43]. LR parsing is an algorithm that is extension of deterministic finite automaton (DFA). LR parsing consists of two operations: *shift* and *reduce*. A shift operation corresponds to a state transition of DFA. A reduce operation expresses a derivation from an expression, which is a sequence of terminals and non-terminals, to the corresponding non-terminal symbol. The operation rewinds transitions as much as the number of the reduced sequence and applies transition that expresses reading the corresponding non-terminal symbol. The language class of LR parsing is *deterministic context free language* [27, 43]. This is enough to use for well-designed languages; however, it is difficult to support syntax composition. This is because composed syntax might be non-deterministic. Hence DSL users cannot compose multiple DSLs if the system for implementing DSLs is based on LR parsing.

Generalized LR (GLR) parsing [72] is a parsing algorithm that is an extension of LR parsing. The difference from LR parsing is that GLR parsing is

non-deterministic. To efficiently parse an input, GLR parsing uses *graph structured stack* (GSS). GLR parsing can parse any *context-free grammars* (CFGs). Note that CFGs can be *ambiguous*; in the context of parsing, the parser might return multiple ASTs from a single input. It enables programmers to compose syntax. Since composed syntax might be ambiguous, programmers should do disambiguation of the resulting ASTs. The worst-case time complexity of GLR parsing is $\mathcal{O}(N^3)$ where N is the length of the input. If the syntax is "mostly unambiguous", namely mostly deterministic, GLR parsing can parse an input in mostly linear time of the length of the input. Earley parsing [15] and CYK parsing [40, 83] are also parsing algorithms that can parse any CFGs. Both of them are based on dynamic programming. CYK parsing is the most basic algorithm for parsing CFGs; it first parses each substrings of the input with length 1, and then it parses each substrings of length 2 by using the previous results. Similarly, it parses substrings of length n by using the parsing results of substrings whose length is less than n. Earley parsing is a variant of CYK parsing that is different order of parsing substrings. Both of them has $\mathcal{O}(N^3)$ time complexity. GLR parsing can be regarded as an optimized variant of Earley parsing.

Recursive descent parsing is a parsing algorithm that implements each non-terminal symbol as a recursive function. The parser first calls the function corresponding to the start symbol. A function call expresses a derivation from its corresponding non-terminal symbol to expressions that are the right-hand side of parsing rules whose left-hand side is the corresponding non-terminal. Packrat parsing [19] is a parsing algorithm based on recursive descent parsing. Packrat parsing adopts memoization for optimization. The unique point of packrat parsing is that it saves only one AST for each position in the input and each non-terminal symbol. If there are multiple ASTs for the same position and non-terminal symbol, the parser selects one by heuristics. A packrat parser recognizes parsing expression grammar (PEG) [20] if the parser adopts ordered choice rules as the heuristics. A packrat parser cannot recognize a grammar that has several ambiguities; since the parser selects an AST for each position and non-terminal, ambiguities should be resolved at each position. Surprisingly, this does not mean the language of the parser is weaker than the language of CFG, which can have ambiguities. Packrat parsing is useful for parsing a program since almost all of programming languages do not have ambiguous syntax, and if so, the parser can parse a program in linear time of the input length. LL(*) parsing [56] is also a parsing algorithm based on recursive descent parsing. We can regard it as an optimized variant of packrat parsing that works faster if the grammar is LL(k).

2.2 Language Workbenches

Language workbenches [23] are a toolkit for DSL development. They provide several DSLs for implementing DSLs. Programmers implement a DSL by writing the specification of the DSL in declarative style. The DSL specifications can

be reused. Language workbenches also generate the integrated development environment (IDE) including editors with syntax highlighting for the defined DSLs. This reduces the cost for development of tools by integrating the implementations of the DSL and the IDE.

Spoofax [41] is a language workbench that is developed as plugins for Eclipse and IntelliJ. In Spoofax, the DSL syntax is defined in *SDF* [30], which is a DSL for syntax definitions. SDF can express any syntax in CFG as well as BNF. SDF also directly supports *operator precedence* and *operator associativity*. The algorithm of the parser generator for SDF is scannerless GLR algorithm, a variant of GLR algorithm. Spoofax includes a DSL named *NaBL* [46] to specify name binding and type checking. Name binding means identifying a name and associating its declaration. This is used for resolving variable names, function names and type names. Spoofax also includes *Stratego* [77, 10], which is a term rewriting language for program transformation. Programmers can implement syntax sugar in Stratego. The dynamic semantics of DSLs are implemented in *DynSem* [75]. In DynSem, programmers declaratively write operational semantics.

Meta Programming System (MPS) [36] is a language workbench developed by JetBrains. MPS itself is an IDE of DSLs including a DSL for implementing DSLs. A unique feature of MPS is that MPS is based on *projectional editing* [24]. Projectional editing is an alternative model of programming to source code editing. Programmers write an AST instead of a source code in projectional editing. In MPS, programmers use a code completion for writing an AST. The AST is projected to the display as if it is a source code. DSL authors should implement a projector of ASTs instead of a parser. This helps developing DSLs since implementing a projector is easier than implementing a parser. The idea of MPS derives from Intentional Programming [65].

A problem of language workbenches, especially that are based on projectional editing, is the risk of vendor lock-in [23]. Language workbenches generates a processor of a DSL and an IDE of the DSL from the definition of the DSL. Such the processor and IDE are designed that run within the language workbench. For example, if we want to use a DSL implemented in MPS, we should write a program by using projectional editing of MPS. Therefore, we cannot use the DSL without MPS.

2.3 Embedded DSLs

DSLs often have the similar features to other DSLs or general-purpose programming languages. For example, a lot of DSLs have string literals, arithmetic expressions, and conditional branches. Several DSLs are often used with other DSLs together, for example, regular expressions or URLs. Reusing such the features or DSLs is important issue.

An *embedded DSL* (or internal DSL) is a DSL implemented as a library of a general-purpose programming language (host language). It is technically not a programming language but it looks like a language. For emulating the DSL syntax, it exploits language features of the host language. For example,

syntactic macros [45, 29] are common technique for embedding DSLs in Lisp. Since an embedded DSL is just a library, we can use it with the constructs of host language together. Hudak said that embedded DSL is an idea for developing a DSL in lower cost by "inheriting the infrastructure of some other language" [32]. An important property of embedded DSLs is *composability* [55]. Composability means that programmers can use multiple embedded DSLs at the same time even if they are designed without considering each other. DSL users can use different embedded DSLs that are suitable for each part of the program and all parts work together.

2.3.1 String Embedding

String embedding is an approach in which users write a DSL code as a string literal. The DSL code is executed by the interpreter implemented in the host language. The following code snippet is an example that uses the embedded DSL based on string embedding in Java:

```
stmt.executeQuery(
  "select name from users" +
  "where name = 'ichikawa' and password = 'password'");
```

This embedded DSL is an implementation of SQL in Java. Here, `stmt` is an object of `java.sql.Statement`. The method `executeQuery` takes the string literal and it executes the given string as a SQL statement. To implement such the DSL, DSL authors should implement its parser and interpreter. The implementation of embedded DSLs based on string embedding is almost same as external (stand-alone) DSLs.

An advantage of string embedding over external DSLs is that DSL users can use host language facilities for building DSL code as a string. For example, DSL users can write the program shown in Figure 2.1. In this program, parts of SQL query are given by the `input` object. The SQL query is changed by end-user's input. Several languages have the feature named string interpolation, which helps building DSL code as a string. In C#, DSL users can write the following program:

```
db.ExecuteQuery(
  $"select name from users" +
  $"where name = '{input.Name}' and password = '{input.Password}'");
```

`$"..."` is a special string literal for string interpolation. In this literal, curly braced code is executed as C# code and the result value is embedded to the string.

String embedding has well-known problems, called code injection problems. The code injection is an attack that changing DSL code by sending unexpected inputs. Attackers can insert their malicious code into the program shown in Figure 2.1 by injecting several SQL code in the password field of `input`. For example, if attackers give `"' or '1' = '1"` as the password of input, the query is expanded as follows:

```
stmt.executeQuery(
  "select name from users" +
  "where name = '" + input.getName() + "'" +
  "and password = '" + input.getPassword() + "'")
```

Figure 2.1: A code snippet using an embedded DSL based on string embedding

```
SQL.select("name")
   .from("users")
   .where("name").equal(input.getName())
   .and("password").equal(input.getPassword());
```

Figure 2.2: Embedded DSL based on fluent interfaces in Java

```
select name from users where name = '...' and password = '' or '1' = '1'
```

This query returns all the names recorded in the `user` table since `'1' = '1'` always return true. This behavior is not intended. To prevent code injection, we should escape several characters such as quotes `'` or validate whether user input is a legal DSL value. Embedded DSLs based on string embedding usually provide the safer API than using raw strings to building DSL programs. Unfortunately, such the API contains several boilerplate codes and its program tends to be verbose.

2.3.2 Fluent Interfaces

Fluent interfaces [22] are an approach for expressing DSL syntax by using method chains. Fluent interfaces are often used in object-oriented programming languages such as Java. Figure 2.2 is a code snippet that uses an embedded DSL based on fluent interfaces in Java. In fluent interfaces, method call `obj.method(param)` is used for expressing DSL syntax `obj method param`. The first identifier of a method chain, `SQL` in this example, indicates the kind of embedded DSLs. This program is equivalent to the code shown in Figure 2.1.

An advantage of fluent interfaces is that they are safer than string embedding and DSL users can receive several tool supports such as code completions. User's input cannot change the structure of a DSL program since DSL code is already structured. Furthermore, in a statically typed language, the compiler checks the types of method calls and method definitions. If users write a vulnerable code in an embedded DSL based on fluent interfaces, the compiler might report a type error. Hence fluent interfaces are much safer than string embedding. Type safety also brings composability since each embedded DSLs can be distinguished by static types.

A disadvantage of fluent interfaces is that syntax of embedded DSLs is restricted and verbose. Symbols that are available in DSL syntax are restricted since the DSL syntax consists of method names, class names, and literals. For example, the symbol `=` cannot be used as a method or class name in Java. In Figure 2.2, the keyword `equal` is used instead. This DSL code also contains several extra keyword `SQL`, periods `.`, and parentheses `()`.

Several languages provide features that reduce verbosity of fluent interfaces. `import static` in Java is a feature that allows programmers to omit class name from calls of static methods. If DSL users use `import static`, they can omit the leading keyword of fluent interfaces such as `SQL`. Scala has syntax sugar that allow programmers to write `obj method param` instead of `obj.method(param)`. Programmers can write the following code by using the syntax sugar:

```
SQL select "name" from "users"
    where "name" equal input.getName()
    and "password" equal input.getPassword()
```

Here, we do not apply the syntax sugar to `input.getName()` because of operator precedence. Unfortunately, we cannot omit the keyword `SQL` even if we use a language feature like `import static`. This is because the receiver object should be specified for applying the syntax sugar in Scala.

To omit the keyword `SQL`, we can choose another design of the embedded DSL: in this design, `select` is an object and `name` is a method name of the object `select`. In such the DSL, we can write the following code that is equivalent to the program shown in Figure 2.2:

```
select.name(from).users
  (where).name(equal).value(input.getName())
  .and("password").equal(input.getPassword())
```

In this code, `select`, `from`, `where`, and `equal` are a variable name and `name`, `users`, `value`, `and`, and `equal` are a method name. If we apply the syntax sugar in Scala to this code, this code becomes the following code:

```
select name from users
  where name equal value(input.getName())
  and "password" equal input.getPassword()
```

This code contains extra keyword `value` since the expression `input.getName()` cannot place at a method name position. `name`, `users`, and `password` are method names in this code but they are given by DSL users and DSL authors cannot know them. Several languages such as Ruby have a feature called *method missing*, which enables DSL authors to implement such the method names. Method missing invokes a special method `method_missing` of a receiver if the given method name does not found in the receiver object. Although DSL authors can implement an embedded DSL in which DSL users can write the program above by using method missing, the implementation of the embedded DSL is complicated. This API design emulates syntax of the DSL but it does not consider the semantics of the DSL. DSL authors should reconstruct the AST that corresponds to the semantics. Furthermore, DSL users might feel

the embedded DSL is not easy to use since this API design is not consistent. Column names are sometimes given as a string literal such as `"password"` but sometimes given as a method name such as `name`. Expressions sometimes have to be enclosed by `value` method but sometimes do not have to be.

2.3.3 Operator Overloading and User-Defined Operators

A user-defined operator is a function with its own syntax. User-defined operators are useful for expressing DSL syntax. A number of languages have their restricted variant of user-defined operators. One of the most famous variant of user-defined operators is operator overloading in C++. It enables programmers to overload predefined operators such as `+` and `new`. Several languages such as Swift have more powerful feature; programmers can define their own infix binary operators, prefix unary operators, and postfix unary operators. In Swift, programmers can also declare operator precedence and associativity. They help implementing various DSL syntax. We can regard that Scala's syntax sugar is also a variant of user-defined operators. In Scala, defining method `m` with one argument for the object `obj` can be regarded as defining an infix operator `m` since we can use it as `obj m arg`. Mixfix operators [13] are a powerful variant of user-defined operators that is provided in Coq, Isabelle, Agda, et al. Mixfix operators are a collective term of infix, prefix, postfix, and outfix operators. Here, infix means syntax that starts with an operand, ends with an operand, and takes operator names between each two operands. Note that an infix operators should have at least one operator name. Prefix means syntax that starts with an operator name, ends with an operand, and alternates operands and operator names. Similarly, postfix means syntax that starts with an operand, ends with an operator name, and alternates operands and operator names. Outfix means syntax that starts and ends with an operator name and alternates operands and operator names.

Developing embedded DSLs by using user-defined operators is similar strategy to fluent interfaces but the approach based on user-defined operators can use more natural syntax than method chains. Since user-defined operators are functions, DSL authors express DSLs by combinations of function calls, just like fluent interfaces. User-defined operators can also receive the benefit of static types since they are just functions. Hence embedded DSLs based on user-defined operators are as safe as fluent interfaces and they are composable. Developing embedded DSLs by user-defined operators is easier than fluent interfaces because DSL authors can define operators that expresses syntax that corresponds to the semantics of the DSL. They do not have to reconstruct the AST unlike fluent interfaces.

User-defined operators are useful for implementing embedded DSLs, however, existing programming languages only have limited set of operators that can be defined by users. For example, mixfix operators supports only operators that do not have two adjacent operands. This means that mixfix operators can express only subset of operator precedence grammar. Such the restrictions of user-defined operators are for simplifying parsing. If a language allows program-

mers to define arbitrary operators, the syntax may be ambiguous.

2.3.4 Higher-Order Abstract Syntax

Fluent interfaces and user-defined operators can express various DSLs, however, they cannot express language constructs involving *name binding*. Name binding is the association of data with identifiers. For example, look at the following code snippet:

```
for (n <- list) println(n)
```

This code snippet uses a `for` statement. The `for` statement have a name binding, a variable `n`. The bound variable `n` is used at the body of `for` statement. Programmers cannot implement such the `for` statements by fluent interfaces or user-defined operators since they express DSLs by combinations of methods or operators, respectively.

Higher-order abstract syntax (HOAS) [58] is a technique for expressing name binding in embedded DSLs. It uses lambda expressions for expressing name binding. The `for` expression shown above is expressed in HOAS style as follows:

```
my_for(list, n -> println(n))
```

The second argument of `my_for` is a lambda expression. This lambda expression binds a variable `n` and the variable is available at the body of the lambda expression. The function `my_for` is known as `map` in functional programming languages. An advantage of the HOAS style is that the compiler can statically check the name binding. DSL authors can use HOAS style name binding within fluent interfaces or user-defined operators.

HOAS can express name binding but programmers cannot extend the syntax of name binding even if they use user-defined operators together. HOAS uses lambda expressions as a primitive of name binding. This is because a lambda expression has functionalities of name binding; it can introduce a new name and it has the scope of the new name. Since function calls, method calls, and operator calls do not have such the functionalities, fluent interfaces and user-defined operators cannot extend the syntax of lambda expressions.

Several languages support syntax overloading [7], which allows programmers to overload pre-defined language constructs such as `if` and `for`. In Scala, `for` expressions are syntax sugar of method calls `foreach`, `flatMap`, `map`, `filter`, and `withFilter` [53]. Programmers can overload `for` expressions by defining the methods `foreach`, `flatMap`, `map`, `filter`, and `withFilter`. Scala-virtualized [60] is an extension of Scala and supports overloadable language constructs such as `if`, `while`, and `new`. Scala-virtualized adopts the same approach to Scala. To overload `if` expressions, programmers define `__ifThenElse` method. Scala and scala-virtualized use HOAS to express name binding. Haskell provides syntax sugar, called `do`-notation [52], for embedded DSLs implemented with monads [78, 62]. Monads have the bind operation `>>=`, which expresses name binding of the DSLs, and the lift operation `return`, which translates the host language value to the DSL value. The bind operation `>>=` takes a lambda expression

for expressing name binding. If we use the `do`-notation, we can write the bind operation like a normal variable declaration instead of using lambda expressions. In other words, programmers can extend the pre-defined construct `do`-notation by implementing their own monads. A limitation of syntax overloading is that programmers cannot define their own language constructs with custom syntax. They can only overload pre-defined constructs.

Recaf [5] supports a limited form of user-defined language constructs with custom syntax based on HOAS. In Recaf, programmers can define a new language construct with one of the supported forms of syntax such as syntax similar to `for` or `while` statement. Recaf supports translates the new language construct into its HOAS representation. To implement the new language construct, programmers should define a function that will be used in the HOAS representation. Unfortunately, Recaf cannot translate every possible syntax into a HOAS representation: Recaf compiler can translate syntax that has the same syntax as pre-defined constructs except for keywords.

2.3.5 Lexical Macros

In C, C++, and TeX, programmers can define lexical macros, which are rewriting rules of sequences of tokens. Lexical macros take several sequences of tokens and build another sequence of tokens. C or C++ have two kinds of lexical macros; object-like macros and function-like macros [25]. An object-like macro is just an identifier and it is replaced by the specified tokens. A function-like macro is a lexical macro that looks like a function call. It takes sequences of tokens as its arguments. To define macros, programmers use `#define` directive.

Lexical macros can implement DSL syntax that does not follow the syntax of the host language. The rewriting by lexical macros is applied before the parsing. If lexical macros translate the DSL code into the host language code, the DSL code can be compiled even if it does not follow the host language syntax. The following code snippet defines function-like macros that emulate SQL syntax:

```
#define SELECT(c) select_from_where(c,
#define FROM(t)   t,
#define WHERE(w)  w)
```

We can write the following code by using these macros:

```
SELECT("name") FROM("users")
  WHERE("name = " + in.getName() + " and password = " + in.getPassword())
```

Here, `SELECT`, `FROM`, and `WHERE` are function-like macros This code is translated into the following code:

```
select_from_where("name", "users",
  "name = " + in.getName() + " and password = " + in.getPassword())
```

Although lexical macros can extend the syntax of the host language, they should not be used for implementing DSL syntax since DSL users cannot use the DSL without knowledge of its implementation. For example, the following code looks okay but it cannot be compiled:

```
SELECT("name") FROM("users")
```

The reported error message will be “expected expression”. This is because the DSL author did not implement the `SELECT` statement without `WHERE` clause. If DSL users do not know the implementation, this error message is not helpful.

Furthermore, lexical macros have a composability problem since they are distinguished only by their macro names. If DSL users use the SQL DSL with another DSL together in the same file and the DSL defines `FROM` macro, a program using the SQL DSL might not properly work. Since the macro processor cannot distinguish the `FROM` macros of two DSLs, it might select the `FROM` macro that is not for SQL DSL even if DSL users intend SQL code.

Several languages such as Common Lisp support reader macros, which enable programmers to replace the reader (also called scanner or lexer) of the language [29]. Programmers can implement arbitrary syntax of DSLs by developing a reader for it. In Common Lisp, use of a reader macro is specified by a macro character. Programmers can define their own macro character and its corresponding function that expresses a reader. Although reader macros are more powerful than lexical macros in C or C++, they also have a composability problem since they are also distinguished only by their macro characters.

2.3.6 Syntactic Macros

Syntactic macros are a language feature that rewrites branches of an AST. Lisp have supported syntactic macros for a long time. Syntactic macros find ASTs that match the given pattern and replaces them into other ASTs that are generated by executing the macro body. Scheme, a dialect of Lisp, provides `define-syntax` and `syntax-rules` for defining syntactic macros [67]. Figure 2.3 is an example defining syntactic macros in Scheme. This code defines two macros: `select` and `sql-cond`. The definitions of the macros use `syntax-rules`, which declares patterns of ASTs and rules of AST transformations. The first argument of `syntax-rules` declares a list of identifiers that are used as a keyword in the macro. The second argument declares patterns of ASTs and their corresponding rules of AST transformations. For example, the `select` macro matches the pattern `select c from t where cond ...` where `select`, `from`, and `where` are keywords. `c`, `t`, and `cond ...` denotes parameters of the macro. `cond ...` takes variable length arguments. According to line 4 in the figure, an expression using the `select` macro is expanded into the call of the function `select_from_where`. `c`, `t`, and `cond ...` are replaced into the corresponding arguments when the macro is expanded. Since the result of the macro expansion includes the macro `sql-cond`, the macro `sql-cond` is recursively expanded. If we use the macros declared in Figure 2.3, we can write the following program:

```
(define (query n p)
  (select name from users where name = n and password = p))
```

This code is expanded as follows:

```
(define-syntax select
  (syntax-rules (from where)
    ((select c from t where cond ...)
     (select-from-where 'c 't (sql-cond cond ...)))))

(define-syntax sql-cond
  (syntax-rules (and =)
    ((sql-cond c1 = v1 and rest ...)
     (sql-and (sql-eq 'c1 v1) (sql-cond rest ...)))
    ((sql-cond c1 = v1)
     (sql-eq 'c1 v1))))
```

Figure 2.3: A definition of syntactic macros in Scheme

```
(define (query n p)
  (select-from-where 'name 'users
    (sql-and (sql-eq 'name n) (sql-eq 'password p))))
```

There are a lot of researches to provide the power of Lisp macros in non-Lisp language. Syntactic macros are powerful in Lisp because Lisp has a very simple syntax, S-expressions. Bachrach and Playford demonstrated that non-Lisp language can provide Lisp-like syntactic macros [3]. They designed macro system for the object-oriented programming language Dylan. Several modern languages such as Rust and Haskell also provide syntactic macros [70, 63].

Hygiene [45] is an important property of syntactic macros. Hygiene means that local variables in a macro body do not accidentally capture variables of caller scope. Scheme, Dylan, and Rust macros are designed hygienic. In hygienic macro definitions, local variables have a lexical scope. This makes macros safer, however, this makes it difficult to define language constructs involving name binding.

Anaphoric macros [29] are also syntactic macros but they are not hygienic. They intentionally capture several variables. They are used for expressing name binding. Several languages such as Common Lisp can define anaphoric macros. The following is a sample code that defines anaphoric macros:

```
(defmacro aif (test-form then-form &optional else-form)
 `(let ((it ,test-form))
    (if it ,then-form ,else-form)))
```

This code is borrowed from [29]. It defines the macro `aif`, which can be used like `if` expression. The macro declares a variable named `it`, and the variable `it` is available at the arguments of the second parameter `then-form` and the third parameter `else-form`. Quasi-quote `` ` `` and unquote `,` are a notation for building an AST. Quasi-quote `` ` `` takes an expression and returns the AST of the expression. Unquote `,` takes an AST and returns the expression that the AST expresses. We can write the following code by using the macro `aif`:

```
(aif (read-line nil nil) (format t "read: ~A~%" it))
```

This code is expanded as follows:

```
(let ((it (read-line nil nil)))
  (if it (format t "read: ~A~%" it) nil))
```

2.3.7 User-Defined Syntax Sugar

SugarJ [18] is a programming language supporting user-defined syntax sugar. In SugarJ, programmers can declare their own syntax and transformation rules. The declared syntax and transformation rules are modularized as a library, named sugar library. Users can use such the syntax by importing the library. The parser of SugarJ is based on the parsing technique of SDF [8, 30] and the transformation of ASTs is processed in Stratego [77, 10].

Users can compose and use several sugar libraries by importing the libraries at the same time, however, library composition is not always possible. Library composition might cause syntactic ambiguities even if each library is well-designed. When the sugar libraries cause syntactic ambiguities, SugarJ compiler cannot correctly parse a program that uses the syntax sugars declared in the sugar libraries.

Wyvern [55, 54] supports type-specific languages (TSLs) and typed syntax macros (TSMs). These are based on similar idea; the syntax is changed into user-defined syntax in the special code block and the syntax is determined by the type of the code block. Wyvern provides "generic literals" for specifying the special code block. Programmers can use any Adams grammar [1] for their DSL syntax in Wyvern. Adams grammar is an extension of CFG that supports off-side rules.

2.3.8 Hybrid Approaches Between Language Workbenches and Embedded DSLs

Cedalion [50] is a programming language supporting a powerful syntax extension system but it is designed to edit by projectional editing. Syntax extensions are composable since the compiler do not need to care about ambiguities. In Cedalion, programmers directly write ASTs instead of text. This makes the compiler to be free from parsing. Since the ambiguity problem is a problem of parsing, Cedalion do not need to care it.

Cedalion has advantages of both language workbenches and embedded DSLs, however, it also has disadvantage of both language workbenches and embedded DSLs. Since Cedalion is a language workbench based on projectional editing, it has the risk of vendor lock-in. In contrast, since Cedalion is a host language of embedded DSLs, the name rules of embedded DSLs follow the name rules of Cedalion. According to the paper [50], programmers can define their own type system for their embedded DSLs in Cedalion. Unfortunately, we could not find the description about the implementation of the type system and the

limitation of the approach. In our opinion, user-defined type system for an embedded DSL might break the composability because such the type system does not know other embedded DSLs.

2.4 Comparison of Approaches for Developing DSLs

To support LOP software development, there are two issues for a platform for LOP: *expressiveness* and *composability*. Expressiveness means the power of the platform, that is, what DSLs the platform can implement. If expressiveness is high, programmers can implement a greater variety of DSLs. Users can use suitable DSL to a problem domain. Since expressiveness of platforms for embedded DSLs is inferior to the platform for external DSLs, several features of DSLs are difficult to implement. Especially, *static checking* is an important feature that is difficult to implement for embedded DSLs. Composability is a property expressing that DSL users can use multiple DSLs at the same time as if they are parts of a single integrated DSL. Programmers use a lot of DSLs for developing software in LOP. Composability helps programmers to use a number of DSLs together and make them interoperable.

Expressiveness means what DSLs the platform can implement. It also means what kinds of components of a compiler can programmers implement. Approaches based on external DSLs can implement any DSLs since they implement all components of a compiler for a DSL. In contrast, since an embedded DSL is just a library, programmers cannot implement all components of a compiler for embedded DSLs as their likes. This means that the host language restricts the expressiveness of embedded DSLs. This dissertation aims at relaxing this restriction by emulating a compiler for DSLs within the host language compiler. The task of a compiler is roughly separated into two: static-checking and optimizing. This dissertation focuses on static-checking.

Static checking is a compiler task that consists of parsing and type checking. Several languages can customize the parsing phase of the host language. We call such the customization a syntax extension. A syntax extension allows programmers to define the syntax of embedded DSLs that is available in the host language. A drawback of a syntax extension is that it might cause an *ambiguity problem*. If programmers use a powerful syntax extension, the parser might output multiple ASTs for the same program. The compiler should determine which one is correct but it is difficult task; the number of ASTs might increase exponentially. A type checker checks a program and find several bugs at compile time. If a host language allows programmers to define type checkers as their like, the compiler should compose different type checkers when the corresponding DSLs are used at the same time. This is because the compiler should determine how to check a program that uses several DSLs. This might look simple but it is difficult. Since a type checker does not know other DSLs, it cannot correctly determine the type of an expression in another DSL. To

check a program including such the expression, the type checker should be just an extension of the type checker of the host language because the type of the expression should be known for the type checker. However, can DSL developers extend the type checkers of the host language without contradiction? Actually, several languages also can implement DSL-specific error checkers but such the system has no support to compose several DSLs. In this dissertation, we choose a partial solution for it; we do not support a whole of type checking but only support *name binding*. Name binding is a part of type checking that resolves the association of a definition with an identifier.

Composability is a property expressing that DSL users can use multiple DSLs at the same time as if they are parts of a single integrated DSL. DSL users can use different embedded DSLs that are suitable for each part of the program and all parts work together. External DSLs cannot guarantee composability because they have their own syntax, name rules, type rules, and run-time semantics. It is unclear whether such the rules can be correctly composed or not. Embedded DSLs are usually composable, however, the level of the composability depends on approaches for embedding DSLs. Ideally, we want to use DSLs in the same code without "glue code" [50]. However, several approaches need glue code such as `ExecuteQuery` in the example of string embedding. We classify the level of the composability into three as follows:

- *bounded-composability*
  the DSL boundary should be explicitly specified
- *heuristic-composability*
  the compiler determines which DSL is used without considering user's intention
- *contextual-composability*
  the compiler determines which DSL is used depending on the "context"

Bounded-composability means that programmers can use multiple DSLs in the same source code but they must explicitly specify the boundary between DSLs. For example, Converge [73] is a programming language having a syntax extension system that is similar to reader macros. In Converge, programmers can use DSL syntax in the special code block called DSL block but they should specify the name of the DSL at the top of the DSL block. This is because the compiler has to know how to compile the code block. Wyvern [55, 54] is better than Converge. In Wyvern, programmers do not have to write the DSL name at the top of the block. The compiler of Wyvern can infer which DSL is used by type information. However, programmers still have to specify the boundary between DSLs. They must use special syntax called generic literals when using DSL syntax in Wyvern. If they want to use multiple DSLs, they must write a glue code that introduces a generic literal for each boundary between DSLs. We consider that such the glue code is verbose and they prevent the users from using several DSLs together.

Heuristic-composability means that programmers can use multiple DSLs without glue code and the compiler automatically infers which DSL is used

based on heuristics, for example, precedence rules. Here, what we call "heuristics" is a rule-base approach that has several deterministic and context-free rules. Such approach often infers a wrong DSL if the DSL and other DSLs cause syntactic ambiguities. Since powerful language extensions tend to cause syntactic ambiguities, we should consider this problem to design and implement a platform of DSLs. SugarJ [18] is a language having a powerful syntax extension system. The compiler of SugarJ reports an error when it cannot uniquely determine the parsing results. We consider that this behavior is not desirable for a platform of DSLs since DSLs that can be used at the same time are limited.

Contextual-composability means that programmers can use multiple DSLs without glue code and the compiler automatically infers which DSL is used depending on the context. The difference from heuristic-composability is that the compiler takes surrounding code into account for the inference of a DSL. For example, mixfix operators [13] are a powerful syntax extension system based on operator overloading. They sometimes cause syntactic ambiguities but the type checker resolves the ambiguities; the compiler adopts the DSL that passes type checking. We consider that the contextual-composability is desirable but mixfix operators are still weak to express various DSLs. Unfortunately, conventional approaches for improving expressiveness often break the composability. This dissertation aims to develop a platform of DSLs that have both high expressiveness and contextual-composability.

Table 2.1 shows the comparison of the approaches for developing DSLs. A green cell in this table expresses that the property is good for LOP. A yellowish cell in this table expresses that the property is acceptable. According to the table, every existing approach does not satisfy both properties; expressiveness and composability. Spoofax and MPS, which are language workbenches, can define flexible syntax and type checking for DSLs but they are not composable. The difference between them is that Spoofax is based on text editing and MPS is based on projectional editing. In our opinion, text editing is still better than projectional editing since most programmers are familiar to text editing and most software resources are written as texts. Several approaches for embedded DSLs such as fluent interfaces and mixfix operators have contextual-composability but they cannot define their own parsers or type checkers for DSLs. SugarJ can define parsers and type checkers for DSLs but it loses composability. Cedalion allows programmers to define any syntax for DSLs with contextual-composability. According to the paper [50], it also allows programmers to define their own type checkers for DSLs. However, it is not clear how to implement a new type system and what type systems can or cannot be implemented. A user-defined type system for an embedded DSL might break the composability since such the type system does not know other embedded DSLs.

In this dissertation, we first propose an approach for developing DSLs based on user-defined operators. It has a powerful syntax extension system (the property 'parsing' is good) and it keeps contextual-composability (the property 'composability' is good). Although these features have been already achieved by Cedalion, our approach achieves them without projectional editing; Projectional editing is good technology but programmers hesitate to use it since it

	expressiveness (static-checking)			
	parsing	type checking	composability	editor
Spoofax	CFG	yes	no	text
MPS	any grammar	yes	no	projectional
string embedding	dynamic checking	no	bounded	text
fluent interfaces	host syntax	no	contextual	text
mixfix operators	mixfix syntax	no	contextual	text
Scala-virtualized	host syntax	no	contextual	text
Recaf	host syntax	partial *1	contextual	text
C / C++ macros	function call syntax	yes	heuristic	text
reader macros	CFG	yes	bounded	text
syntactic macros	host syntax	yes	heuristic	text
SugarJ	CFG	yes	heuristic	text
Wyvern	Adams grammar	no	bounded	text
Cedalion	any grammar	partial? *2	contextual? *2	projectional

*1 only name binding for special syntax

*2 According to the paper [50], programmers can define their own type system for their embedded DSLs in Cedalion, however, we could not find the description how to implement a new type system and the limitation of the approach. If it has no limitation, it might break the composability.

Table 2.1: Comparison of approaches for developing DSLs

is incompatible with existing tool-chains such as clipboard, shells, and version managers. Then, we extend our approach to support name binding. This enables programmers to extend a part of type checking rules (the property 'type checking' is acceptable). It does not weaken the power of syntax extension (the property 'parsing' is good) and it also keeps contextual-composability (the property 'composability' is good). Our approach can be applied to other languages that are statically typed and do not support type inference. The contribution of this dissertation is giving a platform of DSLs with good expressiveness and composability and giving an approach for implementing such a platform. Although our approach basically assumes that the program is written in a text editor, it seems that part of the approach can be applied to projectional editing as well. We consider that we can also implement a language workbench that has high expressiveness and composability by applying part of our approach.

In the following chapter, we first show that we can construct a language supporting a powerful syntax extension system with contextual-composability. Then, we show that we can extend the language to support local name binding, which is a part of type checking, and demonstrate that we can implement several language constructs such as `let` expressions and lambda expressions in our language. Then, we extend our language to support declarations, which are another kind of name binding. A declaration declares something and enables to use it after the declaration. In conclusion, this dissertation proposes a platform of embedded DSLs that supports powerful syntax extensions (parsing) and name

resolution (part of type checking) with contextual-composability.

Chapter 3

Protean Operators

3.1 Introduction

A Domain Specific Language (DSL) is a simple programming language specially designed for only a limited purpose. Since a DSL is specialized for its application domain, its source code is more concise and intuitive than the equivalent code written in a general purpose language. An internal DSL [23] (or Embedded DSL) is a DSL that is implemented as a library in a general purpose language. It can be used together with the general purpose language (called the host language) since a program written in the DSL is still a valid program in the host language. It can be also used together with another DSL implemented on the same host language since both DSL programs are host language programs. An advantage of internal DSLs is this feature, *composability*. On the other hand, internal DSLs have drawbacks in the syntax – the syntax of internal DSLs is restricted by their host language. We aims to relax the restriction of the DSL syntax.

Composable user-defined operators are a useful tool for implementing internal DSLs since we can consider that they define their own syntax and semantics. The overloaded operators in C++ are simple user-defined operators but there have been user-defined operators that enable syntax extension. Mixfix operators [13] are one of the most powerful implementation of composable user-defined operators. However, the expressiveness of the mixfix operators is still limited and they cannot express certain kinds of syntax for internal DSLs. A typical problem is that they cannot express *user-defined literals*. A literal is a token that corresponds to a particular value. Changing the syntax of a token is difficult at a parse level since a token is normally recognized by a lexical analyzer and it is passed to a parser. A number of existing DSLs have their own literals that are unique in their domain. For example, flex [48], which is a DSL for generating a scanner, has literals for expressing regular expressions. Without such user-defined literals, they would have to be expressed by character strings; it weakens maintainability and safety since the compiler does not check that the string character fits the literal syntax. Using user-defined literals makes the

program concise and safe but it makes the parsing difficult. User-defined literals might introduce a large number of ambiguities into the grammar.

In this chapter, we propose new composable user-defined operators, named *protean operators*. They can express user-defined literals such as regular expressions and they are designed to be parsed in pragmatic time. There are two important features for efficient parsing: operator overloading and a precedence rule of operators. The first one is that a protean operator is overloaded on its return type and its parameter types. It enables us to consider static types as non-terminal symbols in the grammar. The compiler can use static type information for parsing. It resolves ambiguities of the rules with the same syntax but a different type. Furthermore, it also guarantees their composability. The second feature is that protean operators with the same return type require that the precedence among them is explicitly specified. These precedence rules completely remove ambiguities from the grammar since all the rules applicable to the same place are ordered. The parser can efficiently parse expressions including protean operators since the grammar has no ambiguities. We have developed *ProteaJ*, which is a subset language of Java supporting protean operators. For supporting the type system of Java, we extend our parsing method to handle subtype relationship. We have conducted an experiment for demonstrating that ProteaJ can efficiently parse expressions including user-defined literals even though a naive parsing method cannot parse them in pragmatic time. We have also conducted experiments for measuring the parsing speed of programs written in not extremely ambiguous syntax.

In the rest of this chapter, we first show the limitation of existing composable user-defined operators. Then we propose new composable user-defined operators, named *protean operators*, and we show the parsing method for them in Section 3. In Section 4, we describe how to apply our proposal into Java and we present a programming language supporting protean operators, named *ProteaJ*. Section 5 mentions experiments for evaluating the efficiency of our parsing method. Section 6 is on related work. We conclude in Section 7.

3.2 Motivation

Composable user-defined operators are useful for implementing internal DSLs. "Composable" means that operators with similar syntax can be safely used together at the same time. For example, composable operators are distinguished by static types. Figure 3.1 shows an example of composable operators. In this figure, there are three `+` operators. The first `+` operator expresses addition of integer values. The second `+` operator expresses concatenation of a string and an integer. The third `+` operator expresses concatenation of two strings. Although these three operators share the same syntax, they are distinguished by their parameter types. In some languages, programmers can define new operators that are not only predefined operators such as `+`. For example, programmers can define new binary infix operators in Scala [53]. Figure 3.2 shows a unit test program with ScalaTest [61] library in Scala. Line 3 in the figure consists of

```
for (int i = 0;  i < 10; i = i + 1) {
  print("Loop " + i + "\n");
}
```

Figure 3.1: An example of composable operators

```
val v1 = calc1()
val v2 = calc2()
v1 should be (0)
v2 should not be (0)
```

Figure 3.2: Composable user-defined operators with new syntax

a binary infix operator `should` and a function call `be(0)`. Line 4 in the figure also includes two binary infix operators `should` and `be`. We can write a unit test program by using these operators as if it is written in a domain-specific or "natural" language. We can consider that composable user-defined operators make a new language on the host language since they have their own syntax and they are separated from the host language syntax, for example, by static types. Programmers can compose a library of composable user-defined operators as an internal DSL.

Mixfix operators [13] are a powerful implementation of composable user-defined operators. Mixfix indicates prefix, postfix, infix, or outfix. An important feature of mixfix operators is that any two operands or any two name parts are not adjacent to each other. For example, the following syntax can be expressed by mixfix operators:

```
if _ then _ else _   // prefix
_ [ _ ]              // postfix
_ < _ < _            // infix
| _ |                // outfix
```

but the following syntax can not be expressed by them:

```
if _ _ else _        // two operands are adjacent to each other
_ _                  // nameless operator
_                    // implicit type coercion
```

here, an underscore `_` indicates an operand. Mixfix operators adopted in several languages such as Isabelle [57], Agda [2], and Pure [28]. Some languages supporting mixfix operators also support nameless operators. In these languages, syntax including operands that are adjacent to each other like `if _ _ else _` can be expressed.

Mixfix operators (with nameless operators) can express various syntax but they do not have sufficient syntactic expressiveness for implementing a certain kind of DSL. They cannot express complicated literals since they do not support

literal-level syntax extension. The following code is an example of a regular expression literal:

```
Regex r = hel+o ;
```

the right-hand side of `=` is a regular expression literal that denotes `helo`, `hello`, `helllo`, and so on. Expressing user-defined literals by mixfix operators is difficult since the definition of tokens read by the scanner cannot be changed. For example, the literal `hel+o` should be tokenized into `[h, e, l, +, o]`, but it is tokenized into three tokens `[hel, +, o]` in typical general purpose languages such as C and Java.

Scannerless parsing is an implementation technique for a parser that handles every character as a token. It enables us to handle literals as non-terminal symbols constructed by tokens. Since each character is a token, the syntax rules of user-defined literals can be handled by a parser. For instance, the literal `hel+o` is tokenized into `[h, e, l, +, o]` in a language implemented by a scannerless parser, and we can express the literal `hel+o` by six operators: four operand-less operators (`h`, `e`, `l`, and `o`) for recognizing a single character as a sub-expression, a nameless operator (`_ _`) for concatenating sub-expressions, and a postfix operator (`_ +`). Mixfix operators can express user-defined literals when the host language is implemented by using a scannerless parser and they support nameless operators.

A typical parser for user-defined operators generates all possible parse trees when parsing an expression. Then the compiler selects the most suitable parse tree from all possible trees by using the language semantics since the syntax of a user-defined operator should be allowed to conflict with another operator or the host language syntax. This is for flexible DSL definitions. The type checker is usually used for selecting the suitable parse tree since the type information holds the semantics of programs – what the programmer intends. For example, the expression `hel+o` has some possible parse trees and the interpretation of the expression should be changed by the context. It should be interpreted as an addition of integers if it is used as follows:

```
int i = hel+o;
```

but it should be interpreted as a regular expression literal if it is used as follows:

```
Regex r = hel+o;
```

Therefore, the syntax including regular expression literals should be an ambiguous grammar such as in Figure 3.3 and the ambiguities must be resolved by the type checker.

A typical scannerless parser is inefficient when parsing a program including user-defined literals. It must generate all possible parse trees but the number of these trees tends to be extremely large due to the ambiguity introduced by user-defined literals. Scannerless Generalized LR (SGLR) [76, 8] parser is a well-known implementation of a scannerless parser. The parsing time of SGLR parsers is proportional to the degree of ambiguities in the grammar, and the worst-case time complexity is $O(n^3)$ (n is the input length). Note that n in this

```
Stmt  → Type Id "=" Expr ";"
Expr  → Regex | Sum
Regex → Star+
Star  → Letter "+" | Letter
Sum   → Sum "+" Id | Id
Id    → Letter+
```

Figure 3.3: An example of grammar including regular expression literals

complexity is the number of tokens and it is equal to the number of characters in the program when an SGLR parser is used. n is sometimes larger than 10000. For example, the definition of the `ArrayList` class in OpenJDK 7 includes more than 12000 characters excluding comments and white-spaces.

3.3 Proposal: Protean Operators

We propose new composable user-defined operators, named *protean operators*. They can express user-defined literals such as regular expressions and parse them in pragmatic time. There are two important features of protean operators for efficient parsing : (1) overloading based on return type, and (2) parsing precedence. The overloading by return type enables the parser to resolve grammar ambiguities by using type information at parse time. The parsing precedence resolves the remaining ambiguities after the type checking by (1). Since these features resolve all the grammar ambiguities at parse time, protean operators that express user-defined literals can be parsed even in pragmatic time.

3.3.1 Protean Operators

Protean operators are composable user-defined operators that can have any number of operator names and operands. Unlike mixfix operators, a protean operator is not only infix, prefix, postfix, or outfix; for example, a "nameless" operator, which is an operator without an operator name, is a protean operator. Nameless operators are useful for implementing a concise internal DSL since they are invisible. Figure 3.4 shows examples of protean operators that express regular expression literals. To express a protean operator, we introduce the following notation: `[S]:T` represents that an operator has syntax `S` and a return type `T`. A double-quoted string denotes an operator name and `_:T` denotes an operand of type `T`. Note that Figure 3.4 is not a complete definition for the sake of simplicity. The complete definition is shown later in section 3.3.4. The literal `hel+o` is parsed as a regular expression literal as shown in Figure 3.5. The literal `hel+o` consists of four literals `h`, `e`, `l+`, and `o` and they are connected with a nameless operator. The nameless operator takes literals as two operands and it returns a new literal expressing a regular expression constructed by the concatenation of the given regular expressions.

```
(A) [ _:Regex _:Regex ]:Regex
(B) [ _:Regex "++" ]:Regex
(C) [ _:Regex "+" ]:Regex
(D) [ _:Letter ]:Regex
(E-a) [ "a" ]:Letter
(E-b) [ "b" ]:Letter
  ...
(E-z) [ "z" ]:Letter
```

Figure 3.4: The protean operators expressing regular expression literals (incomplete)

The details of the parsing of `hel+o` are the following. We assume that any single character is recognized as a token. First, each alphabetic token is interpreted as a simple `Letter` literal by the corresponding operator taking the token as an operator name such as `[ "h" ]:Letter` and `[ "e" ]:Letter`. Each of these operators can be considered as a simple user-defined literal, which consists of one letter. Each `Letter` literal is converted into a `Regex` literal by the nameless operator `[ _:Letter ]:Regex` at (D) in Figure 3.4. This nameless operator takes a `Letter` object as an operand and it returns an object expressing a regular expression that accepts the given letter. It is used as implicit type coercion. The two `Regex` literals, `h` and `e`, are tied by the nameless operator `[ _:Regex _:Regex ]:Regex` at (A) in Figure 3.4. The nameless operator takes two operands of type `Regex`, and it expresses a sequence of regular expressions. In this part, it takes the two `Regex` literals, `h` and `e`, as operands and it returns an object expressing a regular expression that accepts `he`. `l+` forms a literal of a regular expression constructed by a postfix unary operator `[ _:Regex "+" ]:Regex` shown at (C) in Figure 3.4. It represents a regular expression that accepts one or more sequences of `l`. Then `he` and `l+` are tied by `[ _:Regex _:Regex ]:Regex`, and they make a literal expressing `hel`, `hell`, `helll`, and so on. Finally, `hel+` and `o` make a literal that expresses the complete regular expression by `[ _:Regex _:Regex ]:Regex`.

Protean operators are overloaded by their return types and their parameter types. Overloading by return type allows defining operators that have the same syntax but a different return type. The interpretation of an expression is changed by the expected type there. This fact is useful for developing internal DSLs since an operator is used only where it is required. For example, an expression `hel+o` can be interpreted as either of the following two patterns:

```
int hel = 2;
int o = 3;
int   x1 = hel+o;    // 5
Regex x2 = hel+o;    // helo, hello, helllo, ...
```

The expression `hel+o` in the third line is interpreted as an addition expression of integers since the right hand of the assignment expects an integer value. Only

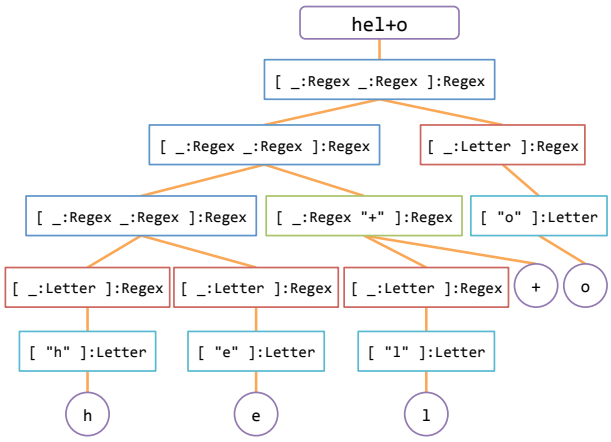

Figure 3.5: The parse tree for the literal `hel+o`

the expression `hel+o` in the fourth line is interpreted as a regular expression literal since a Regex object is expected. It can be considered that the expected type of an expression determines the parsing of the expression.

If two protean operators share the same return type, the user must specify the *parsing precedence* among them. This precedence determines which operator should be selected when multiple interpretations are possible during parsing. In ProteaJ, the earlier declared operator has the higher parsing precedence. For example, the possessive quantifier `[ _:Regex "++" ]:Regex` has higher precedence than the greedy quantifier `[ _:Regex "+" ]:Regex` since `[ _:Regex "++" ]:Regex` is a special case of `[ _:Regex "+" ]:Regex`. An operator with higher precedence is applied for parsing before operators with lower precedence. If the operator with higher precedence is successfully applied, then the other operators with lower precedence are not applied. The literal `hel++o` is interpreted as `he(l++)o` by applying `[ _:Regex "++" ]:Regex` rather than `he((l+)+)o` by `[ _:Regex "+" ]:Regex` since the former has higher precedence. The literal `hel+o` is interpreted as `he(l+)o` since `[ _:Regex "++" ]:Regex` is applied first and fails and then `[ _:Regex "+" ]:Regex` is successfully applied. Note that parsing precedence is different from *operator precedence*. Operator precedence is an useful feature for operator composition. Since a rule including operator precedence can be translated into a rule without it, we do not consider it is a core feature. We will mention it in detail in section 3.3.4.

A drawback of protean operators is a limited kind of places where the operators are available. Protean operators are available only in an expression whose expected type is statically determined before parsing the expression. The places where protean operators are available depend on the host language. For example, in typical general purpose languages such as Java, protean operators can be used in the right-hand side of an assignment but they cannot be used in the left hand of an assignment. The expected type of the right hand of an assignment

is determined since it is the same type of the left-hand side. However, the expected type of the left-hand side of an assignment is not known before parsing the assignment expression. If we use a protean operator on the left-hand side of an assignment, the compiler emits a parse error. It is a drawback that the compiler cannot distinguish between a syntax error and a type error. Table 3.1 lists the expected types of every kind of expressions in Java. It reveals that protean operators are available in any kind of expression in Java except the left-hand side of an assignment, the target of a member access, and the operand of a cast. Since the left-hand side of an assignment is usually a simple expression, programmers rarely want to use protean operators there. The target of a member access could be a complicated expression like:

```
boolean b = (hel+o).matches("hello");
```

In such case, the programmers must rewrite the code as follows:

```
Regex r = hel+o;
boolean b = r.matches("hello");
```

Or, they must rewrite by using another protean operator as follows:

```
boolean b = hel+o matches "hello";
```

Here, `matches` is a binary infix operator of type `boolean`. In Java, protean operators are not available in the operand of a cast operator. A cast operator that expresses a type conversion from `S` (source) to `T` (target) explicitly takes the target type `T` as an argument but it does not take the source type `S`. Thus, the compiler cannot know the expected type of the operand of a cast since it is the source type `S`. Look at the following example:

```
int a = (int)(sin 0.0);
```

Programmers intend that the type of `(sin 0.0)` is `double` but the compiler cannot know the expected type of `(sin 0.0)`. It is because the source type `double` of the cast `(int)` is implicitly specified in Java. If the cast operator explicitly specified the source type as follows:

```
int a = (double -> int)(sin 0.0);
```

Then the expected type of `(sin 0.0)` would be known as `double`.

In the argument of a `throw` statement in Java, it is difficult to determine available protean operators properly. According to Table 3.1, the expected type of the argument of a `throw` statement is `Throwable`; however, it is not proper because it must throw either an `Error`, a `RuntimeException`, an exception declared in the `throws` clause, or an exception caught in surrounding `catch` clauses. Our current compiler does not consider this.

This drawback, protean operators are available only in an expression whose expected type is statically determined, also makes an obstacle to use Java generics. Assuming that the generic type `List[T]` is available, we would like to define the following operator:

```
[ "length" "of" _:List[T] ]:int
```

Place	Expected type
left hand of an assignment	*unknown*
right hand of an assignment	the left-hand side type
target of a method call	*unknown*
target of a field access	*unknown*
operand of a cast	*unknown*
argument of a method call	corresponding parameter type
argument of a constructor	corresponding parameter type
argument of an operator	corresponding parameter type
condition of if, for, while	boolean
argument of switch, case	char or int
argument of throw	Throwable
return expression	the return type of the method
statement expression	void
initial value of a field	the field type

Table 3.1: The expected types of Java expressions

In this operator, the type parameter `T` cannot be inferred from the return type. Hence, the expected type `List[T]` of the operand of this operator cannot be determined. So we cannot use a protean operator in that operand. Since we currently do not have a good solution of this problem, our compiler introduced in section 3.4 does not support generics.

3.3.2 Parsing

To efficiently parse an expression including protean operators, we developed a parsing method based on packrat parsing [19] supporting left recursion [81]. Our parsing method is a recursive descent parsing with backtracking and it considers type information. In this section, we do not regard operator precedence since a grammar having operator precedence can be translated to a grammar that does not have it (see section 3.3.4). For simplicity, we use a host language that has only variables and some control flow statements. We will describe in the next section how to apply our method into a practical language such as Java.

Before parsing statements, the definition of protean operators available in the program is parsed. The compiler parses the definitions excluding their body parts. It collects the meta-information of the protean operators such as syntax and type signature. The collected information is sorted by return types and parsing precedence for later use.

The parser first attempts to parse a given piece of code as a statement such as if until it encounters a non-terminal symbol representing an expression. Protean operators cannot be used for statement-level syntax since a protean operator and its operands constitute only an expression. The statements are parsed by using only the syntax rules of the host language. Once the parser encounters an expression, it first determines the expected type of the expression by analyzing

the code that the parser has already read. For example, an assignment statement is parsed by this rule:

```
Assignment → Id "=" Expr
```

The right-hand side of the statement `Expr` is parsed under the expected type obtained from the L-value, in this case, the variable named `Id`.

The parser first chooses a protean operator that returns the expected type and has the highest parsing precedence. Then it attempts to parse the expression by assuming that the expression is of the chosen operator. If this attempt succeeds, the parser returns the resulting parse tree of the expression. If it fails, the parser backtracks and tries a protean operator with the next highest parsing precedence. If there is no other operator, the parser parses the expression by using the host language syntax for expressions. Each attempt at parsing with a protean operator is to do the following action sequentially for each element of the syntax of the operator:

- An operator name `"n"` : read tokens by assuming that they match `n`
- An operand `_:T` : parse the successive tokens as an expression of the expected type `T`

Figure 3.6 shows a pseudo code of the parsing algorithm for statements. We assume that a host language supports several control flow statements such as while and it also supports local variables. The procedure `parseStmt` is an entry point of the parser. The procedure `parseWhileStmt` parses a while statement. Since a condition expression in the while statement must return a `boolean` value, the expected type of the condition expression is `boolean`. Thus the call `parseExpr(`*Boolean*`, ops, env)` parses it. The procedure `scan` performs token analysis and returns `Success` if the next token matches the given string, otherwise `Failure`. The procedure `parseVarDecl` parses a local variable declaration. An initialization expression of the declaration is parsed by using expected type information that is specified as the variable type. The name and the type of the variable is stored into the environment `env`. The procedure `parseExpr` parses an expression. It takes an expected type as a parameter and attempts to parse an expression returning a value of that type. If all the attempts fail, it calls another procedure `parseExprByPredefinedRule` to parse the expression in the host language rules. The procedure `parseExprByOperator` parses according to the syntax of each protean operator. If it encounters an operand, it recursively calls parseExpr. It passes the operand type to parseExpr as the expected type.

In this figure, memoization is not shown for simplicity; however, it can be easily applied to the algorithm. To apply memoization, the algorithm must be modified so that the result will be memoized before it is returned and parseExpr will first look up the memoization table to avoid redundant parsing attempts.

3.3.3 Parsing Speed and Expressiveness

Our parsing method is sufficiently fast to parse protean operators even if they express user-defined literals since the operators can be regarded as Parsing Ex-

```
// entry point
// ops is operators collected before parsing
// env is variable environment
def parseStmt(ops, env) {
  r = parseWhileStmt(ops, env)
  if (r is Success) return r
  else backtrack
  [ parse by the other control flow rules ]
  r = parseVarDecl(ops, env)
  if (r is Success) return r
  else backtrack
  [ parse by the other statement rules ]
  r = parseExprStmt(ops, env)
  if (r is Success) return r
  return Failure
}

// WhileStmt
// "while" "(" Expr<Boolean> ")" Stmt
def parseWhileStmt(ops, env) {
  w = scan("while")
  l = scan("(")
  c = parseExpr(Boolean, ops, env)
  r = scan(")")
  s = parseStmt(ops, env)
  if (w is Success && l is Success &&
   c is Success && r is Success &&
   s is Success) return WhileStmt(c, s)
  else return Failure
}

// VarDecl
// TypeName<T> Identifier "=" Expr<T>
def parseVarDecl(ops, env) {
  t = parse by the identifier rule
  n = parse by the identifier rule
  e = scan("=")
  v = parseExpr(get a type of t, ops, env)
  if (t is Success && n is Success &&
   e is Success && v is Success) {
    add a variable n whose type is t to env
    return VarDecl(t, n, v)
  }
  else return Failure
}

// ExprStmt
// Expr<Void> ";"
def parseExprStmt(ops, env) {
  e = parseExpr(Void, ops, env)
  s = scan(";")
  if (e is Success && s is Success)
    return ExprStmt(e)
  else return Failure
}
```

```
// typ is expected type
def parseExpr(typ, ops, env) {
  ops_t = ops.filter(op -> op re-
turns typ)
  for (op in
       ops_t sorted by parsing precedence) {
    r = parseExprByOperator(op, ops, env)
    if (r is Success) return r
    else backtrack
  }
  return parseExprByPredefinedRule(typ, ops, env)
}

// op is an operator
def parseExprByOperator(op, ops, env) {
  for (e in the syntax of op) {
    if (e is an operator-name) {
      if (scan(e to string) is Failure)
        return Failure
    }
    else if (e is an operand) {
      r = parseExpr(e's type, ops, env)
      if(r is Failure) return Failure
      else append r to the parse tree
    }
  }
  return the parse tree
}

// variable access rule is a predefined
def parseExprByPredefinedRule(typ, ops, env) {
  r = parse by the identifier rule
  v = get a variable r from env
  if (r is Success && v's type is typ)
    return VarAccess(v)
  else backtrack
  [ parse by any other predefined rules ]
  return Failure
}
```

Figure 3.6: the parsing algorithm for statements

pression Grammar (PEG) [20] with left recursion as shown later. Our parsing method is a scannerless recursive descent parsing with memoization. Memoization is used for eliminating the cost of backtracking. Recursive descent parsing with memoization is called packrat parsing. It is an efficient parsing method and its time complexity is O(n). The original packrat parsing does not support left recursion but it can support left recursion by a small extension according to the paper [81]. Although the worst-case time complexity of the packrat parsing supporting left recursion is not O(n), it is still sufficiently fast since such a case hardly occurs in practical programming languages according to the paper. It is sufficiently fast to apply scannerless parsing; the parsing time is O(n), where n is the number of characters in the program, in most cases.

The expressiveness of protean operators is equivalent to PEG. Any protean operator can be expressed by PEG syntax and any PEG syntax can be expressed by protean operators. Each rule of PEG has the form $A \leftarrow e$, where A is a non-terminal symbol and e is a parsed expression. A parsed expression consists of terminal symbols, non-terminal symbols, the empty string, sequence operators e_1e_2, and ordered-choice operators e_1/e_2. Here, e_1 and e_2 are parsed expressions. The other operators such as optional operators can be expressed by the above operators.

We can translate any protean operator to PEGs by replacing the types of the protean operator with non-terminal symbols. For example, the following protean operator:

```
[ _:Regex "+" ]:Regex
```

can be translated into the following PEG syntax:

```
Expr<Regex> → Expr<Regex> "+"
```

Here, *Expr<Regex>* denotes a non-terminal symbol representing an expression of the expected type Regex. Protean operators with different return types are translated into PEG rules with different non-terminal symbols in the left-hand side of →. If an operator returns Letter, it is translated into a rule for non-terminal symbol *Expr<Letter>*. The parsing precedence is translated into the ordered-choice rule in PEG. For example, see the following protean operators:

```
[ _:Regex "++" ]:Regex
[ _:Regex "+" ]:Regex
```

Here, the two different protean operators return the same type. The first operator has higher parsing precedence than the second operator because the earlier declared operator has the higher parsing precedence. We translate these operators into the following PEG syntax:

```
Expr<Regex> → Expr<Regex> "++"
            | Expr<Regex> "+"
```

Note that the ordered choice | chooses the left operand first and then the right operand. So an operator with a higher precedence is placed at the left-hand side of | and an operator with a lower precedence is placed at the right-hand side of |.

PEG			protean operators
parsing rule	$A \leftarrow e$	$\rightarrow$	an operator op that returns A and the syntax of op is e
terminal	a	$\rightarrow$	an operator-name "a"
non-terminal	T	$\rightarrow$	an operand $\_ : T$
empty string	ε	$\rightarrow$	an operator-name ""
sequence	$e_1 e_2$	$\rightarrow$	a sequence $e_1 e_2$
ordered-choice	e_1 / e_2	$\rightarrow$	an operand $\_ : X$ and operators $op_1 > op_2$ op_1 and op_2 return X and the syntax of op_i is e_i

Table 3.2: The translation from PEGs to protean operators

On the other hand, any PEG rule can be translated into a protean operator. Table 3.2 presents the translation from PEG to protean operators. In this table, $op_1 > op_2$ denotes that op_1 has a higher parsing precedence than op_2. A terminal symbol in PEG is translated into an operator name of a protean operator. A non-terminal symbol at the left-hand side of $\rightarrow$ is translated into a return type of an operator while a non-terminal symbol at the right-hand side is translated into an operand type. The left and right operands of an ordered choice are translated into two distinct protean operators. An operator for the left has a higher parsing precedence than an operator for the right.

3.3.4 Operator precedence and associativity

Since the static types of protean operators serve as non-terminal symbols of PEG, programmers define a new type and operators when they need to define a new non-terminal symbol for new syntax. However, this often causes too many types defined for non-terminal symbols, which are used to resolve ambiguities of the grammar with respect to precedence order. For example, consider the addition and multiplication operator of integers. A multiplication can be used as an operand of the addition operator but an addition cannot be used as an operand of the multiplication operator. To express this syntax, programmers would define new types extending the integer type and use them as the return type and the operand types of the operators. Programmers would also define new operators that express the relationship between the new types and the integer type. The following operators express the syntax:

```
[ _:IntAdd ]:int
[ _:IntAdd "+" _:IntMul ]:IntAdd
[ _:IntMul ]:IntAdd
[ _:IntMul "*" _:IntVal ]:IntMul
[ _:IntVal ]:IntMul
```

In this example, `IntAdd`, `IntMul`, and `IntVal` are types extending the integer type. The first, third, and fifth operators express the relationship between these

types and the integer type.

Operator precedence and associativity are useful features for avoiding such verbosity. They express which operators may be used for building an expression for an operand of an operator. In general, an expression of an operator A is occurred at an operand of an operator B only if A has a higher or equal precedence than B. We support three kinds of associativity for expressing practical syntax; right-associativity, left-associativity, and non-associativity. If an operator B is left-open (meaning that no operator name precedes the left-most operand) and right-associative, an expression of an operator A is occurred at the left-most operand of B only when A has a strictly higher precedence than B. If an operator B is right-open and left-associative, an expression of an operator A is occurred at the right-most operand of B only when A has a strictly higher precedence than B. If the operator B is non-associative, an expression of an operator A is occurred at an operand of B only when A has a strictly higher precedence than B. Figure 3.7 shows an example of protean operators with operator precedence and associativity. It is a complete version of Figure 3.4.

Protean operators with operator precedence and associativity can be translated into operators without them. We below show how to translate them. Assume that operator precedence is represented by a non-negative integer number and a larger number indicates higher precedence. The following is a translation from a protean operator `[S]:T` having operator precedence `P` and associativity `A`. The operator syntax `S` involves n operands and each operand has a type $\mathtt{T}^i$. First, the return type `T` is translated into a type $\mathtt{T_P}$. Here, the subscript `P` is a non-negative integer number that is equivalent to the operator precedence. If the operator is left-open and operator associativity `A` is *right-assoc*, the left-most operand $\mathtt{\_:T}^1$ is translated into $\mathtt{\_:T}^1_{\mathtt{P}+1}$ and the other operand $\mathtt{\_:T}^i$ in the operator syntax `S` is translated into an operand $\mathtt{\_:T}^i_{\mathtt{P}}$. If the operator is right-open and operator associativity `A` is *left-assoc*, the right-most operand $\mathtt{\_:T}^n$ is translated into $\mathtt{\_:T}^n_{\mathtt{P}+1}$ and the other operand $\mathtt{\_:T}^i$ in the operator syntax `S` is translated into an operand $\mathtt{\_:T}^i_{\mathtt{P}}$. If the operator associativity `A` is *non-assoc*, each operand $\mathtt{\_:T}^i$ is translated into $\mathtt{\_:T}^i_{\mathtt{P}+1}$. If the operator is not left-open but *right-assoc* or the operator is not right-open but *left-assoc*, each operand $\mathtt{\_:T}^i$ is translated into $\mathtt{\_:T}^i_{\mathtt{P}}$.

For example, the following operator:

```
[ _:Regex _:Regex ]:Regex { left-assoc }
```

with the operator precedence 0, is translated into:

$$\mathtt{[\ \_:Regex_0\ \_:Regex_1\ ]:Regex_0}$$

Next we add an addition operator $\mathtt{[\_:T_P]:T_{P-1}}$ for each return type $\mathtt{T_P}$ if P is not 0. This operator converts its operand to a value of type $\mathtt{T_{P-1}}$ and returns it. Note that the parsing precedence of the added operator $\mathtt{[\_:T_P]:T_{P-1}}$ is set to the lowest among the operators with the return type $\mathtt{T_{P-1}}$. Finally, we add the operator $\mathtt{[\_:T_0]:T}$ for each return type $\mathtt{T_0}$. It converts an operand from $\mathtt{T_0}$

```
(A) [ _:Regex _:Regex ]:Regex { left-assoc }
(B) [ _:Regex "++" ]:Regex { non-assoc }
(C) [ _:Regex "+" ]:Regex { non-assoc }
(D) [ _:Letter ]:Regex { non-assoc }
(E-a) [ "a" ]:Letter
(E-b) [ "b" ]:Letter
  ...
(E-z) [ "z" ]:Letter

operator precedence:
  (A) < (B) = (C) < (D) < (E-a) = (E-b) = ... = (E-z)
```

Figure 3.7: protean operators with operator precedence and associativity expressing regular expression literals

to `T`. For example, the protean operators in Figure 3.7 are translated into the operators in Figure 3.8.

3.4 Implementation: ProteaJ

We have developed *ProteaJ*, which is a subset language of Java but supports protean operators. ProteaJ recognizes a single character as a token. It enables protean operators to express user-defined literals. In this section, we describe how we have adapted protean operators into Java.

ProteaJ provides a module system called *operator modules* to implement and export user-defined operators; an operator module expresses a DSL. Programmers can use the DSL by importing the module with using declaration. There are two different precedence rules, *operator precedence* and *parsing precedence*, in ProteaJ. Operator precedence is specified by an integer value for each operator. The value is used only for comparing precedence among operators in the same operator module. The entire operator precedence is determined by the order of using declarations. Since DSLs are composable in ProteaJ, operator precedence of each DSL is independent. Parsing precedence of protean operators is specified by the order of their definitions. It is a precedence among operators with the same operator precedence. In ProteaJ, parsing precedence is also needed among operators with different return type because any class type in Java is a subtype of `Object`. Although our parsing method introduced in the previous section does not consider subtype relationship, it is an important feature of Java. To handle subtype relationship, the ProteaJ compiler automatically generates operators that express subtype relationship and translates the return types of user-defined operators. ProteaJ provides some useful features for convenience. For example, ProteaJ provides two kinds of protean operators: *expression operators* and *readas operators*. An expression operator is an operator that recognizes a white space as a special token that is a separator. In contrast, a readas operator recognizes a white space as a normal token. A

```
[ _:Regex0 ]:Regex
[ _:Regex0 _:Regex1 ]:Regex0
[ _:Regex1 ]: Regex0
[ _:Regex2 "++" ]:Regex1
[ _:Regex2 "+" ]:Regex1
[ _:Regex2 ]: Regex1
[ _:Letter3 ]:Regex2
[ _:Letter0 ]:Letter
[ _:Letter1 ]:Letter0
[ _:Letter2 ]:Letter1
[ _:Letter3 ]:Letter2
[ "a" ]:Letter3
 ...
[ "z" ]:Letter3
```

Figure 3.8: The definition of the regular expression literals without operator precedence or associativity (the translation from Figure 3.7)

readas operator is used for building user-defined literals.

We give some examples of DSLs that are implemented in ProteaJ to show the expressiveness of the protean operators. We also give examples in which multiple DSLs are used. We have implemented the compiler of ProteaJ in Java. ProteaJ does not support generics since there is a problem when protean operators and generics are used together (see section 3.3.1). ProteaJ also does not support inner classes because they complicate the compiler. For the same reason, ProteaJ does not support annotations and the other facilities introduced in Java 1.5 or above.

3.4.1 Definitions of Protean Operators

The definitions of protean operators in ProteaJ are similar to the class and method definitions in Java. Figure 3.9 shows definitions of protean operators that express regular expressions. This code defines an operator module named `RegexOperators`. This module defines four protean operators. For example, the third one defines the greedy quantifier operator `[ _:Regex "+" ]:Regex`. The keyword `readas` [1]indicates that this operator expresses a user-defined literal. It specifies that a white space is recognized as a normal token rather than a token separator. The details on `readas` are mentioned later (see 3.4.2). `Regex` next to `readas` represents the return type of the operator. The following part `r "+"` represents the syntax of the operator. The identifier `r` represents the operand of the operator and the double-quoted string `"+"` represents an operator name of the operator. The parameter type of the operand `r` is described in the following part enclosed in parentheses `(Regex r)`. It denotes that the type of the operand named `r` is `Regex`. The following `: priority = 250` represents operator precedence. The remaining part enclosed in curly braces is an operator

```
operators RegexOperators {
  readas Regex rs+ (Regex... rs): priority = 200 {
    return new RegexList(rs);
  }
  readas Regex r "++" (Regex r): priority = 250 {
    return new RegexPossessivePlus(r);
  }
  readas Regex r "+" (Regex r): priority = 250 {
    return new RegexPlus(r);
  }
  readas Regex l (Letter l): priority = 300 {
    return new Regex(l);
  }
}
```

Figure 3.9: Definition of protean operators expressing regular expressions

body. It is equivalent to the method body of a method declaration.

Figure 3.10 shows syntax of an operator declaration in ProteaJ. In ProteaJ, an operator is defined in an operator module. An operator declaration consists of two parts, a header and a body. The body part is described as a method body. The header part consists of modifiers, a return type, syntax, throwable exceptions, and an operator priority. Protean operators can have modifiers rassoc, nonassoc, and readas. The modifiers rassoc and nonassoc specify operator associativity: rassoc specifies right-associative and nonassoc specifies non-associative. The default associativity is left-associative. ProteaJ provides several notations like PEG notations for describing the syntax of the operator more concisely. `?`, `*`, and `+` are annotations that follows operands. `?` indicates an optional operand of the operator. It is used with a default argument as follows:

```
readas Regex r "+" a? (Regex r, Anno a = Anno.greedy)
: priority = 250 {
  return new RegexPlus(r, a);
}
readas Anno "+" () : priority = 300 {
 return Anno.possessive;
}
```

`*` indicates zero or more repetitions, and `+` indicates one or more repetitions. They are used with variable arguments. The lines from 2 to 4 in Figure 3.9 is an example using `+`. The operator `[ _+:Regex ]:Regex`, which concatenates one or more regular expressions, is defined there. ProteaJ also provides *predicate* for building syntax of an operator. `&` and `!` are predicates and they take a type following the symbol. They represent look-ahead; they test the next inputs

[1]The keyword `readas` means that the parser *read*s the next input *as* an instance of a specified type.

```
OpModule → "operators" Id "{" OpDef* "}"
OpDef    → Header Body
Header   → Mod* Type Syntax Params Throws Prty
Mod      → "rassoc" | "nonassoc" | "readas"
Syntax   → ( OpName | Operand | Opt | Rep | Pred )+
OpName   → StringLiteral
Operand  → Id
Opt      → Id "?"
Rep      → Id ( "*" | "+" )
Pred     → ( "&" | "!" ) Type
Params   → "(" Param ( "," Param )* ")"
Param    → Type VarArgs? Id DfltArg?
VarArgs  → "..."
DfltArg  → "=" Expr
Prty     → ":" "priority" "=" IntConst
```

Figure 3.10: Syntax of an operator declaration in ProteaJ

```
using RegexOperators;
using FilePathOperators;
using GrepOperators;

GrepResult r = grep -i hel+o ~/src/Main.java;
```

Figure 3.11: Multiple operator modules can be imported in one source file

and determine whether parsing fails or continues depending on the test result. Either way, they do not consume the inputs. `& T` is a predicate that tries to parse the next inputs assuming that the expected type is the given type `T` and parsing fails when the look-ahead fails. `! T` is similar to `& T` but it fails when the look-ahead succeeds.

To use protean operators, using declaration is needed to import an operator module. For example, regular expression literals defined in Figure 3.9 can be used as follows:

```
using RegexOperators;
...
Regex r = hel+o;
```

the protean operators defined in `RegexOperators` are used for the code `hel+o`. The using declaration is written at the beginning of programs. Multiple operator modules can be imported in one source file by writing multiple using declarations. For example, Figure 3.11 shows a program that imports `GrepOperators`, `RegexOperators`, and `FilePathOperators` together.

3.4.2 Expression Operators and Readas Operators

In ProteaJ, protean operators can be divided into two categories: *expression operators* and *readas operators*. The definitions of an expression operator and a readas operator are almost same but the definition of the readas operator contains the modifier `readas`. When parsing an expression operator, a white space is recognized as a separator of tokens. On the other hand, when parsing a readas operator, a white space is a token. An operand of a readas operator must be an expression of a readas operator. Readas operators are mainly used for defining literals. Expression operators are not suitable for that since white spaces are not allowed in a number of literals. For convenience, if `readas` is not specified, a white space is automatically recognized as a separator. The operators defined without readas are called expression operators.

To parse both of expression operators and readas operators, the parser of the ProteaJ compiler parses an expression as follows:

1. parse the expression by expression operators
2. if 1 fails, parse the expression by the expression rules in the host language (e.g. method call)
3. if 2 fails, parse the expression by readas operators
4. if 3 fails, parse the expression by the literal rules in the host language (e.g. string literal)

In 1 and 3, the parser works in our parsing method introduced in the previous section. White spaces are recognized as a token separator in 1 and 2 but they are recognized as a normal token in 3 and 4. In 1, an operand of an operator is parsed by 1. In 3, an operand of an operator is parsed by 3. An argument of a method call or a constructor call, which can be members of the expression rules, is parsed by 1. Since the compiler knows the expected type of each argument, protean operators are available there. Note that, if the type of the parsing result is not the expected type, 2 also fails.

According to this parsing method, the predefined expression rules have a higher priority over user-defined literals. For example, a variable access has a higher priority over a regular expression literal as follows:

```
Regex r = hel+o;   // helo, hello, helllo, ...
Regex r2 = r;      // helo, hello, helllo, ...
```

This is reasonable in most cases, we believe, since a conflict between a variable and a literal occurs only when they have the same type. ProteaJ adopts the solution of this conflict that gives a higher priority to variables in this case. We can also give priority to user-defined literals by using a certain trick. When we define the following expression operator:

```
Regex2 r (Regex r) : priority = 0 {...}
```

we can use regular expression as follows:

```
Regex2 r = hel+o;   // helo, hello, helllo, ...
Regex2 r2 = r;      // r
```

`r` can be parsed as both of a variable `r` and an expression of the expression operator `[ _:Regex ]:Regex2`. Since an expression operator has a higher priority over the predefined expression rules, `r` is parsed as an expression of the expression operator `[ _:Regex ]:Regex2`. The operand of the operator is parsed as a regular expression literal rather than a variable `r` since the operand type of the operator is `Regex` rather than `Regex2`.

3.4.3 Operator Precedence and Parsing Precedence

ProteaJ has two different precedence rules, *operator precedence* and *parsing precedence*. Operator precedence is specified by an integer value for each operator. For example, `: priority = 250` in Figure 3.9 represents operator precedence. In ProteaJ, as the value of a precedence is larger, the binding of an operator is tighter. For example, the third operator in Figure 3.9 `[ _:Regex "+" ]:Regex` is bound tighter than the first operator `[ _+:Regex ]:Regex`.

The value representing operator precedence is used only for comparing precedence among operators in the same operator module. The entire operator precedence is determined by the order of `using` declarations. Operators imported earlier bind tighter than operators imported later. For example, operators that express regular expression literals bind tighter than `grep` operator in Figure 3.11. This feature improves composability of DSLs since operator precedence of each DSL is independent of each other.

Parsing precedence of protean operators is specified by the order of their definitions. It is a precedence among operators with the same operator precedence. The precedence of an operator defined earlier is higher. For example, the second operator in Figure 3.9 `[ _:Regex "++" ]:Regex` has higher parsing precedence than the third operator `[ _:Regex "+" ]:Regex`. Although parsing precedence was given to operators with the same return type in section 4.3, we here change this because all operators in ProteaJ return a subtype of the same type `Object`.

3.4.4 Subtype Relationship between an Expected Type of the Expression and the Return Type of an Operator

Programmers may want to use an operator in an expression whose expected type is a supertype of the return type of that operator. For example, a programmer might want to use operators in Figure 3.9 as follows:

```
using RegexOperators;
...
Regex r = hel+o;
Object obj = hel+o;
```

However, our parsing method introduced in the previous section does not consider subtype relationship between the expected type of an expression and the

return type of an operator. In the last line of the example above, the expected type of the right-hand side expression of the equal sign is `Object`. Since the return type of each operator defined in `RegexOperators` is `Regex` rather than `Object`, our parsing method does not select these operators when parsing the expression.

To support subtypes, the ProteaJ compiler automatically generates operators that express subtype relationship and translates the return types of user-defined operators after the translation explained in section 3.3.4. First, the return type $\texttt{T}_\texttt{P}$ of a user-defined operators is translated into $\texttt{T}_\texttt{P}^\texttt{Q}$. Here, the subscript `P` is a non-negative integer number that is equivalent to the operator precedence. The superscript `Q` is an integer number that expresses the parsing precedence of the operator and a larger number indicates higher precedence. Then for every $\texttt{T}_\texttt{P}^\texttt{Q}$, the compiler generates an operator $\texttt{[\_:T}_\texttt{P}^\texttt{Q}\texttt{]:Super}_\texttt{P}$ for each `T`'s supertype `Super`. The parsing precedence of the generated operator is higher than the operator $\texttt{[\_:Super}_{\texttt{P}+1}\texttt{]:Super}_\texttt{P}$, which is added by the translation explained in section 3.3.4. When `Q1` is larger than `Q2`, $\texttt{[\_:T}_\texttt{P}^\texttt{Q1}\texttt{]:Super}_\texttt{P}$ has higher parsing precedence than $\texttt{[\_:T}_\texttt{P}^\texttt{Q2}\texttt{]:Super}_\texttt{P}$. For example, the protean operators in Figure 3.8 are translated into the operators in Figure 3.12.

The reader might think there is a simpler approach, for example, generating an operator like $\texttt{[\_:Regex}_\texttt{P}\texttt{]:Object}_\texttt{P}$ for each operator precedence `P`, but it does not work. This approach might break operator precedence. According to section 3.3.4, the operators $\texttt{[\_:Regex}_1\texttt{]:Regex}_0$ and $\texttt{[\_:Regex}_2\texttt{]:Regex}_1$ are automatically generated. So an operator with the return type $\texttt{Regex}_2$ is selected before an operator with the return type $\texttt{Object}_1$. This approach might also break parsing precedence. The compiler cannot determine the parsing precedence between $\texttt{[\_:Regex}_\texttt{P}\texttt{]:Object}_\texttt{P}$ and $\texttt{[\_:String}_\texttt{P}\texttt{]:Object}_\texttt{P}$.

The parser of ProteaJ is still sufficiently fast although it supports subtype relationship. Operators supporting subtype relationship can be translated into protean operators without the support as shown above since all user-defined operators are totally ordered by the operator precedence and the parsing precedence in ProteaJ. Even if the translation generates a large number of operators that return `Object` type, it does not cause a large impact on performance in a practical case since the parser chooses those operators one-by-one in the order of the precedence and hence the parsing time is a proportion to the number of expressions expecting `Object` type. The number of such expressions are usually not many in a statically typed language.

3.4.5 Case Study

In the rest of this section, we show several DSLs implemented in ProteaJ.

Ruby-like print statement

In ProteaJ, programmers can define a new statement since ProteaJ allows them to define an operator returning `void`. Programmers can use such an operator for building a user-defined statement since the expression type of a *statement*

```
[ _:Regex0 ]:Regex
[ _:Regex0^0 ]:Regex0
[ _:Regex1 ]:Regex0
[ _:Regex0 _:Regex1 ]:Regex0^0
[ _:Regex1^1 ]:Regex1
[ _:Regex1^0 ]:Regex1
[ _:Regex2 ]:Regex1
[ _:Regex2 "++" ]:Regex1^1
[ _:Regex2 "+" ]:Regex1^0
[ _:Regex2^0 ]:Regex2
[ _:Letter3 ]:Regex2^0

[ _:Letter0 ]:Letter
[ _:Letter1 ]:Letter0
[ _:Letter2 ]:Letter1
[ _:Letter3 ]:Letter2
[ _:Letter3^25 ]:Letter3
 ...
[ _:Letter3^0 ]:Letter3
[ "a" ]:Letter3^25
 ...
[ "z" ]:Letter3^0

[ _:Object0 ]:Object
[ _:Regex0^0 ]:Object0
[ _:Object1 ]:Object0
[ _:Regex1^1 ]:Object1
[ _:Regex1^0 ]:Object1
[ _:Object2 ]:Object1
[ _:Regex2^0 ]:Object2
[ _:Object3 ]:Object2
[ _:Letter3^25 ]:Object3
 ...
[ _:Letter3^0 ]:Object3
```

Figure 3.12: The definition of the regular expression literals supporting subtype relationship (the translation from Figure 3.8)

expression is `void` in ProteaJ. A statement expression is a statement that consists of only one expression. A statement expression whose expression is built by a user-defined operator looks like a user-defined statement. The following code is a definition of an operator returning `void`:

```
operators OutputOperators {
  void "p" msg (String msg): priority = 0 {
    System.out.println(msg);
  }
}
```

and we can use it as follows:

```
using OutputOperators;
...
p "Hello world!";
```

In the code above, the last line is a *statement expression*. We can use `p` to build a statement taking a string and printing this string since `OutputOperators` provides the operator `[ p _:String ]:void`.

Regular Expression

Programmers can define complex literals by using readas operators. For example, regular expression literals can be defined as in Figure 3.13. This operator module `RegexOperators` provides `Regex` literals, which express regular expressions. The following code is an example using `RegexOperators`:

```
using OutputOperators;
using RegexOperators;
...
Regex stnumber = [0-9]{2}(B|M|D)[0-9]{5};
Matcher m = stnumber.matcher(text);
if(m.find()) {
  p "match : " + m.group();
}
```

`Regex` literals are used in the statement in line 4 in the code above. This regular expression literal consists of several operators: `[ _+:Regex ]:Regex`, `[ _:Regex | _:Regex ]:Regex`, `[ _:Regex { _:Nat }]:Regex`, `[ [ _+:ClsElm ] ]:Regex`, and so on. Parentheses operator `( _ )` is an operator provided by ProteaJ. It resets operator precedence of the operand.

Simple Optimization

Another usage of protean operators is performance optimization. For example, the binary operator `[ _:String + _:String]:String`, which is used for string concatenation, is not efficient when it is successively used more than once. To make string concatenation more efficient, we should instead use the `StringBuilder` class. Protean operators in ProteaJ can be used in this case.

```
operators RegexOperators {
 readas Regex l "|" r (Regex l, Regex r): priority = 100
 readas Regex rs+ (Regex... rs): priority = 200
 readas Regex r "?+" (Regex r): priority = 250
 readas Regex r "*+" (Regex r): priority = 250
 readas Regex r "++" (Regex r): priority = 250
 readas Regex r "??" (Regex r): priority = 250
 readas Regex r "*?" (Regex r): priority = 250
 readas Regex r "+?" (Regex r): priority = 250
 readas Regex r "?" (Regex r): priority = 250
 readas Regex r "*" (Regex r): priority = 250
 readas Regex r "+" (Regex r): priority = 250
 readas Regex r "{" n "}" (Regex r, Nat n): priority = 250
 readas Regex "[" es+ "]" (ClsElm... es): priority = 270
 readas ClsElm f "-" t (Letter f, Letter t): priority = 280
 readas ClsElm l (Letter l): priority = 300
 readas Regex "." (): priority = 300
 readas Regex l (Letter l): priority = 300
}
```

Figure 3.13: Regular expression literals as an internal DSL

The definition in Figure 3.14 is an operator module that defines optimized string concatenation. When the operator module is used, concatenation of two strings such as `"foo" + "bar"` is interpreted as `"foo".concat("bar")`, but concatenation of three strings such as `"foo" + "bar" + "baz"` is interpreted as the following:

```
new StringBuilder().append("foo")
 .append("bar").append("baz").toString()
```

Like this, protean operators enable us to optimize an expression that conforms to such typical patterns. An important fact is that this optimization is defined by a library, not the compiler.

```
operators ExStringOperators {
  String buf (StringBuilder buf): priority = 200
    { return buf.toString(); }
  StringBuilder l "+" r
  (StringBuilder l, String r): priority = 250
    { return l.append(r); }
  String l "+" r (String l, String r): priority = 300
    { return l.concat(r); }
  StringBuilder s1 "+" s2 "+" s3
  (String s1, String s2, String s3): priority = 350
    { return new StringBuilder().append(s1).append(s2).append(s3); }
}
```

Figure 3.14: Optimized string concatenation operators

```
// PrimitiveOperators is predefined operator module
// and it is imported implicitly like java.lang
operators PrimitiveOperators {
  ...
  int a + b (int a, int b): priority = 900 { ... }
  int a - b (int a, int b): priority = 900 { ... }
  int a * b (int a, int b): priority = 1000 { ... }
  int a / b (int a, int b): priority = 1000 { ... }
  ...
}
operators FilePathOperators {
  readas FilePath dir? name
   (DirPath dir = CurDir.v, Identifier name)
   : priority = 100 { ... }
  readas DirPath parent? name "/"
   (DirPath parent = CurDir.v, Identifier name)
   : priority = 200 { ... }
  readas DirPath dir? "./" (DirPath dir = CurDir.v)
   : priority = 200 { ... }
  readas DirPath dir? "../" (DirPath dir = CurDir.v)
   : priority = 200 { ... }
  readas DirPath "/" (): priority = 200 { ... }
  readas DirPath "~/" () : priority = 200 { ... }
}
```

Figure 3.15: File path operator module

SQL

In ProteaJ, programmers can implement more complex internal DSLs. For example, they can implement a subset of SQL. We have implemented two operator modules, `FilePathOperators` and `SQLOperators`. `FilePathOperators` module enables us to write a file path like `~/Documents/file.txt`. The definition of `FilePathOperators` is shown in Figure 3.15. `SQLOperators` module defines some SQL operators, for example, select, create table, and insert into. The definitions of these operator modules are available from our github page.[2]With these modules, programmers can write the program shown in Figure 3.16.

3.5 Experiments

3.5.1 Parsing Expressions Including User-Defined Literals

We have conducted an experiment for demonstrating that our method can efficiently parse expressions including user-defined literals even though a naive

[2]The source codes of ProteaJ and DSLs introduced in this section are available from github: `https://github.com/csg-tokyo/proteaj`

```
import java.sql.*;

using FilePathOperators;
using SQLOperators;
using OutputOperators;
using ExStringOperators;

public class Main {
  private static boolean existTable
   (String tbl) throws Exception
  {
    ResultSet tables = select tablename from sys.systables
                         where tablename = tbl.toUpperCase();
    return tables.next();
  }

  private static void insertMember
   (int id, String name) throws Exception
  {
    insert into members ( user_id, name ) values ( id, name );
  }

  public static void main(String[] args) throws Exception {
    connect to ./database.db;
    if(existTable("members")) drop table members;
    create table members (
      user_id int not null primary key,
      name varchar(64) not null
    );

    if(existTable("posts")) drop table posts;
    create table posts (
      id int not null generated always as identity,
      date timestamp default current timestamp,
      user_id int,
      comment long varchar
    );

    insertMember(123, "ichikawa");
    insertMember(345, "ohtani");
    insertMember(567, "hiramatsu");
    insert into posts ( user_id, comment )
                values ( 123, "Ohayo!" );

    ResultSet rs = select * from members;
    while(rs.next()) {
      p rs.getInt(1) + " " + rs.getString(2);
    }
    commit;
    disconnect;
  }
}
```

Figure 3.16: A program using `SQLOperators`

method cannot parse them in pragmatic time. In a typical naive method for parsing user-defined literals, a parser generates all possible parse trees and a semantic analyzer determines the most suitable tree. On the other hand, our method uses type information, which is obtained by a semantic analyzer, for parsing. The most important difference between the naive method and our method is grammar ambiguity when parsing; the naive method uses ambiguous grammar but our method uses unambiguous grammar. This experiment aims to show the performance impact caused by this difference. We used JSGLR parser [39], which is a well-known implementation of SGLR parser in Java, as a parser of the naive method for mixfix operators supporting user-defined literals. Since the parser of ProteaJ cannot be detached from the compiler, we compared the compilation time (parsing time + code generation time) by ProteaJ and the parsing time by JSGLR. The machine used for the experimentation had a 2.67GHz Core i5 processor and 8 GB of memory. The installed operating system on the machine was OpenSUSE 12.3. We used Oracle JDK 1.8.0_05.

The problem setting of the experiment is as follows:

- Grammar: basic arithmetic operators and file path literals.
  The grammar for the experiment of JSGLR is shown in Figure 3.17. ProteaJ uses the two operator modules in Figure 3.15 as an implementation of this grammar.

- Input: `a/a/a/.../a` (a sequence of `a` separated by `/`)
  The input size is the number of `a` in the input. For example, the input size of `a/a/a` is 3.
  In the experiment of ProteaJ, the input source is more complex since it should be a valid ProteaJ source code. Figure 3.18 shows the input source for ProteaJ.

- Measurement: an average parsing or compilation time of ten executions.

The grammar shown in Figure 3.17 is a simple grammar only including basic arithmetic operators and file path literals. It has ambiguities, for example, `a` can be parsed as both of a variable and a file name. `a/a` might be a division expression of two numbers, a division expression of a number and a file name, a division expression of two file names, and a file path literal. The possible parsing results of the input `a/a/.../a` explode exponentially. We designed this grammar supposing the host language has operator precedence and associativity for reducing ambiguities. When the two modules in Figure 3.15 are imported by `using` declarations, ProteaJ can parse any expressions that can be expressed by the grammar in Figure 3.17 except that a file name is not restricted. We have measured the parsing or compilation time by changing the input size.

Figure 3.19 shows the result of the experiment. It is a semilog graph. The vertical axis indicates the parsing or compilation time and the horizontal axis indicates the input size. The diamond indicates an average parsing time by JSGLR and the rectangle indicates an average compilation time (parsing time + code generation time) by ProteaJ. This graph is plotted for the input size

```
S → Expr
Expr → AddE
AddE → AddE "+" MulE | AddE "-" MulE | MulE
MulE → MulE "*" Primary | MulE "/" Primary | Primary
Primary → "a" | FilePath
FilePath → DirPath FileName | FileName
DirPath → DirPath FileName "/" | FileName "/"
          | DirPath "./" | "./"
          | DirPath "../" | "../"
          | "/" | "~/"
FileName → "a"
```

Figure 3.17: The grammar of the language only supporting file-path names and arithmetic calculations

```
using FilePathOperators;

public class Test {
  public static void main(String[] args) {
    FilePath path = a/a/.../a;
    System.out.println(path.getAbsolutePath());
  }
}
```

Figure 3.18: The input source for the experiment of ProteaJ

from 0 to 20. According to the figure, JSGLR parser is getting slow as the input size is getting large. The parsing time increases exponentially. The worst-case time complexity of a GLR parser is $O(n^3)$ if it is implemented carefully. This fact shows that implementing an efficient scannerless GLR parser is difficult. Moreover, JSGLR could not parse when the input size is more than 20, due to a lack of memory. From this result, the naive method, in which a parser generates all possible trees and a semantic analyzer determines the most suitable one, is not fast enough to implement user-defined literals.

The compilation time by ProteaJ increases linearly with the input size. Figure 3.20 presents the compilation time by ProteaJ against the input size. The vertical axis indicates the compilation time and the horizontal axis indicates the input size. This figure presents the same data as Figure 3.19 but at a different scale. The graph is plotted with the input size from 0 to 1000. The vertical axis of Figure 3.19 is at a logarithmic scale while that of Figure 3.20 is at a linear scale.

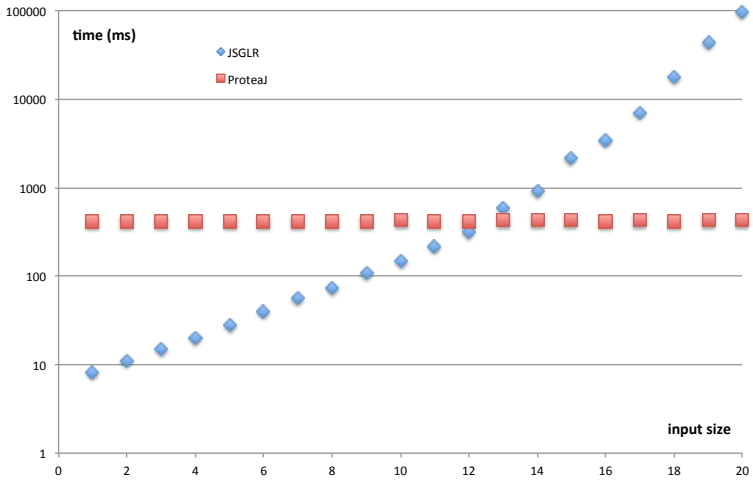

Figure 3.19: Comparison between generating all possible trees by JSGLR and our method by the ProteaJ compiler

3.5.2 User-Defined Literals and Identifiers

We have conducted an experiment for showing composability of protean operators; the parsing time does not increase as a number of operators are imported. A user-defined literal that looks like an identifier is a useful construct of an internal DSL. In the SQL DSL shown in Section 3.4.5, for example, a select-expression is written as follows:

```
select name from students where age < 20
```

In this expression, `students` is a table name in a database and `name` and `age` are column names of the table. They look like identifiers but they are user-defined literals. Similarly, a number of user-defined literals share the same syntax with normal identifiers in Java. If we use a naive parsing method, these literals introduce syntactical ambiguity and slow down parsing. In ProteaJ, these literals do not cause ambiguity and do not slow down parsing. This experiment aims to watch the change of the parsing/compilation time when changing the number of user-defined literals. We compared the compilation time by ProteaJ and the parsing time by JSGLR just like the experiment in Section 3.5.1. The experiment setting is also the same as in 3.5.1.

We measured the parsing/compilation time while changing the number of different kinds of user-defined literals from 1 to 20. The base grammar given to JSGLR is simplified Java statement syntax shown in Figure 3.21. The full version of the grammar can be obtained from our github page mentioned in Section 3.4.5. The grammar of the user-defined literal is as follows:

$Primary \rightarrow ID_i$
$ID_i \rightarrow Identifier$

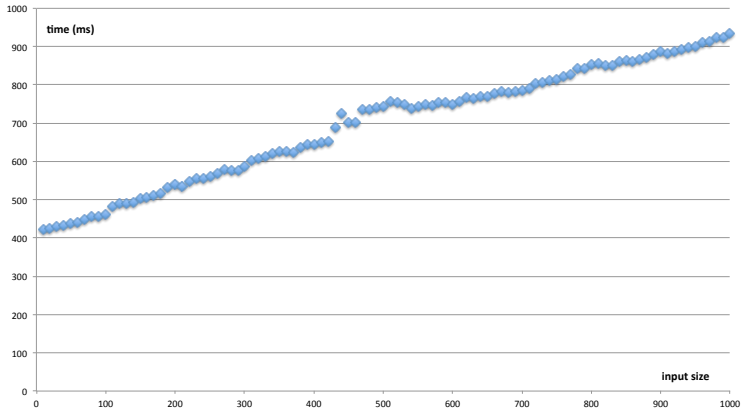

Figure 3.20: The compilation time by ProteaJ

Here, `i` indicates the kind of the user-defined literal. It is a non-negative integer number less than the number of the kinds of user-defined literals `n`. For ProteaJ, we defined user-defined literals as follows:

```
operators AmbIDsOperators {
  readas ID0 id (Identifier id) { return new ID0(id); }
  ...
  readas IDn−1 id (Identifier id) { return new IDn−1(id); }
}
```

The input of this experiment contains local variable declarations like:

```
ID0 idj = a;
```

The right-hand side of the declaration is ambiguous because `a` can be parsed as an $\texttt{ID}_0$ literal, an $\texttt{ID}_1$ literal, an $\texttt{ID}_2$ literal, and so on. The number of the possible parsing results is n. The literal of type $\texttt{ID}_0$ is finally selected since the type of $\texttt{id}_j$ is $\texttt{ID}_0$. The source code given to JSGLR contains variable declarations as shown below:

```
{
  ID0 id0 = a;
  ...
  ID0 idm−1 = a;
}
```

The source code given to ProteaJ contains a class declaration with only one method:

S → *SingleStmt*
Stmts → *Stmts Stmt* | *Stmt*
Stmt → *SingleStmt* | *LocalVarDecl*
SingleStmt → "{" *Stmts* "}" | *Expr* ";" | *IfStmt*
LocalVarDecl → *Type Identifier* "=" *Expr* ";" | *Type Identifier* ";"
Expr → *Expr1*
...
Expr9 → *Primary*
Primary → "(" *Expr* ")" | *Variable* | *IntLiteral* | *StringLiteral*
Type → *Identifier*
Variable → *Identifier*

Figure 3.21: Simplified Java statement syntax

```
using AmbIDsOperators;
class Test_m {
  public static void main (String[] args) {
    ID_0 id_0 = a;
    ...
    ID_0 id_{m-1} = a;
  }
}
```

The input length `m` is the number of local variable declarations in the source code. For different `m`, 1000, 2000, ..., 10000, we measured the average parsing/-compilation time after running ProteaJ or JSGLR ten times.

Figure 3.22 shows the result of the experiment with JSGLR. The vertical axis denotes the parsing time and the horizontal axis denotes the number of the kinds of user-defined literals `n`. Each line denotes the parsing time with a different `m`. The numbers at the right side indicate `m`. According to the figure, the parsing time increases linearly with the number of user-defined literals. Figure 3.23 shows the compilation time by the ProteaJ compiler. The vertical axis denotes the compilation time and the horizontal axis denotes the number of user-defined literals. The figure reveals that the number of different kinds of user-defined literals does not affect the compilation time.

The compilation time by ProteaJ is 3 to 10 times longer than the parsing time by JSGLR. Although the compilation time by ProteaJ does not slow down as the number of user-defined literals increases, its absolute speed is not fast. At least part of the problem is due to the fact that the current ProteaJ compiler is a prototype and not well tuned. On the other hand, JSGLR parser is fast since SGLR parsing is efficient when the grammar has only a few ambiguities. Unlike the experiment in Section 3.5.1, the ambiguities in this experiment do not exponentially explodes because there is no operator taking multiple ambiguous operands.

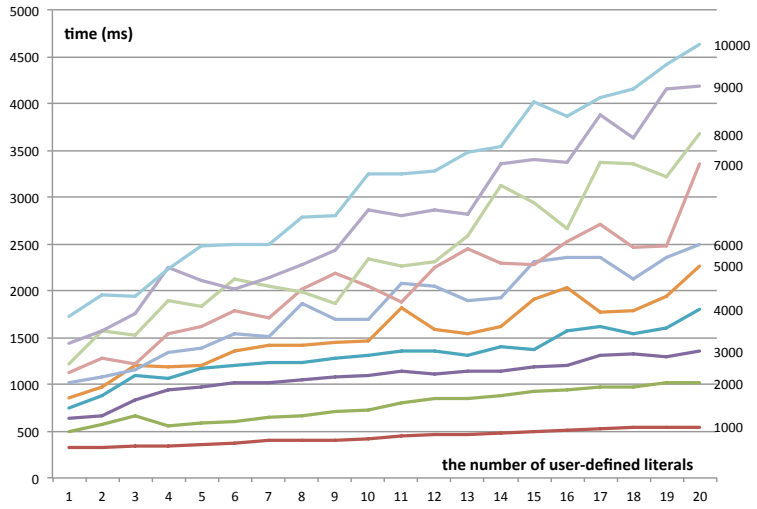

Figure 3.22: The parsing time by JSGLR of the source code containing user-defined literals

3.5.3 Parsing Java Source code

Although the proposed parsing method is flexible and runs efficiently, it is slower than the parser of a normal Java compiler. If any user-defined operators are not used, the flexibility of our method is just a performance penalty. We hence conducted an experiment for measuring the speed of parsing normal Java source code. We measured the compilation time of Apache log4j 1.2 [21], which is an open source project in Java. It consists of 213 Java source files and 21050 lines of code. We compared the compilation time by the ProteaJ compiler, `javac` of Oracle JDK, and the JastAddJ [16] compiler. The reason we chose JastAddJ is that it is a compiler developed as a research product. It is a mechanically generated compiler and its implementation is not hand-optimized. Note that ProteaJ does not support inner classes and some other constructs available in Java. Thus we modified some source files of log4j for the experiment. We substituted 1472 lines of code for 1134 lines of code including 89 declarations of inner classes. In this experiment, our compiler generated Java source files and compiled them by javac because of a bug of the backend compiler of the ProteaJ compiler. The experiment setting is the same as in Section 3.5.1 but in the experiment with JastAddJ, we used OpenJDK 1.7.0_51 since the JastAddJ compiler did not work with JDK 1.8. We measured the average of the total compilation time for ten compilations by each compiler.

Figure 3.24 shows the average compilation time of log4j by the ProteaJ compiler, javac in Oracle JDK, and the JastAddJ compiler. The vertical axis indicates the compilation time. The bar of ProteaJ is divided into two bars; the lower one expresses the translation time by the ProteaJ compiler and the upper

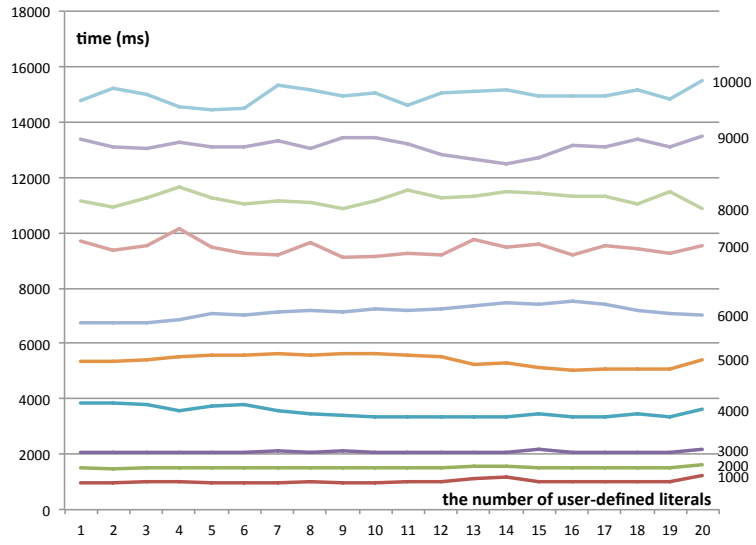

Figure 3.23: The compilation time by the ProteaJ compiler of the source code containing user-defined literals

one expresses the compilation time by javac. A vertical line on each column represents an error based on standard deviation. According to the figure, the ProteaJ compiler is six times slower than javac and four times slower than the JastAddJ compiler. The compilation time by ProteaJ is about 18 seconds. It is not fast but acceptable in practice.

3.6 Related Work

Macros

Syntactic macros are a common language facility to extend language semantics. They are based on Abstract Syntax Tree (AST) transformation. We can use them for implementing a new language construct. Lisp is the most famous language that supports syntactic macros. Syntactic macros are powerful especially in Lisp since Lisp programs are represented by simple syntax, S-expressions. We can define any kinds of special form if the syntax is an expression surrounded with parentheses. A drawback of syntactic macros is that they cannot lexically extend the syntax of the host language since they are applied after parsing a program. There are many languages supporting syntactic macros, besides Lisp. For instance, Dylan [3], MetaML [62], Template Haskell [63], Nemerle [66], and Scala [53] support syntactic macros. They have the same drawback as Lisp macros.

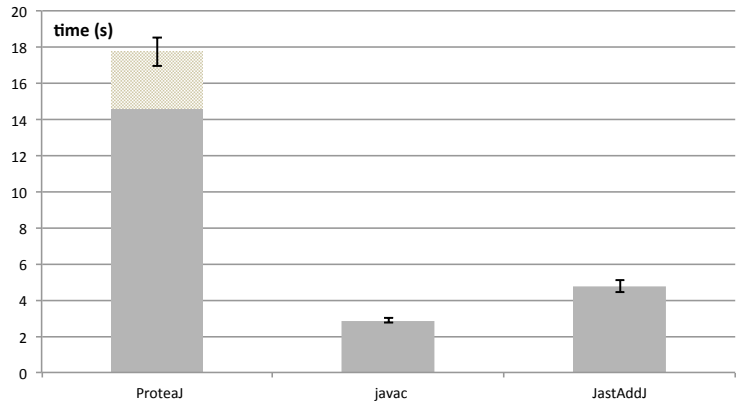

Figure 3.24: The compilation time of log4j

Common Lisp has syntactic macros and it also has a syntax extension system that is known as reader macros. Reader macros switch the scanner and the parser to user-defined ones when a special token is read. We can define a new syntax by using reader macros and we can define the semantics of it by using syntactic macros. A drawback of reader macros is lack of composability. Multiple syntax definitions in different reader macros cannot be used at the same time. User-defined scanners and parsers used in reader macros may be implemented by different programmers. Since it is difficult to merge them, the syntax defined in them would be difficult to be used together. Template Haskell [63] and Converge [73] have the same facilities.

Honu [59] provides syntactic macros with binary infix syntax. Macros with binary infix syntax improve the drawback of syntactic macros. Just like a language supporting user-defined binary infix operators, they can express various domain specific syntax. However, the drawback of syntactic macros still remains; they cannot express syntax that cannot be expressed in the host language syntax or binary infix syntax.

C/C++ provides traditional macro system, called lexical macros. They are based on text transformation. Unlike syntactic macros, they can extend syntax but they are not safe. For example, the hygiene problem is famous problem of lexical macros. Furthermore, they are not composable; macros cannot share the same syntax since each macros is identified by its first token.

Mixfix Operators with Empty Syntax

Isabelle [57] and Maude [12] are programming languages supporting mixfix operators with *empty syntax*. The empty syntax supports a nameless operator syntax like the protean operator `[_:Regex _:Regex]:Regex`. Arbitrary context-free grammars (CFGs) can be expressed by mixfix operators with empty syntax. Although mixfix operators with empty syntax have good expressiveness, they

cannot express user-defined literals. Their naive extension by using a scannerless parser is not practical due to the inefficiency of parsing as we mentioned.

External Tools

JastAdd [17] and Silver [74] are language construction systems based on attribute grammars [44]. These systems allow us to describe a language definition in declarative and modular fashion. We can extend an existing language by defining a new language extension module. Since they are systems for language developers to implement a new or extended language, they are not suitable in our case; as far as we know, there is no system where programmers can reflectively extend the underlying parser.

Metaborg [9] is a meta-programming toolkit that enables us to create syntax extensions. Since Metaborg uses SGLR parser, programmers can define both of user-defined expressions and user-defined literals in the same way. Metaborg is designed to be used for creating an extended language that has new language features. It is not designed to combine a number of language extensions that are selected by users (not language developers).

SugarJ

SugarJ [18] is a language that supports *sugar libraries*. A sugar library is a library that provides DSL syntax as syntax sugar. Programmers can use any context-free grammars for DSL syntax. A sugar library can be used with another sugar library together. However, ambiguities might arise in composed grammar when composing multiple sugar libraries. In such case, additional glue code might be necessary to integrate the libraries.

Type-Oriented Island Parsing

Type-oriented island parsing [64] is a parsing algorithm based on island parsing [68], which is a parsing algorithm for CFG, but uses type information for efficient parsing. It can efficiently parse expressions including composable user-defined operators even if the operators introduce a number of ambiguities into the grammar. It uses static type information to prune parsing paths that will make ill-typed parse trees. However, it is unclear whether or not type-oriented island parsing can be applied to a scannerless parser since it uses heuristics for parsing tokens.

Type-Specific Language

A type-specific language [55] is a language that allows programmers to define literals of a given type. The syntax of literals is arbitrary. The compiler uses type information for parsing literals. Programmers can write literals only in a body of a *generic literal*. Grammar ambiguities do not occur since the host language only provides a delimitation strategy. A literal body is parsed during

type-checking phase. It is parsed by a parser associated with the expected type of the literal. The main difference between type-specific languages and our work is where DSL syntax is available. A protean operator can be used in any expression whose expected type is statically determined before parsing the expression. Special syntax such as a general literal is not needed for escaping from the host language.

3.7 Conclusion

In this chapter, we proposed new composable user-defined operators, named *protean operators*. They can express various language extensions including user-defined literals as well as user-defined expressions. They can have any number of operator names and operands, and their order is arbitrary. Protean operators have two important features for efficient parsing: *overloading by return type* and *parsing precedence*. Overloading by return type enables a parser to resolve grammar ambiguities by using type information at parse time. The parsing precedence resolves the remaining ambiguities after resolving ambiguities by a return type. Since these features resolve all the grammar ambiguities at parse time, protean operators that express user-defined literals can be parsed in pragmatic time. We showed an efficient parsing method for protean operators based on packrat parsing supporting left recursion. This parsing method is a recursive descent parsing with backtracking and it considers type information. A drawback of protean operators is a limited kind of places where an operator is available. A protean operator is available only in an expression whose expected type is statically determined before parsing the expression.

We have developed *ProteaJ*, which is a subset language of Java but supports protean operators. ProteaJ provides a module system called *operator module* to implement and modularize user-defined operators. An operator module implements a DSL. Programmers can use the DSL by importing the module by using declaration. ProteaJ also provides two kinds of precedence rules: *operator precedence* and *parsing precedence*. Operator precedence is a convenient feature for avoiding redundant type declarations. The entire operator precedence is determined by the order of using declarations. This feature improves composability of DSLs since operator precedence of each DSL is independent. ProteaJ supports subtype relationship in Java. To handle subtype relationship, the ProteaJ compiler automatically generates operators that express subtype relationship and translates the return types of user-defined operators. Evaluation of the module system and the translation for subtype relationship is our future work. We implemented the compiler of ProteaJ in Java. It is available from our github page mentioned in section 3.4.5. We have conducted an experiment for demonstrating that ProteaJ can efficiently parse expressions including user-defined literals even though a naive parsing method cannot parse them in pragmatic time. We have also conducted experiments for measuring the parsing speed of programs written in syntax with a limited number of ambiguities. The experiments revealed that our parsing method has relatively large performance

overheads when parsing a program written in simple syntax that traditional parsing methods can efficiently deal with. The current ProteaJ compiler was 4 to 6 times slower than a normal Java compiler.

Chapter 4

Context-Sensitive Expressions

4.1 Introduction

Currently, programmers can implement libraries that look like new language constructs. These libraries can be used as if they were built-in language constructs [71]. Modern languages such as Scala and Ruby provide features that enable us to define such libraries. Programmers can implement these libraries by defining functions, methods, or operators so that they will appear to be a built-in construct. For example, we can define `unless` in Scala as follows:

```
def unless (cond: Boolean)(body: => Unit): Unit = { if (!cond) body }
```

We can use it as follows:

```
unless (a < 100) { println("a is smaller than 100") }
```

Our `unless` looks like a built-in language construct `if`. User-defined language constructs can make a program shorter and hopefully easier to understand. In particular, they are useful for improving the readability of code written in an embedded domain specific language (EDSL).

Syntactic macros [45, 29] are well-known functionality for implementing new language constructs. A syntactic macro is a translation rule from one abstract syntax tree (AST) to another. We can implement a new language construct by defining a syntactic macro that translates the language construct into a code snippet that uses predefined language constructs. Syntactic macros are widely used in Lisp since a Lisp program consists of only S-expressions. Programmers can easily modify an AST since the AST is very simple. Furthermore, a macro call looks like a built-in language construct. Syntactic macros have also been implemented in non-Lisp languages, such as Java and Haskell [4, 63].

Although syntactic macros themselves do not extend syntax, they are sometimes used as the back-end of a syntax extension system [18]. Nemerle [66]

provides a powerful syntax extension system based on syntactic macros. For example, we can define a language construct `foldfor` as follows:

```
macro FoldForMacros (id1, r, id2, list, body)
syntax ("foldfor", "(", id1, "assign", r, ";", id2, "of", list ")", body) { <[
  mutable $id1 = $r;
  foreach ($id2 in $list) { $body }
  $id1
]> }
```

Line 2 in this definition declares the syntax of the language construct. The first token `foldfor` is a unique identifier to distinguish this syntax from other user-defined syntax. The rest of the syntax consists of operators or identifiers expressed by string literals and the names of parameters. The semantics of the syntax is defined by a syntactic macro. Lines 3 to 5 define the macro expansion rule. This syntax can be used as follows:

```
def sumOfSquare = foldfor ( a assign 0 ; i of xs ) { a = a + i * i }
```

Syntactic macros are not the only approach to syntax extension systems. Another approach is to use user-defined operators. A user-defined operator is a function that has its own syntax. In Scala, for example, programmers can define an operator that has arbitrary binary infix syntax. An advantage of user-defined operators over user-defined syntactic macros is that an operator can be statically typed. The compiler can find bugs in the definition of an operator before the operator is used. Our previous work, called ProteaJ [35], is an extended Java language. It provides user-defined operators that can express the arbitrary syntax of parsing expression grammars (PEGs) [20]. Although ProteaJ can express various kinds of syntax found in many programming languages, it cannot express language constructs such as `foldfor`. This is because ProteaJ user-defined operators cannot support name binding. Name binding is the association of names with values. A user-defined operator is just a function with custom syntax and hence it does not have the ability to customizing name binding. In fact, a compiler resolves name bindings after the parsing or syntactic analysis processes are complete. While the resolution of name bindings is not part of parsing, a user-defined operator in ProteaJ is just a mechanism for customizing the parser.

To enable a user-defined operator with custom name binding, we proposes *context-sensitive expressions*, which are a variant of lambda expressions. A context-sensitive expression looks like a normal expression but it is a lambda-like expression implicitly taking parameters. These parameters are accessed through public members such as methods or operators of one parameter object instead of through parameter names. We can emulate name binding by using such public members; for example, we can emulate a variable named `''it''` by a nullary operator `''it''` and a unary operator `''it = _''`. We have developed ProteaJ2, which is a programming language based on Java but supports context-sensitive expressions. ProteaJ2 supports *turnstile types*, *DSL classes*, and *generic names* to express context-sensitive expressions for syntax extension with name binding. To efficiently parse a program using a context-sensitive

expression, the parser of ProteaJ2 adopts *eager disambiguation* using expected types.

In the rest of this chapter, we first show existing approaches for implementing user-defined language constructs with custom name binding. Then we propose *context-sensitive expressions* and present our language design for supporting them in Section 4.3. Section 4.4 explains a parsing method and the restrictions of our language for efficient parsing. In Section 4.5, we discuss the time complexity of our parsing method and show several examples to demonstrate the expressiveness of our proposal. Section 4.6 is on related work. We conclude in Section 4.7.

4.2 Motivation

4.2.1 User-Defined Operators

Powerful user-defined operators can be used for implementing user-defined language constructs with a user-friendly syntax. For example, the following code uses a user-defined language construct `"p"`:

```
p "hello, world!";
```

This code displays "hello, world!" to the user. The language construct `"p"` is defined as a unary operator as follows:

```
void "p" _ (String s) { System.out.println(s); }
```

This is a code snippet in ProteaJ, shown in the previous chapter. ProteaJ has Java-like syntax and supports powerful user-defined operators. In this chapter, as in ProteaJ, we assume that we can define an operator with an arbitrary name like `"p"`. The above code snippet is a definition of a user-defined operator. `void` at the beginning of the snippet is the return type of the operator. The following `"p" _` indicates the syntax of the operator. The double-quoted string `"p"` expresses a *name part* of the operator and the underscore `_` expresses an *operand* of the operator. A name part is a symbol for identifying the operator. An operand takes an expression with the corresponding type that is specified in the parameter list of the operator. The parameter list is a parenthesized part following the syntax definition. According to the parameter list, the corresponding type of the operand is `String`. The curly braced code is an operator body that is evaluated when the operator is called.

An advantage of user-defined operators over user-defined syntax extensions based on macros is that an operator can be statically typed. The compiler can find type errors in the definition of an operator before the operator is used. In addition, it allows for another useful feature: the compiler can resolve syntactic ambiguities by using type information. User-defined syntax extensions such as user-defined operators often conflict with another syntax extension. This is because the authors of different syntax extensions do not know about extensions created by each other. In such a case, the parser might generate multiple valid abstract syntax trees (ASTs) for single fragment of source code. We call such

```
fold-for (acc = {}; n : list) {
  if-exists (acc[n]) it = it + 1
  else acc[n] = 1
}
```

Figure 4.1: Code snippet using user-defined control flow statements `fold-for` and `if-exists`

source code ambiguous. To resolve these ambiguities, the compiler should infer the user's intention: which AST represents the user's intent. The compiler can exploit type information to infer the user's intention.

4.2.2 Name Binding

A disadvantage of user-defined operators is that it is difficult to implement language constructs involving *name binding*. Name binding is the association of data with identifiers. Language constructs, especially control-flow statements, often have name binding. For example, Figure 4.1 shows a code snippet using user-defined operators, which implement control flow statements `fold-for` and `if-exists`. This code snippet traverses a list and calculates the number of occurrences of each identical item in the list. The control flow statement `fold-for` introduces two names, `acc` and `n`, and they are available in the body of `fold-for`. `acc` expresses an accumulator and `n` expresses an element of the given list. Here, `acc` is a `Map` and `n` is a key of the `Map`. The initial value of `acc` is `{}`, which is an empty `Map`. The accumulator `acc` is updated for each element of the given list according to the body of `fold-for`. The control flow statement `if-exists` also introduces a name, `it`. `it` expresses a reference to the value of `acc[n]` if it exists. `it` is available only in the body of `if-exists`, in which `it` behaves as a mutable variable that corresponds to the given key `n` in the given `Map` `acc`. Note that `if-exists` takes three arguments rather than two: `acc` and `n` are different arguments of `if-exists`. It is because `if-exists` can modify the entry of the given `Map`, `acc`, by the assignment operator `it = _`.

If we do not need custom syntax, name binding can be expressed using a lambda expression. *Higher-order abstract syntax* (HOAS) [58] is a generalized technique of this idea. For example, the following code snippet using the language construct `let`:

```
let x = 10
x = x + 1
println(x)
```

is represented in HOAS as follows:

```
fold-for (Collections.emptyMap(), list, (acc, n) -> {
  if-exists (acc.get(), n.get(), (it) -> {
    it.set(it.get() + 1)
  }, () -> {
    acc.get().put(n.get(), 1)
  })
})
```

Figure 4.2: Code snippet using `fold-for` and `if-exists` represented by higher-order abstract syntax

```
let (10, (x) -> {
  x.set(x.get() + 1)
  println(x.get())
})
```

In the HOAS representation, a variable name is encoded into a parameter name of a lambda expression and the variable value is given by the application of the lambda expression. If we use HOAS representation, we can represent arbitrary language constructs having name binding. For example, Figure 4.2 shows the HOAS representation of a code snippet in Figure 4.1. According to line 1 of Figure 4.2, the language construct `fold-for` is expressed as a function with three arguments. The first argument is the initial value of the accumulator and the second argument is the target list. The last argument is the body of `fold-for`, which is represented as a lambda function with two arguments. The arguments of the lambda expression expresses bound names, `acc` and `n`. `if-exists` is also represented in the same manner.

Recaf [5] supports a limited form of user-defined syntax based on HOAS. In Recaf, programmers can define a new language construct with one of the supported forms of syntax such as syntax similar to `for` or `while` statement. The new language construct is transformed into its HOAS representation. The programmer only implements a method that corresponds to the name of that language construct. The following code snippet is a definition of the `let` statement in Recaf, which has the same syntax as `for` statement:

```
public <U> IExec Let (ISupply<U> v, Function<U, IExec> body) {
  return (s) -> { body.apply(v.get()).exec(null); };
}
```

The `let` statement can be used as follows:

```
let (String s : "hello, world!") {
  System.out.println(s);
}
```

Unfortunately, Recaf cannot translate every possible syntax into a HOAS representation. To transform a code snippet into a HOAS representation, the compiler has to know which code snippets introduce names and where the names

are available. In Recaf, such knowledge only comes from Java, so programmers can only define language constructs that have similar syntax and the same name binding rule as predefined language constructs in Java. The language construct `fold-for` cannot be defined in Recaf since there is no language construct in Java that has similar syntax and name binding to `fold-for`.

If we use a syntax extension system based on syntactic macros, we can translate arbitrary syntax into its HOAS representation. However, this approach is known to be dangerous. It is difficult to verify that a syntactic macro always generates correct code. When a syntactic macro causes an error after expansion, programmers cannot know whether the usage of the macro is wrong or the macro definition is wrong. Furthermore, syntactic macros make it difficult to cooperate with external tools such as integrated development environments (IDEs).

4.3 Proposal : Context-Sensitive Expressions

HOAS can represent name binding but its syntax extensibility is weak without support for syntactic macros since HOAS uses lambda expressions as a primitive for representing name binding. We present a variant of a lambda expression, which we call a *context-sensitive expression*, as another primitive for representing name binding. A context-sensitive expression takes several parameters as a lambda expression does but the individual parameters are not explicitly given. They are given through the visible members, such as methods and operators, of a few parameter objects. They are available within the expression as if they are local variables. This enables more natural syntax for name binding. We have developed ProteaJ2, which is a programming language based on Java but supports context-sensitive expressions. ProteaJ2 supports *turnstile types*, *DSL classes*, and *generic names* to express context-sensitive expressions for syntax extension with name binding. The source code of the compiler is available from our github repository[1].

4.3.1 Context-Sensitive Expressions

A *context-sensitive expression* looks like a normal expression but it implicitly takes parameters. These parameters are accessed through public members of one parameter object. For example, consider the following code snippet:

```
if-exists (acc[n]) it = it + 1
else acc[n] = 1
```

This code snippet is a part of Figure 4.1. The expression `it = it + 1` is an argument of the language construct `if-exists`. This expression can be regarded as a context-sensitive expression that implicitly takes a reference to each entry of the given `Map`. This expression can be also written with a lambda expression as follows:

```
(it) -> it.set(it.get() + 1)
```

[1] `https://github.com/csg-tokyo/proteaj2`

With a context-sensitive expression, we can write it as follows:

```
set(get() + 1)
```

As you can see this code, the declaration of the parameter `''it''` and the receiver of `set` and `get` are omitted. This is similar to the omission of `this` (or `self`) in an instance method. What is different from the `this` omission is that the implicit receiver is given by the context; here, it is given by `if-exists`. The code snippet `set(get() + 1)` does not look natural but programmers can change the syntax of the getter and setter by using user-defined operators. If programmers define the getter syntax is `''it''` and the setter syntax is `''it = _''`, the code snippet is written as follows[2]:

```
it = it + 1
```

Since a context-sensitive expression is a variant of a lambda expression, the value of a context-sensitive expression is an anonymous function. The language construct `if-exists` can be defined as following pseudo code:

```
def "if-exists" "(" _ "[" _ "]" ")" _ "else" _ (map, key, thn, els) {
  if (map.contains(key)) thn.apply(new MapEntryRef(map, key));
  else els.apply(new Lazy());
}
```

This implements a new operator. Here, `thn` is a parameter taking a context-sensitive expression. The value of `thn` is an anonymous function that takes an argument expressing an implicit receiver of the getter `''it''` and the setter `''it = _''`, here, a `MapEntryRef` object. `thn.apply(...)` denotes a call of the anonymous function `thn`.

Note that a context-sensitive expression is lazily evaluated. If an object passed to a context-sensitive expression does not have a method or an operator, the context-sensitive expression is just for lazy evaluation. The parameter `els` above is an example of such a context-sensitive expression. A `Lazy` object passed to `els` by `apply` does not have any members, and hence any operators such as `''it''` and `''it = _''` are not available within the expression that `els` refers to. The expression is just a normal expression although it is treated as a lambda expression and lazily evaluated.

If context-sensitive expressions are nested, public members of the implicit receiver of the outer expression can also be used in the inner expression. For example, if the language construct `fold-for` and `if-exists` are defined by using context-sensitive expressions, we can write the program shown in Figure 4.1. The body of `fold-for` is a context-sensitive expression in which the operators `acc` and `n` are available and the body of `if-exists` is also a context-sensitive expression as mentioned above. The operators `acc` and `n` are also available in the body of `if-exists`. If we nest `if-exists`, multiple identical operators are visible from the inner expression. In this case, the operator of the innermost expression is available there.

[2] `''it''` can be regarded as a nullary operator and `''it = _''` as a unary operator taking an assigned value

4.3.2 Turnstile Types

To give a type to context-sensitive expressions and distinguish from normal expressions, ProteaJ2 provides *turnstile types*. A turnstile type is written as `S ⊢ T`, which read as *"a type `T` under the assumption `S`"*. A turnstile type cannot be used everywhere; it is only available as a parameter type. If a turnstile type is specified as a parameter type, an argument of the parameter is a context-sensitive expression. For example, the following code defines the language construct `if-exists` by using turnstile types:

```
<K, V> void "if-exists" "(" _ "[" _ "]" ")" _ "else" _
 (Map<K, V> map, K key, MapEntryRef<K, V> |- Void thn, Lazy |- Void els)
{ ... }
```

The body is the same as one shown in the previous pseudo code. In this code, the third parameter and the fourth parameter have turnstile types. Assuming `MapEntryRef` defines getter and setter operators, `''it''` and `''it = _''`, we can write the following program using `if-exists`:

```
if-exist (acc[n]) it = it + 1
else acc[n] = 1
```

The argument passed to `thn` is the expression `it = it + 1`. Because of its type, `Void` is the type of the resulting value of the expression under the assumption that `MapEntryRef<K,V>` is the type of the implicit receiver object of `''it''` and `''it = _''`. The compiler will type-check the expression (and also `thn.apply(...)` in the body of `if-exists`) to ensure that it satisfies this assumption.

As mentioned above, a context-sensitive expression is an anonymous function. In ProteaJ2, a value of turnstile type `S ⊢ T` is implemented by a function object `Function<S, T>`. For example, `thn` is an object of type `Function< MapEntryRef<K, V>, Void>` in the body of `if-exists`. Hence, the context-sensitive expression `it = it + 1` is compiled into the following lambda expression:

```
(ref) -> ref.set(ref.get() + 1)
```

In other words, a program with context-sensitive expressions can be regarded as being translated into its HOAS representation using lambda expressions.

4.3.3 DSL Classes

A *DSL class* is a module system for user-defined operators. It is similar to a normal class in an object-oriented language like Java but it contains operators rather than methods. When a DSL class type appears on the left-hand side of a turnstile type, the operators declared in that DSL class are available in the context sensitive expression of that turnstile type. The instance operators are invoked on an instance of the DSL class passed to that expression. Figure 4.3 shows the definition of the DSL class `MapEntryRef`. Line 2 defines the getter

```
dsl MapEntryRef <K, V> {
  V "it" () { return map.get(key); }
  void "it" "=" _ (V value) { map.put(key, value); }
  MapEntryRef (Map<K, V> map, K key) { this.map = map; this.key = key; }
  private Map<K, V> map;
  private K key;
}
```

Figure 4.3: The definition of the DSL class `MapEntryRef`

operator `''it''` and line 3 defines the setter operator `''it = _''`. These operators are available in an expression of the turnstile type `MapEntryRef<K, V>` ⊢ `R`. Line 4 defines a constructor of `MapEntryRef`. Line 5 and line 6 defines instance fields of `MapEntryRef`. As a normal Java class, a DSL class is instantiated by `new` and an instance operator can access instance fields.

A DSL class can also contain static fields and operators. In ProteaJ2, a static operator of a DSL class is available anywhere, not only within a context-sensitive expression. For example, `if-exists` is a static operator. Figure 4.4 shows the definition of the DSL class `MapUtils`, which contains the operator `if-exists`. It also contains several other operators for accessing a `Map` object such as `_ [ _ ] = _`. These operators are available anywhere in a source file that imports the DSL class `MapEntryRef`. A DSL class is imported by an `import dsl` declaration. The following code is an example using the operators defined in Figure 4.4:

```
import dsl MapUtils;
...
Map<String, Color> colors = {};
colors["red"] = Color.RED;
Color c = colors["red"];    // Color.RED
```

As in Python, we can use the syntax of associative arrays for `Map` by importing `MapUtils`. In ProteaJ2, the `import dsl` declaration is written at the top of the source file (at the same position as an ordinary `import` declaration).

4.3.4 Generic Names

To recognize arbitrary names given by users, ProteaJ2 provides *generic names*, which extend generics. A generic name is a type parameter of a method or an operator. It is used for checking whether a name is identical to a name that appeared before. For example, in Figure 4.1, the language construct `fold-for` binds two names given by the user, `acc` and `n`. These names also appear in the body of `fold-for`. Generic names are used for recognizing their identity.

Figure 4.5 shows the definition of `fold-for`, which uses generic names for dealing with names given by users. The generic names are `id1` and `id2` in the type parameters of the DSL class `FoldFor` (line 1) and the static operator

```
dsl MapUtils {
  static <K, V> void "if-exists" "(" _ "[" _ "]" ")" _ "else" _
   (Map<K, V> map, K key, MapEntryRef<K, V> |- Void thn, Lazy |- Void els) {
     if (map.contains(key)) thn.apply(new MapEntryRef<K, V>(map, key));
     else els.apply(new Lazy());
   }
  static <K, V> Map<K, V> "{}" ()
   { return new HashMap<K, V>(); }
  static <K, V> V _ "[" _ "]" (Map<K, V> map, K key)
   { return map.get(key); }
  static <K, V> void _ "[" _ "]" "=" _ (Map<K, V> map, K key, V value)
   { map.put(key, value); }
}
```

Figure 4.4: The definition of the DSL class `MapUtils` that contains the operator `if-exists`

`fold-for` (line 2). `Id` is the type name of the generic names. The generic names `id1` and `id2` can appear in an operand part of operator syntax. They specify that all the occurrences of `id1` or `id2` refer to an identical expression (i.e. name). For example, line 2 and 3 in Figure 4.5 specifies that the operand `id1` of `fold-for` is identical to the operands `id1` of the following operators if they occur in the context-sensitive expression `f`.

```
R id1 () { return acc; }
void id1 "=" _ (R r) { this.acc = r; }
```

This is specified by the assumption part of `f`'s type, `FoldFor<T,R,id1,id2>`. The expression `f` can invoke the operators declared in `FoldFor<...>` where `id1` is identical to the operand `id1` of `fold-for`. Thus, the invocation of `fold-for` in Figure 4.1 is valid. Here, `id1` is bound to `acc` and `id2` is bound to `n`. `FoldFor<...>` declares the getter operators `acc` and `n` and the setter operator `acc = _`. They are available in the expressions at line 2 and 3 in Figure 4.1.

The generic names `id1` and `id2` refer to an expression of type `Id`. The compiler recognizes that two generic names are identical if the abstract syntax trees of their expressions are equivalent. We chose this design since we express a name by the composition of several user-defined operators. Note that the type `Id` is not a primitive or meta type. It is a type that users can define for expressing an arbitrary sequence of alphabet letters as an expression. See the following operators for `Id` values:

```
literal Id _ _ (Letter letter, Id rest)
 { return new Id(letter.toString() + rest.toString()); }
literal Id _ (Letter letter) { return new Id(letter.toString()); }
literal Letter "a" () { return Letter.a; }
...
literal Letter "z" () { return Letter.z; }
```

```
dsl FoldFor <T, R, id1: Id, id2: Id> {
  static <T, R, id1: Id, id2: Id> R "fold-for" "(" id1 "=" _ ";" id2 ":" _ ")" _
   (R ini, List<T> list, FoldFor<T, R, id1, id2> |- Void f) {
    FoldFor<T, R, id1, id2> env = new FoldFor<T, R, id1, id2>(ini);
    for (T t : list) {
      env.elem = t;
      f.apply(env);
    }
    return env.acc;
  }
  R id1 () { return acc; }
  T id2 () { return elem; }
  void id1 "=" _ (R r) { this.acc = r; }
  FoldFor (R ini) { this.acc = ini; }
  private R acc;
  private T elem;
}
```

Figure 4.5: The definition of `fold-for` in ProteaJ2

`literal` is a modifier indicating the operator is for user-defined literals in ProteaJ2. The syntax of a literal operator does not recognize whitespace characters as a separator between name parts and operands. An operand of a literal operator is an expression that consists of only literal operators. User-defined literals defined by literal operators can be used as a normal expression. Therefore, if we import the operators above, the following code is valid:

```
Id id = acc;
```

The right-hand side of `=` is the composition of operators. It is parsed like `toId(a(), toId(c(), toId(c())))`. As in our previous work ProteaJ [35], `a` and `c` are recognized as operators since the expected type of the expression is `Id`. If a generic name refers to an expression like `acc`, the expression is not evaluated at runtime or at compile time. It is compiled out to be a unique identifier.

4.3.5 Operator Priorities

ProteaJ2 supports *operator priorities* for helping users to define and use operators. Figure 4.6 shows a definition of the DSL class `MapUtils` with operator priorities. Line 2 in the figure is a declaration of priorities. This declaration declares three priority names `p1`, `p2`, and `p3`, and the order of each. Note that `p1` is a short-hand name; its qualified name is `MapUtils.p1`. We can use a qualified priority name of another DSL class in the declaration as follows:

```
priorities p1, p2, p3 { p1 < p2 < PredefOperators.add < p3 }
```

```
dsl MapUtils {
  priorities p1, p2, p3 { p1 < p2 < p3 }
  static <K, V> void [p1] "if-exists" "(" _ "[" _ "]" ")" _ "else" _[p1]
   (Map<K, V> map, K key, MapEntryRef<K, V> |- Void thn, Lazy |- Void els) {
     if (map.contains(key)) thn.apply(new MapEntryRef<K, V>(map, key));
     else els.apply(new Lazy());
   }
  static <K, V> Map<K, V> [p3] "{}" ()
   { return new HashMap<K, V>(); }
  static <K, V> V [p2] _ "[" _ "]" (Map<K, V> map, K key)
   { return map.get(key); }
  static <K, V> void [p2] _ "[" _ "]" "=" _ (Map<K, V> map, K key, V value)
   { map.put(key, value); }
}
```

Figure 4.6: The definition of the DSL class with operator priorities

The order of priority names `p1 < p2` indicates that an operator with the priority `p1` can take as an operand an expression of an operator with the priority `p2`. Conversely, an operator with the priority `p2` cannot take an expression of an operator with the priority `p1`. A priority name can be given to an operator and its operands by attaching the name enclosed by brackets. For example, the operator `if-exists` has the priority `p1` and the right-most operand of the operator has the priority `p1` in the figure. This specifies that `if-exists` is right-associative and `if-exists` can be used as an operand that has a priority lower than or equal to `p1`. An operand without a priority name takes an expression with a higher priority than its own operator priority. Therefore, the first operand of `if-exists` can take an expression of the operator `''it = _''` but cannot take an expression of `if-exists`.

Operator priorities declared in a DSL class are a partial order. The total order is determined at each call site of operators. When a user imports multiple DSL classes, the compiler sorts all the priority names in the DSL classes in the declared orders. If priorities are cyclic, the compiler fails to compile a program and reports invalid operator priorities. To control operator priorities between independent DSL classes, a user can specify the order of priority names in the DSL classes at the import declaration as follows:

```
import dsl MapUtils { MapUtils.p2 < PredefOperators.add < MapUtils.p3 }
```

4.4 Parsing

The compiler of ProteaJ2 adopts *eager disambiguation* for fast parsing. The syntax of user-defined operators in ProteaJ2 might be ambiguous if we do not consider types and operator priorities. When syntax is ambiguous, a parser may

generate a large, sometimes exponentially large, number of potentially valid ASTs for the same source code. All these ASTs except the semantically correct one are filtered out after parsing by applying semantic rules; in ProteaJ2, they are filtered out by checking types and operator priorities. If multiple ASTs are semantically correct, the language itself is ambiguous. In general, the generation of a large number of potentially valid ASTs slows down the parsing speed. Eager disambiguation is a technique to filter out semantically invalid ASTs while a parser is running. It enables faster parsing since ambiguities are resolved before the number of ASTs exponentially increases.

For eager disambiguation, ProteaJ2 uses *expected* types and operator priorities. The ProteaJ2 compiler parses a source program with a top-down parsing algorithm. Before parsing an expression, it prunes away syntax rules not in effect in the context of that expression so that the number of generated ASTs will be reduced. If the expression expects a turnstile type `D ⊢ T`, the compiler tries to parse it as an expression with the expected type `T` where the syntax of instance operators of `D` are in effect. Such operators are also in effect in sub-expressions of the expression. Static operators are effective everywhere if the DSL class that declares them has been imported. When the compiler tries to parse an expression with the expected type `T` (which is not a turnstile type), it examines only the syntax of operators that are in effect and that return the expected type `T`. For example, when the compiler parses an expression on the right-hand side of the assignment operator `=`, it examines the syntax of the operators that return a subtype of the left-hand side of the assignment. A bottom-up parsing algorithm is not suitable for such eager disambiguation. It is difficult to determine what type is expected at the expression.

Due to this approach, user-defined operators are available in an expression whose expected type is statically determined during top-down parsing. For example, user-defined operators are available in the following locations: the argument of a method call, the right-hand side of an assignment, the condition expression of an `if` statement, an operand of another user-defined operator, and so forth. On the other hand, they are not available at the receiver of a method call or the left-hand side of an assignment. Details of this restriction has been mentioned in our previous manuscript [35].

After parsing each (sub-)expression, if ambiguities still remain, they are resolved with heuristics; the compiler selects the AST that consumes the maximum number of characters in source code. Note that the compiler completely resolves ambiguities at every sub-expression. In other words, the compiler selects the partially optimal AST. If users define and use a dangling-`else` like construct, for example, the `else` belongs to the innermost `if`. This is because the compiler selects the longest match choice, `if-else`, for the sub-expression of outer `if`. This approach is similar to parsing expression grammar (PEG) [20] and packrat parsers [19]. PEG adopts the heuristics in which if there are multiple choices, the parser selects the left-most choice. The ProteaJ2 parser is based on packrat parsing supporting left recursion [81].

4.5 Evaluation

4.5.1 Time Complexity

The time complexity of our parsing algorithm is $\mathcal{O}(N * L)$ if the grammar is not left-recursive, where N is the length of an input and L is the number of *languages* that can possibly be used in parsing. In ProteaJ2, programmers can introduce local syntax within an expression by using a context-sensitive expression. It can be regarded as changing a language in the local scope. The compiler should use a different parser for each language, so the time complexity of our parsing algorithm is in proportion to the number of languages L. The base complexity of our parser is derived from the complexity of packrat parsing [19]. The time complexity of packrat parsing is $\mathcal{O}(N)$ where N is the length of the input.

The number of languages L can be approximated to a constant in normal cases. In some cases, however, L could increase faster than linear in the length of an input N. Figure 4.7 shows a definition of operators that might increase compilation time faster than linear time. The definitions of these operators might look ambiguous. They have the same return type and the same syntax except their postfix name. Each of them has a parameter with the turnstile type like `Dx ⊢ String`. Such a type indicates that its argument is an expression of type `String` and the expression is written in a language including instance operators of `Dx`. If the operators in the figure are nested as follows, the number of languages exponentially increases:

```
begin begin begin ... begin "hello" endP endP endP ... endP
```

When the compiler reads the first `begin`, it cannot yet determine which of the operators in Figure 4.7 is used. In the worst case, the compiler should use a number of languages equal to P to parse the argument of the outermost `begin-end`. Similarly, the compiler also cannot determine which of `begin-end` operators is used when it reads the next `begin`. Hence, for each P languages for the first `begin`, it should examine P languages to parse the expression following the second `begin`. The number of languages L is P^M in the worst case where M is the number of `begin` operators. Since M is in proportion to the length of the input, the time complexity of our parsing algorithm might be an exponent of N.

The cause of this problem is that the parser cannot determine which operator is used before it parses deeply nested operands. This case would rarely occur since the authors of the operators would likely design the syntax so that the operator could be uniquely determined after just reading the operator prefix such as `begin`.

We have conducted two micro benchmarks to examine the performance of our compiler. The first benchmark was an experiment to show that the compilation time complexity could be square or cubic time in the number of operators in unfavourable cases. The second benchmark shows that the number of operators has little impact on compilation time in normal cases. The machine used for the benchmarks had a 2.6 GHz Core i5 processor and 16 GB DDR3 memory. The

```
String "begin" _ "end1" (D1 |- String f) {...}
String "begin" _ "end2" (D2 |- String f) {...}
...
String "begin" _ "endP" (DP |- String f) {...}
```

Figure 4.7: Operators that could cause compilation time to increase faster than linear time

operating system of the machine was Mac OS X 10.11.5.

The first benchmark uses the operators shown in Figure 4.7. We measured the compilation time of programs including each of the following lines while changing the number of the declared `begin-endX` operators P:

```
1. begin "hello, world!" endP
2. begin begin "hello, world!" endP endP
3. begin begin begin "hello, world!" endP endP endP
```

The rest of the input program includes the import declaration of the DSL class, a class declaration, and a main method. According to the discussion above, the number of the languages L of code 1 is in proportion to P, that of code 2 is in proportion to P^2, and that of code 3 is in proportion to P^3. If this is true, the parsing time of code 2 increases faster than code 1 and the parsing time of code 3 increases faster than code 2 as the number of operators P increases. Figure 4.8 (1) shows the result of the benchmark. The vertical axis represents the compilation time and the horizontal axis represents the number of `begin-endX` operators P. According to this figure, the parsing time of code 2 increases as the number of `begin-endX` operators P does and it increases faster than the parsing time of code 1. The parsing time of code 3 increases faster than code 2.

The second benchmark ran different operators, whose syntax rule can be uniquely determined after just reading the prefix. Their definitions are as follows:

```
String "begin1" _ "end1" (D1 |- String f) {...}
String "begin2" _ "end2" (D2 |- String f) {...}
...
String "beginP" _ "endP" (DP |- String f) {...}
```

The input of this benchmark includes one of the following lines. Note that the prefix is not `begin` but `beginP`.

```
1. beginP "hello, world!" endP
2. beginP beginP "hello, world!" endP endP
3. beginP beginP beginP
                  "hello, world!" endP endP endP
```

Since the operators can be determined by their prefix, the number of the languages L is a constant. Hence the parsing time of codes 1, 2, and 3 do not show notable differences even if the number of `beginX-endX` operators P increases.

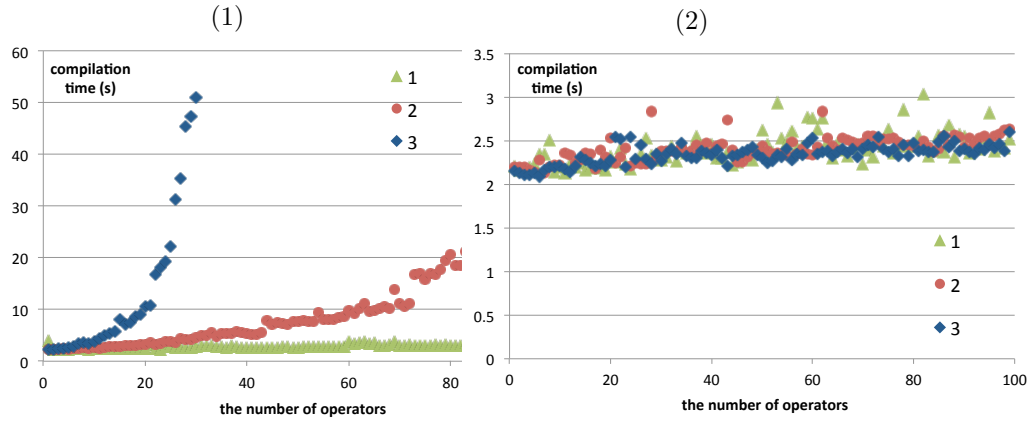

Figure 4.8: The result of the micro benchmarks

Figure 4.8 (2) shows the result of this benchmark. The vertical axis represents the compilation time, and the horizontal axis represents the number of `beginX-endX` operators P. According to this figure, the number of `beginX-endX` operators P has little impact on the compilation time.

4.5.2 Case Study

In the rest of this section, we show several use cases of our proposal. Although ProteaJ2 does not currently support anonymous classes, we use anonymous classes for simplifying a program shown in this section.

Lambda Expressions

We can implement lambda expressions in ProteaJ2. Lambda expressions are a common feature of modern programming languages and they have also been introduced in Java. They create an anonymous function as a first-class object. Figure 4.9 shows the implementation of lambda expressions. The operator `"{" var "->" _ "}"` takes a parameter name as a generic name `var`. The operator `var` (declared at line 7) is available as the operand of the operator `"{" var "->" _ "}"` since the operand type is the turnstile type `Lambda<A, var>` ⊢ `B`. Here, the operator `"{" var "->" _ "}"` and the operator `var` refer to the same generic name. Therefore, this definition correctly expresses name binding of a lambda expression. The value bound to the name is given when the `apply` method of the returned object (defined in line 4) is called. When we import this definition, we can write the following code snippet:

```
Function<String, String> f = { s -> s + "!" };
```

We can also implement a variant of lambda expressions in ProteaJ2; although a typical lambda expression consists of a single function, we can implement a lambda-like expression that consists of multiple functions. Figure 4.10 shows

```
dsl Lambda <A, var: Id> {
  static <A, B, var: Id> Function<A, B> "{" var "->" _ "}" (Lambda<A, var> |- B body) {
    return new Function<A, B> {
      B apply (A a) { return body.apply(new Lambda<A, var>(a)); }
    };
  }
  A var () { return value; }
  Lambda (A a) { value = a; }
  private A value;
}
```

Figure 4.9: The implementation of lambda expressions

a DSL class that implements such syntax for `KeyListener`. It can be used as follows:

```
Component c = ...;
c listens key {
  pressed  -> System.out.println(key name + " pressed!")
  released -> System.out.println(key name + " released!")
  typed    -> key_typed(key)
};
```

This code registers a `KeyListener` that prints a key name when the key is pressed or released and also invokes the `key_typed` method when a key is typed. On the right-hand side of each `->`, `key` acts as a variable of the type `KeyEvent`. Note that `key name` acts as a variable of the type `String`. We can provide multiple accessors to obtain the same underlying value as values of different types.

The Loan Pattern

The loan pattern is a design pattern to avoid explicitly releasing resources. It is useful for expressing I/O processing. Several languages support the loan pattern as a language construct, for example, `using` in C# and `try-with` in Java. The following code uses `try-with` in Java:

```
try (BufferedReader reader = new BufferedReader(new FileReader(file))) {
  System.out.println(reader.readLine());
}
```

It is guaranteed that `reader` is closed at the end of the `try` statement.

In ProteaJ2, we can implement this as if it is a built-in language construct. Figure 4.11 shows the definition of the DSL class `TryWith`, which implements the language construct `try-with`. Similar to a lambda expression, the operator `try` declares the name `id` and this name is available in the body of `try`. The name `id` is bound to the given resource from the first operand of `try`. It is

```
dsl KeyListenerDSL {
  static void
   _ "listens" "key" "{"
     "pressed"  "->" _
     "released" "->" _
     "typed"    "->" _
   "}"
   (Component c, KeyListenerDSL |- Void f1,
    KeyListenerDSL |- Void f2, KeyListenerDSL |- Void f3) {
      c.addKeyListener(new KeyListener {
        void keyPressed (KeyEvent e) { f1.apply(new Key(e)); }
        void keyReleased (KeyEvent e) {...}
        void keyTyped (KeyEvent e) {...}
      });
  }
  KeyListenerDSL (KeyEvent e) { keyEvent = e; }
  KeyEvent "key" () { return keyEvent; }
  String "key" "name" () { return KeyEvent.getKeyText(keyEvent.getKeyCode()); }
  private KeyEvent keyEvent;
}
```

Figure 4.10: The definition of the DSL class that implements syntax like lambda expressions for `KeyListener`

guaranteed to release the resource at the end of the `try` since the `close` method is called in the `finally` clause from line 7 to line 9.

Hiding Resources

In ProteaJ2, we can develop a much safer DSL than `try-with` for managing resources. With the loan pattern, users directly access a resource object, which is dangerous since users can assign the resource object to a variable in an external scope. In ProteaJ2, we can encapsulate the resource object as a field of a DSL class and provide instance operators only within a context-sensitive expression. Figure 4.12 shows the definition of such the DSL class that encapsulates a `BufferedReader` object[3]. We can write the following code snippet if we import that DSL class:

```
open "file.txt" {
  while (has next) { System.out.println(read line); }
};
```

The users cannot directly access a `BufferedReader` object but they can use operators `has next` or `read line` in the body of the operator `open`. Here, `while`

[3]Technically, this code does not work in the current version of ProteaJ2 since we implement a turnstile type by using `java.util.function.Function`. ProteaJ2 does not allow checked exceptions, so the value of the turnstile type `FileRead` $\vdash$ `Void` cannot throw checked exceptions.

```
dsl TryWith <R extends AutoCloseable> {
  static <R extends AutoCloseable, id: Id> void "try" "(" R id "=" _ ")" "{" _ "}"
   (Lazy |- R resource, TryWith<R> |- Void body) throws Exception {
    R r = resource.apply(new Lazy());
    try { body.apply(new TryWith<R>(r)); }
    finally { if (r != null) r.close(); }
  }
  R id () { return resource; }
  TryWith (R r) { resource = r; }
  private R resource;
}
```

Figure 4.11: The definition of the DSL class for implementing `try-with`

statement appears in the body of the operator `open`. In ProteaJ2, a turnstile type like `D` ⊢ `Void` is a special type that can take a block of statements as its argument. Since the body of the operator `open` has the turnstile type `FileRead` ⊢ `Void`, it can take a block that contains `while` statement.

Dynamic Scoping

ProteaJ2 provides a `requires` clause for implementing a method or an operator that is only available within an operand of a user-defined operator. A method or an operator with the `requires` clause becomes available if all the specified types in the `required` clause are assumptions (i.e., within a context-sensitive expression of a turnstile type containing the specified DSL type as the assumption). The following code uses a method with a `requires` clause:

```
List<String> allLines = open "file.txt" { getLines() };
List<String> getLines() requires FileRead {
  List<String> lines = new ArrayList<String>();
  while (true) {
    String line = read line;
    if (line == null) return lines; else lines.add(line);
}}
```

The `getLines` method is available only within the second operand of `open`. In the method body of `getLines`, the operator `read line` is available since `FileRead` is the assumption there. We can regard a `required` clause as an emulation of a dynamic scope since `getLines` can access the operators available in the caller's scope.

Pattern Matching

In ProteaJ2, programmers can implement expressions involving more complex name binding. For example, they can implement a simple pattern matching

```
dsl FileRead {
  static void
   "open" _  _ (String fileName, FileRead |- Void body)
   throws IOException
  {
    BufferedReader reader = new BufferedReader(new FileReader(fileName));
    try { body.apply(new FileRead(reader)); }
    finally { if (reader != null) reader.close(); }
  }
  String "read" "line" () throws IOException {
    String res = nextLine;
    this.nextLine = reader.readLine();
    return res;
  }
  boolean "has next" () { return nextLine != null; }
  FileRead (BufferedReader reader) throws IOException {
    this.reader = reader;
    this.nextLine = reader.readLine();
  }
  private BufferedReader reader;
  private String nextLine;
}
```

Figure 4.12: The definition of the DSL class that encapsulates `BufferedReader`

mechanism, which is a common language construct in functional programming languages such as Haskell and Scala. Figure 4.13 shows part of its definition. The complete definition is available from our github repository.[4] It can be used as follows:

```
String s = "hello, world" match {
  | "hello" + xs => "goodbye" + xs
  | Empty         => "empty string"
  | otherwise     => otherwise
};
```

This code displays `''goodbye, world''` to the user. In this code, `xs` and `otherwise` act as variables. The names of the variables are bound by the operator defined at line 16 in Figure 4.13. Note that a variable name is restricted here to be an identifier that starts with a lower case letter. This restriction is borrowed from the pattern matching mechanism in Scala.

[4] `https://github.com/csg-tokyo/proteaj2`

4.6 Related Work

SugarJ [18] is a programming language that supports user-defined syntax extensions based on macros. Programmers can define any context-free syntax in SDF [30] and any AST transformation rules in Stratego [77, 10]. The defined syntax and AST transformation rules are modularized as a *sugar library*. Users can import the sugar library and use the defined syntax. The users can also use multiple sugar libraries in the same source file. Unfortunately, the compiler does not have the capability to handle ambiguous syntax. This limitation breaks the composability of libraries, as several libraries cannot be used together. Several languages such as Nemerle [66] also support user-defined syntax extensions based on macros but their capabilities are more limited. In the case of Nemerle, the users can only define the syntax where the first token is an identifier. Moreover, terminal symbols must be selected from a specific set of identifiers and operators.

Type-specific languages (TSLs) [55] and typed syntax macros (TSMs) [54] are a language feature that supports composable user-defined syntax extensions. Here, composable means that users can use several syntax extensions in the same source file. When the users use the multiple syntax extensions together, syntactic ambiguities might occur. The compiler resolves such syntactic ambiguities by the expected type information. TSLs are similar to our previous work, ProteaJ, because they can give custom syntax to a function. TSMs are similar to SugarJ since they are based on syntactic macros, but they are more limited since each macro is statically typed. TSLs and TSMs can implement various syntax extensions and the extensions are safe to compose, however, they do not have the capability to define custom name binding. Since TSLs are just a function with custom syntax, they cannot extend name binding rules. TSMs are focused on the composability of syntactic macros. Hence they only handle hygienic macros [45], which are syntactic macros that do not affect name binding. Metamorphic syntax macros [6] are also syntactic macros that guarantee type safety. They do not affect name binding for the same reason as TSMs. TSLs and TSMs partially adopt eager disambiguation to parse a program in acceptable time. They provide several literal forms for using user-defined syntax. The compiler initially skips parsing such literal forms until after type checking the surrounding code. When parsing the literal, only syntax extensions that return the expected type in that position are considered.

Mixfix operators [13] are another kind of user-defined operator. They were adopted in several languages such as Coq [69] and Isabelle [57]. Unlike ProteaJ2, to reduce syntactic ambiguities, mixfix operators only allow users to use particular forms of syntax: prefix, postfix, infix, or outfix syntax. Several languages also allow programs to define operators without a name part. Since mixfix operators are user-defined operators, it is difficult to use them to implement language constructs involving name binding. Generalized method names [31] extend multi-part method names, which are found in several languages such as Smalltalk, so that they take regular expressions instead of a fixed name. They can express various syntax similar to user-defined operators. They also have

the same problem with name binding as user-defined operators since they just extend the syntax of method calls.

Implicit parameters in Scala [53] are a similar idea to our proposal. An implicit parameter is a method parameter whose argument is given implicitly if the argument is not explicitly specified at the call site of the method. The actual argument is found among implicit values visible in the current scope. The difference is that the arguments of our context-sensitive expression are given from the outer method or operator. Implicit parameters enable a context-sensitive expression to express syntax available only within a local scope. We propose a turnstile type for expressing such an expression. The Glasgow Haskell Compiler (GHC) has an extension named ImplicitParams based on [49]. It enables programmers to use variables with a dynamic scope in a type safe fashion. ImplicitParams is similar to our `requires` clause mentioned in section 4.5.2.

Kotlin provides "function literals with receiver", which enables calling a function literal with a specified receiver object. It is used for writing type-safe Groovy-style builders in Kotlin. Function literals with receiver provides similar capability as a context-sensitive expression. A major difference from our proposal is Kotlin does not aim to implement name binding, so it does not have the functionality to support syntax extension with name binding such as DSL classes and generic names. `instance_eval` in Ruby is also related to context-sensitive expressions. `instance_eval` evaluates the given block as if it is an instance method of the given receiver. `instance_eval` can be regarded as a context-sensitive expression implemented in a dynamically typed language. Since name binding is resolved at runtime, `instance_eval` is not safe.

Type-oriented island parsing [64] is a parsing technique to efficiently parse a source program. It is an extension of island parsing [68], which is an efficient bottom-up parsing algorithm using heuristics. The type-oriented island parsing performs eager disambiguation for a completed child AST by using type information. The paper also shows that they have implemented a syntax extension system with name binding (called variable binders in that paper). To express name binding, they adopted labeled symbols [37] and a scoping construct [11]. Labeled symbols are similar to generic names in our system, and a scoping construct is similar to the left-hand type of a turnstile type. However, these constructs express a variable naively and the variable is not encapsulated. They cannot change the syntax of names or publish a different interface for accessing a variable. In other words, a DSL class is a module system that reorganizes and generalizes labeled symbols and a scoping construct to integrate them with an object-oriented language. Furthermore, as far as we know, it is not clear whether or not the type-oriented island parsing can be applied to scannerless parsers since type-oriented island parsing uses heuristics for parsing tokens. If it cannot be applied to scannerless parsers, programmers cannot extend literal syntax.

4.7 Conclusion

This chapter proposes a variant of a lambda expression, named a *context-sensitive expression*, for implementing user-defined language constructs with custom name binding. A context-sensitive expression looks like a normal expression, but it implicitly takes parameters. These parameters are accessed through public members such as methods and operators of a parameter object instead of through parameter names. We have developed ProteaJ2, which is a programming language based on Java but supports context-sensitive expressions. ProteaJ2 supports *turnstile types*, *DSL classes*, and *generic names* to express context-sensitive expressions for syntax extension with name binding. The parser of ProteaJ2 adopts *eager disambiguation* using expected types to efficiently parse a program. We showed several micro benchmarks to illustrate that compilation time is acceptable. We also showed several case studies to demonstrate our system can implement various language constructs with custom name binding.

```
dsl MatchDSL {
  static <T, R> R _ "match" "{" _+ "}" ( T t, Case<T, R>... cases ) {
    for (Case<T, R> c : cases) {
      Optional<R> result = c.apply(t);
      if (result.isPresent()) return result.get();
    }
    return null;
  }
  static <T, R> Case<T, R> "|" _ "=>" _ ( Matcher0<T> m, Lazy |- R f ) {
    return new Case<T, R> ({ t -> m.ifMatch(t, f) });
  }
  static <T, A, name: Id_Lower, R> Case<T, R>
   "|" _ "=>" _ ( Matcher1<T, A, name> m, Var<A, name> |- R f ) {
    return new Case<T, R> ({ t -> m.ifMatch(t, f) });
  }
  static <T> Matcher0<T> "_" () { return new Matcher0 <T> ({ t => Boolean.TRUE }); }
  static <A, name: Id_Lower> Matcher1<A, A, name> name () {
    return new Matcher1 <A, A, name> ({ a -> Optional.<A>of(a) });
  }
  static Id_Lower _ _* (LowerCase lc, Letter... letters) {
    return new Id_Lower(lc, letters);
  }
}
dsl StringMatchers {
  static Matcher0<String> "Empty" () {
    return new Matcher0 <String> ({ t -> Boolean.valueOf(t.isEmpty()) });
  }
  static <A, name: Id_Lower> Matcher1<String, A, name>
   _ "+" _ ( String prefix, Matcher1<String, A, name> m ) {
    return new StringPrefixMatcher <A, name> (prefix, m);
  }
}
dsl Var <A, name: Id_Lower> {
  Var (A a) { this.a = a; }
  A name () { return a; }
  private A a;
}
```

Figure 4.13: Part of the definition of pattern matching

Chapter 5

User-Defined Declaration Statements

In the previous chapter, we proposed context-sensitive expressions and turnstile types, which enable users to declare language constructs involving name binding such as `let` expressions. However, users still cannot declare a "declaration", which declares a name and makes it available after the declaration. To be precise, the users can implement such declarations but the compiler cannot statically check the implementation and its usage. Most programming languages adopt declarations as language features; for example, Java has local variable declarations and class declarations. In this chapter, we propose a language extension mechanism for enabling a limited kind of declarations.

5.1 Declarations and Scopes

In the rest of this chapter, we use QML [51] as an example of DSLs with declarations. Figure 5.1 is a program in QML. This program declares two rectangles, `rect1` and `rect2`. `rect1` has `size` property and the value is `100`. The `width` of `rect1` is the same value as the `size` property and the `height` of `rect1` is the same as the `width` of `rect1`. The `width` of `rect2` is the half of the `size` of `rect1`. The `height` of `rect2` is the same as the `width` of `rect2`.

Line 3 of this program declares `size`. Note that the users can access to `size` in two ways; `size` and `rect1.size`. The expression `size` can be used only within the declaration of `rect1`. Therefore, its scope extends from the declaration (Line 3) to the end of the Rectangle block (Line 6). The expression `rect1.size` can be used anywhere after the declaration, so its scope extends from the declaration (Line 3) to the end of the QML program (Line 11). `width` and `height` are the default properties of `Rectangle`. Similar to `size`, they can be accessed in two ways; `width` and `rect1.width`. The expression `width` is available within the `Rectangle` declaration (Line 1 - 6) and it returns the value of `width` of the declared rectangle. The expression `rect1.width` can be used

```
Rectangle {
  id : rect1
  property int size : 100
  width : size
  height : rect1.width
}
Rectangle {
  id : rect2
  width : rect1.size / 2
  height : width
}
```

Figure 5.1: A QML program

anywhere, so its scope extends from the declaration of `rect1` (Line 2) to the end of the QML program (Line 11).

It is difficult to implement such the DSL as an embedded DSL. There are two reasons for this difficulty. First, the declared name is available not only in the arguments of the declaration. The `let` expression is easy to implement since its scope is only in the argument of that expression. However, the scopes of `size` and `rect1.size` in Figure 5.1 are after the declaration. Hence such a declaration cannot be implemented by only using local analysis; for example, syntactic macros cannot easily implement such a declaration since they translate an AST into another AST without considering contexts. Second, the scopes of declared names might overlap with each other. For example, the scope of `rect1.size` is from Line 3 to Line 11 but the scope of `width` is from Line 1 to Line 6. Such scope rules are difficult to implement by the name binding of the host language since the existing language constructs can only deal with nested structures. For example, higher-order abstract syntax (HOAS) cannot emulate those scope rules since HOAS uses lambda expressions for expressing name binding. Two lambda expressions can implement nested code regions but not overlapping ones.

5.2 Context Activation within a User-Defined Scope

Our basic idea is evaluating the delimited continuation of a user-defined declaration as a context-sensitive expression. The root cause of the difficulty about user-defined declarations is that the range of the scope of the user-defined declarations is not a single expression. Delimited continuations can solve this problem – they enable programmers to treat the rest of the calculation as an expression. If the programmers evaluate the expression as a context-sensitive expression with the type `S ⊢ T`, they can use instance operators of `S` within the expres-

sion, that is, they can use the declared names in the rest of the calculation after the declaration. Our idea is based on delimited continuations but it does not need to implement delimited continuations. It is just an application of delimited continuations. We have chosen a design similar to checked exceptions in Java for implementing our idea into ProteaJ2. We introduce three language features, `activate` statements, `activates` clauses, and `scope for` clauses. The `activate` statements correspond to the `throw` statements, the `activates` clauses correspond to the `throws` clauses, and the `scope for` clauses correspond to the `try` statements. In the rest of this section, we describe these language features and how to implement user-defined declarations by them.

`activate` statements take a DSL object and make its visible members available without a receiver in the rest of the program. If the DSL object expresses the syntax of a variable `x` for example, programmers can use the syntax of the variable `x` after the `activate` statement. Therefore, the `activate` statement works as a variable declaration. Similarly, the operators using `activate` statements within their body also work as declarations. The visible members of the DSL object taken by the `activate` statement in the method body are also available without a receiver after the call of the operators. Hence, we can implement an operator that emulates a variable declaration; if the operator activates an DSL object that contains a getter operator and a setter operator for `x`, programmers can write the code in which variable `x` is available after calling the operator. In this case, the operator must have an `activates` clause. The `activates` clause takes the type of the DSL object and it indicates that the operator activates a DSL object of the given type. This is statically checked. To express the range where the visible members of a DSL object are available, programmers can use `scope for` clauses. A `scope for` clause is attached to an operand of an operator (or a parameter of a method) and it takes a type of DSL objects. If a DSL object of the given type is activated in the operand, the visible members of the DSL object are available after the activation but they are not available outside the operand. The `scope for` clauses allow programmers to use different scope rules for each type of DSL objects.

5.2.1 Simplified QML

To simplify out explanation, we slightly modify the QML syntax. Figure 5.2 shows a program in simplified QML. This program is almost same as Figure 5.1 but it does not have `id` property. Instead, it takes a name as an argument of `Rectangle`. This is because `id` property is a special property and it has a different scope rule from other properties. We think it is better that every property of `Rectangle` has the same scoping rule.

5.2.2 Activate Statements

An `activate` statement takes a DSL object and it turns the given DSL object into *active*. Here, 'a DSL object X is active' means 'visible members of X are available without a receiver'. For example, see the following DSL class:

```
Rectangle rect1 {
  property int size : 100
  width : size
  height : rect1.width
}
Rectangle rect2 {
  width : rect1.size / 2
  height : width
}
```

Figure 5.2: A program in simplified QML

```
dsl SizeDSL {
  SizeDSL (Rectangle rect) { this.rect = rect; }
  int "size" () { return (int)rect.get("size"); }
  void "size" ":" _ (int x) { rect.set("size", x); }
  private Rectangle rect;
}
```

this DSL class has two operators: one is a getter operator for `size` (Line 3) and the other is a setter operator for `size` (Line 4). If you activate an instance of `SizeDSL`, you can use these operators after the activation. The following code fragment shows an example code using an `activate` statement:

```
Rectangle rect = ...;
activate new SizeDSL(rect);
size : 100;
System.out.println(size);   // print 100
```

Line 2 activates an instance of `SizeDSL`, so the getter and setter operators are available at Line 3 and Line 4. Actually, Line 3 uses the setter operator and Line 4 uses the getter operator. You can find that the `activate` statement works as a declaration of a variable `size`.

The scope where an activated DSL object is active does not follow the scope rules for variables in the host language. If an DSL object is active at the end of a block, it is also active at the code following the block. Similarly, if an DSL object is active at the end of the body of an operator/method, it is also active at the code following a call of the operator/method. Figure 5.3 is a pseudo code showing the scope of activated DSL object. Since an instance of DSL class `SizeDSL` is active at the code following the method call `declare_size()` (Line 12), we can use getter and setter operators for `size` there. Therefore, the method call `declare_size()` works as a variable declaration.

To determine which DSL object is active at compile time, we introduce `activates` clauses. An `activates` clause is attached to an operator/method where DSL objects that activated at the inside of its body are active at the end of its body. An `activates` clause takes types of activated DSL objects. The

```
void declare_size () {
  // inactive
  Rectangle rect = ...;
  activate new SizeDSL(rect);
  // active
}
void func () {
  // inactive
  {
    // inactive
    declare_size()
    // active
  }
  // active
}
```

Figure 5.3: A pseudo code showing the scope of activated DSL object

```
dsl RectangleDSL {
  void "property" "int" "size" ":" _ (int value)
   activates SizeDSL
    {
      activate new SizeDSL(self);
    }
  RectangleDSL (Rectangle self) { this.self = self; }
  private Rectangle self;
}
```

Figure 5.4: A definition of `property` declaration specialized for `size`

relation between `activate` statements and `activates` clauses is similar to the relation between `throw` statements and `throws` clauses in Java.

Now, we can implement `property` declaration in Figure 5.2. Figure 5.4 is the definition of `property` declaration specialized for `size`. This program declares a DSL class named `RectangleDSL`. `RectangleDSL` contains an instance operator that starts with `property` (Line 2 - 6), a constructor (Line 7), and an instance field named `self` (Line 8). The `property` operator activates a DSL object of type `SizeDSL` in Line 5 and the DSL object is also active at the end of the body. Hence it has `activates` clause taking `SizeDSL`. We can use this operator as follows:

```
property int size : 100;
System.out.println(size);   // print 100
```

```
dsl RectangleDSL <rectName: Ident> {
  void "property" "int" "size" ":" _ (int value)
   activates SizeDSL, QualifiedSizeDSL<rectName>
    {
      self.defineProp("size", value);
      activate new SizeDSL(self);
      activate new QualifiedSizeDSL<rectName>(self);
    }
  RectangleDSL (Rectangle self) { this.self = self; }
  private Rectangle self;
}
```

Figure 5.5: A definition of `property` declaration supporting `rect1.size`

The `property` declaration in Figure 5.2 declares not only `size` but also `rect1.size`. To simulate this, we should implement a DSL class as follows:

```
dsl QualifiedSizeDSL<rectName: Ident> {
  QualifiedSizeDSL (Rectangle rect) { this.rect = rect; }}
  int rectName "." "size" () { return (int)rect.get("size"); }
  private Rectangle rect;
}
```

and then, we should modify the definition of `RectangleDSL` in Figure 5.4. Figure 5.5 is a modified definition of `property` declaration supporting `rect1.size`. The name of a rectangle `rect1` is expressed as a generic name `rectName`. Since the `property` operator activates not only a DSL object of type `SizeDSL` a DSL object of type `QualifiedSizeDSL<rectName>`, we can use instance operators of `QualifiedSizeDSL` after the `property` declaration. Therefore, we can use an expression such as `rect1.size` after the `property` declaration as Figure 5.2.

5.2.3 Scoping Rule for Activated DSL Objects

An `activate` statement can express name binding but the bound name has no scope rules. We should delimit the scope of the bound name to express a local variable. To express scope rules for names, we propose `scope for` clauses. A `scope for` clause is specified to an operand of an operator (or a parameter of a method). The operand expresses the range where the bound names are available.

The `scope for` clause takes a type of a DSL object. If the DSL object is activated in the operand, the instance operators of the DSL object are available after the activation but the operators are not available at outside of the operand. Therefore, if the operators of the DSL object express a variable, the operand works as a scope of the variable. By specifying `scope for` clauses, programmers can apply different scope rules for each type of DSL objects. See Figure 5.2. To emulate the `property` declaration in this figure, programmers should designate

```
<rectName: Ident> void "Rectangle" rectName _
  (RectangleDSL<rectName> |- Void body scope for SizeDSL)
{
  Rectangle rect = new Rectangle();
  body.apply(new RectangleDSL<rectName>(rect););
  declareRectangle(rect);
}
```

Figure 5.6: A definition of `Rectangle` declaration that works as a scope for `size`

the argument of `Rectangle` declaration as the scope rule for `size`. Similarly, they should designate the whole QML program as the scope for `rect1.size`. Since the expression `size` is a call of the instance operator of `SizeDSL`, the scope rule for `size` can be specified by `scope for SizeDSL`.

Figure 5.6 is a definition of `Rectangle` declaration that works as a scope for `size`. This defines an operator starts with `Rectangle`. The operator takes an operand with the type `RectangleDSL<rectName>` $\vdash$ `Void`. Here, the type `RectangleDSL<rectName>` $\vdash$ `Void` is a turnstile type proposed in the previous chapter. A type `T` $\vdash$ `Void` is a special type for a parameter that takes a block as its argument. The operand has a `scope for` clause with the type `SizeDSL`. This specifies that if a DSL object of `SizeDSL` is activated in this argument, the DSL object is not active at the outside of the argument. Therefore, programmers can use the `Rectangle` operator, `property` operator, and `size` operator as follows:

```
Rectangle rect1 {
  // System.out.println(size);   // compile error
  property int size : 100;
  System.out.println(size);   // print 100
};
// System.out.println(size);   // compile error
```

Since the scope of `SizeDSL` is the argument of the `Rectangle` operator, the expression `size` cannot be used at the outside of the `Rectangle` declaration.

Since the scope of `rect1.size` is the whole QML program, a region delimiter that separates the inside and outside of QML program is needed. Figure 5.7 is a definition of a DSL class that contains a static operator expressing the QML region delimiter. An operator starts with `QML` is the QML region delimiter (Line 2 - 4). It takes an operand with the turnstile type `QML` $\vdash$ `Void`. It expresses that users can write a QML program in the operand. The operand has a `scope for` clause with the type `QualifiedSizeDSL<?>`. `?` is a wildcard argument. The scope of `rect1.size` is the operand since it is an operator call of the instance operator of `QualifiedSizeDSL`. From the above, the users can write the following program:

```
dsl QML {
  static void "QML" _ (QML |- Void program scope for QualifiedSizeDSL<?>) {
    program.apply(new QML());
  }
  <rectName: Ident> void "Rectangle" name _
   (RectangleDSL<rectName> |- Void body scope for SizeDSL)
  { ... }
}
```

Figure 5.7: A definition of a DSL class that contains an operator expressing the QML region delimiter

```
QML {
  Rectangle rect1 {
    // System.out.println(rect1.size);   // compile error
    property int size : 100;
    System.out.println(rect1.size);   // print 100
  };
  System.out.println(rect1.size);   // print 100
};
// System.out.println(rect1.size);   // compile error
```

Note that `rect1.size` is available at the outside of the declaration of `rect1`. This is because the scope of `rect1.size` does not follow the scope rules for variables but follows the scope rule of `QualifiedSizeDSL<?>`.

5.2.4 Implementation

We have implemented our proposal by modifying the compiler of ProteaJ2, which is introduced in the previous chapter. The source code of the compiler is available from our github repository[1]. We have modified environments that manage local variables to also manage active DSL objects. Local variables and each active DSL objects can have different scope rules and they can be overlap each other. Therefore, an environment copies its active DSL objects into its parent environment when a program exits a block. `activate` statements and calls of operators/methods that have `activates` clauses add the given types of active DSL objects into the environment. The activated DSL objects are removed from the environment at the end of an argument of a parameter that has a `scope for` clause with the corresponding type. Operands with a turnstile type also add the given types of DSL objects into the environment. Since the turnstile type make several operators available at the operand, the activated DSL objects are removed from the environment at the end of the argument.

[1] https://github.com/csg-tokyo/proteaj2

5.2.5 Definition of simplified QML

By using `activate` statements, `activates` clauses, and `scope for` clauses, we can implement simplified QML shown in Figure 5.2 in ProteaJ2. Figure 5.8 shows a definition of simplified QML. When we use this DSL, we can write the following code:

```
QML MyQMLProgram {
  Rectangle rect1 {
    property int size : 100;
    width : size;
    height : rect1.width;
  };
  Rectangle rect2 {
    width : rect1.size / 2;
    height : width;
  };
};
```

You can find that this code looks very similar to the code in Figure 5.2.

Figure 5.8 defines `Prop1DSL` and `Prop2DSL` instead of `SizeDSL` and `QualifiedSizeDSL`. `Prop1DSL` and `Prop2DSL` are more general than `SizeDSL` and `QualifiedSizeDSL` because they take a generic name `propName` for a property name instead of the fixed property name `size`. Similarly, the `property` operator defined in `RectangleDSL` also takes a generic name `propName` and it activates `Prop1DSL` and `Prop2DSL` that take the generic name `propName` as their type argument. Hence users can define properties with their own names and access the properties by their names. `Prop1DSL` and `Prop2DSL` have a field `id`, which is an identifier of the property at run-time. The value of `id` is not the same as `propName` since programmers cannot get the value of a generic name at run-time in the current definition of ProteaJ2.

`RectangleDSL` in Figure 5.8 has getter and setter operators for `width` and `height` in addition to the `property` operator. These operators are available at the body of `Rectangle` declarations since the `Rectangle` operator defined in the DSL class `QML` takes a parameter with the turnstile type `RectangleDSL<progName, rectName> ⊢ Void`. Hence the expressions `width` and `height` are available at the body of `Rectangle` declarations. The expression `rect1.width` is also available at the body of `Rectangle` declarations for the same reason, and furthermore, it is also available at the outside of the `Rectangle` declarations. The `Rectangle` operator activates `RectanglePropsDSL<progName, rectName>`. Since the DSL class `RectanglePropsDSL` defines getter operators for `width` and `height` by qualified name such as `rect1.width`, the expression `rect1.width` is available after the definition of the rectangle `rect1`.

```
dsl QML <progName: Ident> {
  static void "QML" progName _
   (QML<progName> |- Void program
      scope for Prop2DSL<progName, ?, ?, ?>, RectanglePropsDSL<progName, ?>) {
    program.apply(new QML<progName>());
  }
  <rectName: Ident> void "Rectangle" rectName _
   (RectangleDSL<progName, rectName> |- Void body scope for Prop1DSL<rectName, ?, ?>)
   activates RectanglePropsDSL<progName, rectName>
  {
    Rectangle rect = new Rectangle();
    body.apply(new RectangleDSL<progName, rectName>(rect););
    declareRectangle(rect);
    activate new RectanglePropsDSL<progName, rectName>(rect);
  }
}
dsl RectangleDSL <scope: Ident, rectName: Ident> {
  <propName: Ident, T> void "property" T propName ":" _ (T value)
    activates Prop1DSL<rectName, propName, T>, Prop2DSL<scope, rectName, propName, T>
     {
       String id = genSym();
       self.defineProp(id, value);
       activate new Prop1DSL<rectName, propName, T>(self, id);
       activate new Prop2DSL<scope, rectName, propName, T>(self, id);
     }
  int "width" () { return self.getWidth(); }
  int "height" () { return self.getHeight(); }
  int rectName "." "width" () { return self.getWidth(); }
  int rectName "." "height" () { return self.getHeight(); }
  void "width" ":" _ (int w) { self.setWidth(w); }
  void "height" ":" _ (int h) { self.setHeight(h); }
  RectangleDSL (Rectangle self) { this.self = self; }
  private Rectangle self;
}
dsl RectanglePropsDSL <scope: Ident, rectName: Ident> {
  int rectName "." "width" () { return self.getWidth(); }
  int rectName "." "height" () { return self.getHeight(); }
  RectanglePropsDSL (Rectangle self) { this.self = self; }
  private Rectangle self;
}
dsl Prop1DSL <scope: Ident, propName: Ident, T> {
  Prop1DSL (Rectangle rect, String id) {
    this.rect = rect;
    this.id = id;
  }
  T propName () { return (T)rect.get(id); }
  void propName ":" _ (T x) { rect.set(id, x); }
  private Rectangle rect;
  private String id;
}
dsl Prop2DSL <scope: Ident, name1: Ident, name2: Ident, T> {
  Prop2DSL (Rectangle rect, String id) {
    this.rect = rect;
    this.id = id;
  }
  T name1 "." name2 () { return (T)rect.get(id); }
  private Rectangle rect;
  private String id;
}
```

Figure 5.8: A definition of simplified QML

5.3 Discussion

5.3.1 Contributions

Our proposal enables to implement embedded DSLs that have declarations. Most programming languages have declarations such as local variable declarations, so our proposal enables programmers to implement various programming languages as embedded DSLs. In our proposal, a declaration is expressed by an activation of a DSL object with `activate` statements. If a DSL object is active, the instance operators of the DSL object are available there without a receiver. The instance operators simulate syntax that is available at the local scope.

The important feature of our proposal is that scope of activated DSL objects is not the same as the scope of local variables. The scope of activated DSL objects is specified by `scope for` clauses for each type of DSL objects. This enables programmers to define a kind of namespaces of DSLs. Programmers can control visibility of operators (i.e. syntax) by using `scope for` clauses without considering the scope of local variables of the host language.

5.3.2 Limitations

In our proposal, `size` and `rect1.size` are expressed by calls of different operators. Similarly, `x.y` and `x.y.z` are expressed by calls of different operators. This implies that the nesting level of DSL namespaces has limitation. Programmers should define similar operators for each nesting level.

Our proposal cannot implement arbitrary declarations. For example, it cannot implement class declarations in Java. This is because `activate` statements can enable several syntax locally but cannot define new types.

5.3.3 Future Work

In the current design, programmers cannot define the scope rules of `if`-like statement for a DSL object. If programmers define their own `if` statement and users activate a DSL object in the `then` block in the `if` statement, the activated DSL object is active in the `else` block. This is because current design does not consider lazy evaluation or function literals. To resolve this problem, we should be able to specify `activates` clauses as a part of a type. This makes it possible to express an activation of a DSL object after execution of the function object.

Chapter 6

Concluding Remarks

In this dissertation, we propose

1. user-defined operators for composable syntax extensions, named protean operators
2. a variant of lambda expressions for expressing user-defined language constructs involving name binding, named context-sensitive expressions,
3. and language constructs for defining user-defined declarations, named `activate` statements, `activates` clauses, and `scope for` clauses.

We also show programming languages named ProteaJ and ProteaJ2, which support our proposals. The source code of the compiler of ProteaJ and ProteaJ2 is available from our github repository[1].

Protean operators are user-defined operators whose syntax consists of operator names and operands. The characteristic property of protean operators is that a protean operator is overloaded by operators that have the same syntax but have different return type or operand types. This means that static types work as if they are non-terminal symbols because a protean operator is available only at an expression where its return type is expected. Hence programmers can define various syntax by declaring protean operators and using their return type and operand types as non-terminal symbols. Actually, programmers can define any parsing expression grammars (PEGs) in ProteaJ. Protean operators can also express user-defined literals. Furthermore, protean operators have contextual-composable because the compiler can distinguish operators even if they have the same syntax except types. Users can use operators without care about conflicts of syntax.

Context-sensitive expressions are a variant of lambda expressions, for expressing name binding and scope rules. They take parameters but the parameters are not explicitly written. Since the parameter names are not given, the parameters cannot be accessed via the parameter names. Instead, instance

[1] https://github.com/csg-tokyo/proteaj and https://github.com/csg-tokyo/proteaj2

members of the parameters are available without receivers in context-sensitive expressions. In ProteaJ2, protean operators can be instance members. This means that we can express syntax that is available only at context-sensitive expressions. Hence, we can express name binding and scope rules by using context-sensitive expressions. ProteaJ2 supports turnstile types, which express the types of context-sensitive expressions. ProteaJ2 also supports generic names to recognize arbitrary names given by end-users. User-defined language constructs implemented by using context-sensitive expressions can be safely composed because the scope of protean operators is expressed by turnstile types.

An `activate` statement takes an object as its argument and enables its members after the statement without receivers. The scope of activated members does not follow the scope of local variables in the host language. If the `activate` statement is used in the body of an operator/method, the visible members are also available at the call-site of the operator/method. Hence, we can implement an operator that emulates a variable declaration. The scope of activated members is given by `scope for` clauses. A `scope for` clause takes several types and it expresses the scope of activated members of given objects. Programmers can implement different scope rules for each embedded DSLs. To statically check which members are activated, ProteaJ also provides `activates` clauses. Since the compiler can check what members are available by checking signatures of methods/operators, user-defined declarations and their scope rules can be safely composed.

The contribution of this dissertation is giving a platform of DSLs with good expressiveness and composability. Our approach is based on embedded DSLs for achieving composability. Protean operators are a powerful syntax extension mechanism, which improves the expressiveness of DSLs without breaking the composability. Context-sensitive expressions enable programmers to define language constructs involving name binding. `activate` statements and `scope for` clauses enable programmers to define declarations. Both also improve the expressiveness of DSLs by supporting a part of type checking of DSLs. We have developed ProteaJ and ProteaJ2, which are programming languages supporting our proposals. Table 6.1 shows the positions of ProteaJ and ProteaJ2. Since ProteaJ only supports protean operators, programmers can implement parsers for DSLs with contextual-composability but they cannot implement type checkers for DSLs. In ProteaJ2, programmers also can implement parsers for DSLs with contextual-composability, and in addition to that, they can implement a part of type checkers, name binding.

	expressiveness (static-checking)			
	parsing	type checking	composability	editor
Spoofax	CFG	yes	no	text
MPS	any grammar	yes	no	projectional
ProteaJ	PEG	no	contextual	text
ProteaJ2	PEG	partial	contextual	text
string embedding	dynamic checking	no	bounded	text
fluent interfaces	host syntax	no	contextual	text
mixfix operators	mixfix syntax	no	contextual	text
Scala-virtualized	host syntax	no	contextual	text
Recaf	host syntax	partial *1	contextual	text
C / C++ macros	function call syntax	yes	heuristic	text
reader macros	CFG	yes	bounded	text
syntactic macros	host syntax	yes	heuristic	text
SugarJ	CFG	yes	heuristic	text
Wyvern	Adams grammar	no	bounded	text
Cedalion	any grammar	partial? *2	contextual? *2	projectional

*1 only name binding for special syntax

*2 According to the paper [50], programmers can define their own type system for their embedded DSLs in Cedalion, however, we could not find the description how to implement the type system and the limitation of the approach. If it has no limitation, it might break the composability.

Table 6.1: Positions of ProteaJ and ProteaJ2

Bibliography

[1] Michael D. Adams. "Principled Parsing for Indentation-sensitive Languages: Revisiting Landin's Offside Rule". In: *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '13. Rome, Italy, 2013, pp. 511–522. DOI: 10.1145/2429069.2429129.

[2] Ulf Norell et al. *Agda's documentation*. 2016. URL: https://agda.readthedocs.io (visited on 05/29/2017).

[3] Jonathan Bachrach and Keith Playford. *D-expressions: Lisp power, Dylan style*. 1999. URL: https://people.csail.mit.edu/jrb/Projects/dexprs.pdf (visited on 05/25/2017).

[4] Jonthan Bachrach and Keith Playford. "The Java Syntactic Extender (JSE)". In: *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '01. Tampa Bay, FL, USA, 2001, pp. 31–42. DOI: 10.1145/504282.504285.

[5] Aggelos Biboudis, Pablo Inostroza, and Tijs van der Storm. "Recaf: Java Dialects As Libraries". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2016. Amsterdam, Netherlands, 2016, pp. 2–13. DOI: 10.1145/2993236.2993239.

[6] Claus Brabrand and Michael I. Schwartzbach. "Growing Languages with Metamorphic Syntax Macros". In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*. PEPM '02. Portland, Oregon, 2002, pp. 31–40. DOI: 10.1145/503032.503035.

[7] Edwin Brady and Kevin Hammond. "Resource-Safe Systems Programming with Embedded Domain Specific Languages". In: *Practical Aspects of Declarative Languages: 14th International Symposium, PADL 2012, Philadelphia, PA, USA, January 23-24, 2012. Proceedings*. Ed. by Claudio Russo and Neng-Fa Zhou. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 242–257. ISBN: 978-3-642-27694-1. DOI: 10.1007/978-3-642-27694-1_18.

[8] Mark G. J. van den Brand et al. "Disambiguation Filters for Scannerless Generalized LR Parsers". In: *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*. Ed. by R. Nigel Horspool. Vol. 2304. Lecture Notes in Computer Science. Springer, 2002, pp. 143–158. DOI: 10.1007/3-540-45937-5_12.

[9] Martin Bravenboer and Eelco Visser. "Concrete Syntax for Objects: Domain-specific Language Embedding and Assimilation Without Restrictions". In: *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '04. Vancouver, BC, Canada, 2004, pp. 365–383. DOI: 10.1145/1028976.1029007.

[10] Martin Bravenboer et al. "Stratego/XT 0.17. A language and toolset for program transformation". In: *Science of Computer Programming* 72.1 (2008), pp. 52–70. DOI: 10.1016/j.scico.2007.11.003.

[11] Luca Cardelli, Florian Matthes, and Martin Abadi. *Extensible syntax with lexical scoping*. Tech. rep. SRC-RR-121. Systems Research Center, Feb. 1994. URL: http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-121.pdf (visited on 03/14/2017).

[12] M. Clavel et al. "Maude: specification and programming in rewriting logic". In: *Theoretical Computer Science* 285.2 (2002), pp. 187–243. DOI: 10.1016/S0304-3975(01)00359-0.

[13] Nils Anders Danielsson and Ulf Norell. "Parsing Mixfix Operators". In: *Proceedings of the 20th International Conference on Implementation and Application of Functional Languages*. IFL'08. Hatfield, UK, 2011, pp. 80–99. DOI: 10.1007/978-3-642-24452-0_5.

[14] F. L. Deremer. *Practical translators for LR(k) languages*. Tech. rep. Cambridge, 1969. URL: http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-065.pdf (visited on 04/12/2017).

[15] Jay Earley. "An Efficient Context-free Parsing Algorithm". In: *Commun. ACM* 13.2 (Feb. 1970), pp. 94–102. ISSN: 0001-0782. DOI: 10.1145/362007.362035.

[16] Torbjörn Ekman and Görel Hedin. "The Jastadd Extensible Java Compiler". In: *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*. OOPSLA '07. Montreal, Quebec, Canada, 2007, pp. 1–18. DOI: 10.1145/1297027.1297029.

[17] Torbjörn Ekman and Görel Hedin. "The JastAdd System – Modular Extensible Compiler Construction". In: *Sci. Comput. Program.* 69.1-3 (Dec. 2007), pp. 14–26. DOI: 10.1016/j.scico.2007.02.003.

[18] Sebastian Erdweg et al. "SugarJ: Library-based Syntactic Language Extensibility". In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '11. Portland, Oregon, USA, 2011, pp. 391–406. DOI: 10.1145/2048066.2048099.

[19] 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. Pittsburgh, PA, USA, 2002, pp. 36–47.

[20] 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. Venice, Italy, 2004, pp. 111–122. DOI: 10.1145/964001.964011.

[21] Apache Software Foundation. *Apache log4j 1.2*. (This software is end of life but we can get the source code yet.) 1999. URL: http://logging.apache.org/log4j/1.2/ (visited on 05/29/2017).

[22] Martin Fowler. *FluentInterface*. Dec. 2005. URL: https://www.martinfowler.com/bliki/FluentInterface.html (visited on 04/12/2017).

[23] Martin Fowler. *Language Workbenches: The Killer-App for Domain Specific Languages?* 2005. URL: https://www.martinfowler.com/articles/languageWorkbench.html (visited on 05/26/2017).

[24] Martin Fowler. *ProjectionalEditing*. Jan. 2008. URL: https://martinfowler.com/bliki/ProjectionalEditing.html (visited on 05/24/2017).

[25] Inc. Free Software Foundation. *The C Preprocessor*. 1987. URL: https://gcc.gnu.org/onlinedocs/cpp/index.html (visited on 05/23/2017).

[26] R. Frost and J. Launchbury. "Constructing Natural Language Interpreters in a Lazy Functional Language". In: *Comput. J.* 32.2 (Apr. 1989), pp. 108–121. DOI: 10.1093/comjnl/32.2.108.

[27] S. Ginsburg and S. Greibach. "Deterministic context free languages". In: *6th Annual Symposium on Switching Circuit Theory and Logical Design (SWCT 1965)*. Oct. 1965, pp. 203–220. DOI: 10.1109/FOCS.1965.7.

[28] Albert Gräf. *The Pure Manual*. 2009. URL: https://puredocs.bitbucket.io/pure.html (visited on 05/29/2017).

[29] Paul Graham. *On LISP: Advanced Techniques for Common LISP*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1994. ISBN: 9780130305527.

[30] J. Heering et al. "The Syntax Definition Formalism SDF – Reference Manual –". In: *SIGPLAN Not.* 24.11 (Nov. 1989), pp. 43–75. DOI: 10.1145/71605.71607.

[31] Michael Homer, Timothy Jones, and James Noble. "From APIs to Languages: Generalising Method Names". In: *Proceedings of the 11th Symposium on Dynamic Languages*. DLS 2015. Pittsburgh, PA, USA, 2015, pp. 1–12. DOI: 10.1145/2816707.2816708.

[32] Paul Hudak. "Modular Domain Specific Languages and Tools". In: *Proceedings of the 5th International Conference on Software Reuse*. ICSR '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 134–. DOI: 10.1109/ICSR.1998.685738.

[33] Graham Hutton. "Higher-order functions for parsing". In: *Journal of Functional Programming* 2.3 (July 1992), pp. 323–343. DOI: 10.1017/S0956796800000411.

[34] Graham Hutton and Erik Meijer. *Monadic Parser Combinators*. Technical Report NOTTCS-TR-96-4. Department of Computer Science, University of Nottingham, 1996. URL: http://www.cs.nott.ac.uk/~pszgmh/monparsing.pdf (visited on 04/12/2017).

[35] Kazuhiro Ichikawa and Shigeru Chiba. "Composable User-defined Operators That Can Express User-defined Literals". In: *Proceedings of the 13th International Conference on Modularity*. MODULARITY '14. Lugano, Switzerland, 2014, pp. 13–24. DOI: 10.1145/2577080.2577092.

[36] JetBrains. *Meta Programming System*. URL: https://www.jetbrains.com/mps/ (visited on 05/24/2017).

[37] Trevor Jim, Yitzhak Mandelbaum, and David Walker. "Semantics and Algorithms for Data-dependent Grammars". In: *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '10. Madrid, Spain, 2010, pp. 417–430. DOI: 10.1145/1706299.1706347.

[38] Stephen C Johnson. *Yacc – Yet another compiler-compiler*. Computer Science Technical Report 32. Murray Hill, NJ, USA: AT&T Bell Laboratories, 1975. URL: http://dinosaur.compilertools.net/yacc/index.html (visited on 04/12/2017).

[39] *JSGLR: An SGLR Parse Table Evaluator for Java*. URL: http://strategoxt.org/Stratego/JSGLR (visited on 05/29/2017).

[40] Tadao Kasami. *An Efficient Recognition and Syntax-Analysis Algorithm for Context-Free Languages*. Coordinated Science Laboratory Report R-257. Coordinated Science Laboratory, Mar. 1966. URL: http://hdl.handle.net/2142/74304 (visited on 05/29/2017).

[41] Lennart C. L. Kats and Eelco Visser. "The Spoofax language workbench: rules for declarative specification of languages and IDEs". In: *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*. Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. Reno/Tahoe, Nevada, 2010, pp. 444–463. DOI: 10.1145/1869459.1869497.

[42] Donald E. Knuth. "Backus Normal Form vs. Backus Naur Form". In: *Commun. ACM* 7.12 (Dec. 1964), pp. 735–736. ISSN: 0001-0782.

[43] Donald E. Knuth. "On the translation of languages from left to right". In: *Information and Control* 8.6 (1965), pp. 607–639. DOI: 10.1016/S0019-9958(65)90426-2.

[44] Donald E. Knuth. "Semantics of context-free languages". In: *Mathematical systems theory* 2.2 (1968), pp. 127–145. DOI: 10.1007/BF01692511.

[45] Eugene Kohlbecker et al. "Hygienic Macro Expansion". In: *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*. LFP '86. Cambridge, Massachusetts, USA, 1986, pp. 151–161. DOI: 10.1145/319838.319859.

[46] Gabriël D. P. Konat et al. "Declarative Name Binding and Scope Rules". In: *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*. Ed. by Krzysztof Czarnecki and Görel Hedin. Vol. 7745. Lecture Notes in Computer Science. Springer, 2012, pp. 311–331. DOI: 10.1007/978-3-642-36089-3_18.

[47] Daan Leijen and Erik Meijer. *Parsec: Direct Style Monadic Parser Combinators For The Real World*. Technical Report UU-CS-2001-35. Departement of Computer Science, Universiteit Utrecht, July 2001. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/parsec-paper-letter.pdf (visited on 04/12/2017).

[48] John Levine. *Flex & Bison: Text Processing Tools*. 1st ed. O'Reilly Media, Aug. 2009. ISBN: 9780596155971.

[49] Jeffrey R. Lewis et al. "Implicit Parameters: Dynamic Scoping with Static Types". In: *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '00. Boston, MA, USA, 2000, pp. 108–118. DOI: 10.1145/325694.325708.

[50] David H. Lorenz and Boaz Rosenan. "Cedalion: A Language for Language Oriented Programming". In: *SIGPLAN Not.* 46.10 (Oct. 2011), pp. 733–752. ISSN: 0362-1340. DOI: 10.1145/2076021.2048123.

[51] The Qt Company Ltd. *QML Reference*. 2017. URL: http://doc.qt.io/qt-5/qmlreference.html (visited on 10/13/2017).

[52] Simon Marlow. *Haskell 2010 Language Report*. 2010. URL: https://www.haskell.org/onlinereport/haskell2010/ (visited on 04/12/2017).

[53] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-Step Guide, 2nd Edition*. 2nd. USA: Artima Incorporation, 2011. ISBN: 9780981531649.

[54] Cyrus Omar, Chenglong Wang, and Jonathan Aldrich. "Composable and Hygienic Typed Syntax Macros". In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. SAC '15. Salamanca, Spain, 2015, pp. 1986–1991. DOI: 10.1145/2695664.2695936.

[55] Cyrus Omar et al. "Safely Composable Type-Specific Languages". In: *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*. 2014, pp. 105–130. DOI: 10.1007/978-3-662-44202-9_5.

[56] Terence Parr and Kathleen Fisher. "LL(*): The Foundation of the ANTLR Parser Generator". In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA, 2011, pp. 425–436. DOI: 10.1145/1993498.1993548.

[57] Lawrence C. Paulson. *Isabelle: a Generic Theorem Prover*. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 1994. DOI: 10.1007/BFb0030541.

[58] F. Pfenning and C. Elliott. "Higher-order Abstract Syntax". In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*. PLDI '88. Atlanta, Georgia, USA, 1988, pp. 199–208. DOI: 10.1145/53990.54010.

[59] Jon Rafkind and Matthew Flatt. "Honu: Syntactic Extension for Algebraic Notation Through Enforestation". In: *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*. GPCE '12. Dresden, Germany, 2012, pp. 122–131. DOI: 10.1145/2371401.2371420.

[60] Tiark Rompf et al. "Scala-Virtualized: linguistic reuse for deep embeddings". In: *Higher-Order and Symbolic Computation* 25.1 (2012), pp. 165–207. DOI: 10.1007/s10990-013-9096-9.

[61] *ScalaTest*. URL: http://www.scalatest.org/ (visited on 05/29/2017).

[62] Tim Sheard. "Using MetaML: A Staged Programming Language". In: *Advanced Functional Programming: Third International School, AFP'98, Braga, Portugal, September 12-19, 1998, Revised Lectures*. Ed. by S. Doaitse Swierstra, José N. Oliveira, and Pedro R. Henriques. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 207–239. DOI: 10.1007/10704973_5.

[63] Tim Sheard and Simon Peyton Jones. "Template Meta-programming for Haskell". In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*. Haskell '02. Pittsburgh, Pennsylvania, 2002, pp. 1–16. DOI: 10.1145/581690.581691.

[64] Erik Silkensen and Jeremy Siek. "Well-Typed Islands Parse Faster". In: *Trends in Functional Programming: 13th International Symposium*. TFP '12. St. Andrews, UK, 2013, pp. 69–84. DOI: 10.1007/978-3-642-40447-4_5.

[65] Charles Simonyi. *The Death of Computer Languages, The Birth of Intentional Programming*. Technical Report MSR-TR-95-52. Microsoft Research, Sept. 1995. URL: https://www.microsoft.com/en-us/research/publication/the-death-of-computer-languages-the-birth-of-intentional-programming/ (visited on 05/24/2017).

[66] Kamil Skalski, Michal Moskal, and Pawel Olszta. *Meta-programming in Nemerle*. 2004. URL: http://camlunity.ru/swap/Library/Computer%5C%20Science/Metaprogramming/Macros/Meta-programming%5C%20in%5C%20Nemerle.pdf (visited on 05/30/2017).

[67] Michael Sperber et al. *Revised^6 Report on the Algorithmic Language Scheme*. Sept. 2007. URL: http://www.r6rs.org/final/html/r6rs/r6rs.html (visited on 05/25/2017).

[68] Oliviero Stock, Rino Falcone, and Patrizia Insinnamo. "Island Parsing and Bidirectional Charts". In: *Proceedings of the 12th Conference on Computational Linguistics - Volume 2*. COLING '88. Budapest, Hungry, 1988, pp. 636–641. DOI: 10.3115/991719.991768.

[69] The Coq Development Team. *The Coq Proof Assistant Reference Manual*. Version 8.6. INRIA. Dec. 2016. URL: https://coq.inria.fr/refman/ (visited on 03/14/2017).

[70] *The Rust Programming Language*. URL: https://doc.rust-lang.org/book/README.html (visited on 05/26/2017).

[71] Sam Tobin-Hochstadt et al. "Languages As Libraries". In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA, 2011, pp. 132–141. DOI: 10.1145/1993498.1993514.

[72] Masaru Tomita. "An Efficient Context-free Parsing Algorithm for Natural Languages". In: *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 2*. IJCAI'85. Los Angeles, California: Morgan Kaufmann Publishers Inc., 1985, pp. 756–764. ISBN: 0-934613-02-8, 978-0-934-61302-6.

[73] Laurence Tratt. "Domain Specific Language Implementation via Compile-time Meta-programming". In: *ACM Trans. Program. Lang. Syst.* 30.6 (Oct. 2008), 31:1–31:40. DOI: 10.1145/1391956.1391958.

[74] Eric Van Wyk et al. "Silver: An Extensible Attribute Grammar System". In: *Electron. Notes Theor. Comput. Sci.* 203.2 (Apr. 2008), pp. 103–116. DOI: 10.1016/j.entcs.2008.03.047.

[75] Vlad A. Vergu, Pierre Néron, and Eelco Visser. "DynSem: A DSL for Dynamic Semantics Specification". In: *26th International Conference on Rewriting Techniques and Applications, RTA 2015, June 29 to July 1, 2015, Warsaw, Poland*. Ed. by Maribel Fernández. Vol. 36. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 365–378. DOI: 10.4230/LIPIcs.RTA.2015.365.

[76] Eelco Visser. *Scannerless Generalized-LR Parsing*. Tech. rep. P9707. Programming Research Group, University of Amsterdam, July 1997. URL: http://researchr.org/publication/Vis97.sglr (visited on 05/29/2017).

[77] Eelco Visser. "Stratego: A Language for Program Transformation Based on Rewriting Strategies". In: *Proceedings of the 12th International Conference on Rewriting Techniques and Applications*. RTA '01. 2001, pp. 357–362. DOI: 10.1007/3-540-45127-7_27.

[78] Philip Wadler. "Comprehending Monads". In: *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. LFP '90. Nice, France, 1990, pp. 61–78. DOI: 10.1145/91556.91592.

[79] Philip Wadler. "How to Replace Failure by a List of Successes". In: *Proc. of a Conference on Functional Programming Languages and Computer Architecture*. Nancy, France, 1985, pp. 113–128. DOI: 10.1007/3-540-15975-4_33.

[80] Martin P Ward. "Language-oriented programming". In: *Software-Concepts and Tools* 15.4 (1994), pp. 147–161. URL: http://www.cse.dmu.ac.uk/~mward/martin/papers/middle-out-t.pdf (visited on 04/12/2017).

[81] Alessandro Warth, James R. Douglass, and Todd Millstein. "Packrat Parsers Can Support Left Recursion". In: *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*. PEPM '08. 2008, pp. 103–110. DOI: 10.1145/1328408.1328424.

[82] Niklaus Wirth. "What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions?" In: *Commun. ACM* 20.11 (Nov. 1977), pp. 822–823. DOI: 10.1145/359863.359883.

[83] Daniel H. Younger. "Context-free language processing in time n^3". In: *7th Annual Symposium on Switching and Automata Theory (swat 1966)*. Oct. 1966, pp. 7–20. DOI: 10.1109/SWAT.1966.7.