Composable User-Defined Operators
That Can Express User-Defined Literals

Kazuhiro Ichikawa
The University of Tokyo
ichikawa@csg.cs.i.u-tokyo.ac.jp

Shigeru Chiba
The University of Tokyo
chiba@acm.org

Abstract
This paper proposes new composable user-defined operators, named protean operators. They can express various language extensions including user-defined literals such as regular expression literals as well as user-defined expressions. Their expressiveness is equivalent to Parsing Expression Grammar (PEG). The operators have two important features to be parsed in pragmatic time: overloading by return type and a precedence rule for operators. They can be parsed efficiently even if they express user-defined literals since ambiguities in the grammar are removed by these two features. The overloading by return type enables us to consider static types as non-terminal symbols in the grammar. The compiler can use static type information for parsing. It can resolve ambiguities of the rules with the same syntax but a different type. Protean operators with the same return type require programmers to declare the precedence among them. These precedence rules enable completely removing ambiguities from the grammar since all the rules applicable to the same place are ordered. Thus, the expressions including protean operators can be parsed in pragmatic time. We have implemented a language that is a subset of Java but supports protean operators. We present an experiment to show that the programs including user-defined literals cannot be parsed in pragmatic time in existing approaches but can be efficiently parsed in our approach.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features

General Terms Languages, Algorithms, Experimentation

Keywords user-defined operators; parsing; syntax extensions

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 [11] (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. This paper 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 [7] 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 number of DSLs have their own literals that are not included in general purpose languages for describing programs concisely and safely. For example, flex [15], which is a DSL for generating a scanner, has literals for expressing regular expressions. Without user-defined literals, they must be expressed by character strings; it weakens maintainability and safety since the compiler does not check that the string character fits the literal syntax. User-defined literals introduce a large number of ambiguities and user-defined literals are included at a number of places in the source program. Thus, the parser cannot parse a program in pragmatic time.

In this paper, 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. We have conducted an experiment for demonstrating that ProteaJ can parse expressions including user-defined literals efficiently even though a naive parsing method cannot parse them in pragmatic time.

In the rest of this paper, we first show the limitation of existing composable user-defined operators. Then we propose new compos-
operands are separated by an operator-name. For example, the fol-

Mixfix indicates prefix, postfix, in-

these operators as if it is written in a domain-specific or "natural"

Figure 1. An example of composable operators

Figure 2. Composable user-defined operators with new syntax

able user-defined operators, named protean operators, and we show

Composable user-defined operators are useful for implementing in-

Mifix operators [7] are a powerful implementation of com-

Scannerless parsing is one of the implementation techniques of

Figure 2 shows a unit test program

A typical parser for user-defined operators generates all possible

The following code is an example of a regular expression literal:

A typical scannerless parser is inefficient when parsing a pro-

Scannerless Generalized LR (SGLR) [25][23] parser is a well-known im-

the right-hand side of = is a regular expression literal that denotes

we propose new composable user-defined operators, named pro-

3. Proposal: Protean Operators

We propose new composable user-defined operators, named protean operators. They can express user-defined literals such as regu-

However, mifix operators do not have sufficient syntactic expressiveness for implementing a certain kind of internal DSLs. Mifix operators cannot express complicated literals since they do not support literal-level syntax extension. The following code is an example of a regular expression literal:

3. Motivation

Composable user-defined operators are useful for implementing in-

Mifix operators can express user-defined literals when the host language is implemented by using a scannerless parser and they support nameless operators.

A typical unit test program

Therefore, the syntax including regular expression literals should be an ambiguous grammar such as in Figure 3 and the ambiguities must be resolved by the type checker.

A typical scannerless parser is inefficient when parsing a pro-

Scannerless parsers are proportional to the degree of ambiguities (nondeterministics) in the grammar, and the worst-case time complexity is \(O(n^3)\) (\(n\) is the input length). Note that \(n\) in this 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.

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

Figure 3. An example of ambiguous grammar

\[\begin{align*}
& h &| e | l &| + | o \\
\end{align*}\]

\[\begin{align*}
& h &| e | l &| + | o \\
\end{align*}\]

\[\begin{align*}
& h &| e | l &| + | o \\
\end{align*}\]

\[\begin{align*}
& h &| e | l &| + | o \\
\end{align*}\]
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.

3.1 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, and 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. Protean operators support operator precedence and associativity for ease of use. Protean operators are totally ordered by operator precedence. Figure 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. The optional part enclosed by curly braces indicates an operator associativity. left-assoc is left-associative and non-assoc is non-associative. The operator precedence is shown in the last two lines in the figure. The literal hel+o is parsed as a regular expression literal as shown in Figure 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 operands and it returns a new literal expressing a regular expression constructed by the concatenation of the given regular expressions.

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. These operators can be considered as a simple user-defined literal, which consists of one token. Each Letter literal is converted into a Regex literal by the nameless operator [ _:Letter ]:Regex at (A) in Figure 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 at (B) in Figure 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 h. It is formed as a sequence of a regular expression constructed by a postfix unary operator [ _:Regex ]:Regex shown at (C) in Figure 4. It represents a regular expression that accepts one or more sequences of l. Then h and l+ tied by [ _: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.

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 the 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 hel = 3;
- int x1 = hel+o; // 5
- Regex x2 = hel+o; // helo, hello, hello, ...

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 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 this paper, 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(1++)o by applying [ _:Regex ++ ]:Regex rather than he((1++)+o) by [ _:Regex * ]:Regex since the former has higher precedence. The literal hel+o is interpreted as...
A drawback of protean operators is a limited kind of places where the operators are available. Protean operators are available only in the expressions whose expected type are statically determined before parsing the expression. The places where protean operators are available depend on a 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 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, protean operators would not be desirable there. The target of a member access could be a complicated expression like:

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

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

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

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

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

Here, matches is a binary infix operator. 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) takes the target type T 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. For example, assuming that \[ \sin \] is a binary operator that returns the sine value of the given angle, in the following code, the expected type of \( \sin 0.0 \) is unknown:

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

If the cast operator explicitly specified the source type as follows:

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

Then the expected type of \( \sin 0.0 \) would be known as \( \text{double} \).

In the argument of throw statement in Java, it is difficult to determine available protean operators properly. According to Table 1, the expected type of the argument of throw statement is \( \text{Throw} \); however, it is not proper because, it must throw either an \( \text{Error} \), a \( \text{RuntimeException} \), 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 the expressions whose expected type are statically determined, also makes an obstacle to use generics. Assuming that the generic type \( \text{List}[T] \) is available, we would like to define the following operator:

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

In this operator, the type parameter \( T \) cannot be inferred from the return type. Hence, the expected type \( \text{List}[T] \) of the argument cannot be determined. We cannot use protean operators at the operand of this operator. Since we currently do not have a good solution of this problem, our compiler introduced in section 4 does not support generics.

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

**Table 1. The expected types of Java expressions**
The procedure parseStmt is an entry point of the parser. The procedure parseWhileStmt parses a while statement. Since the condition expression in the while statement must return a boolean value, the expected type of the condition expression is boolean. Thus the call callExp(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. The initialization expression of the declaration is parsed by using the expected type specified by the type of the declared variable. 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 an expression in the host language. 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 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 Expression Grammar (PEG) [10] with left recursion as shown later. Our parsing method can be regarded as a variant of recursive descent parsing with memoization for the PEG generated from the operators. The memoization is used for eliminating the cost of backtracking. Our method can be used for scannerless parsing since its parsing-time complexity is $O(n)$. The original packrat parsing does not support left recursion, however, we added the left-recursion support by a small extension. Unfortunately, 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.

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 ← 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 $e_1\, e_2$, and ordered-choice operators $e_1\,|\,\, e_2$. Here, $e_1$ and $e_2$ are a parsed expression. 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 rules 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. A protean operator returning a value of different type is translated into a different non-terminal symbol. If an operator returns Letter, it is translated into a non-terminal symbol Expr<Letter>. The parsing precedence is translated into the ordered-choice rule in PEG. For example, see the following protean operators:

```
// entry point
// ops is the definitions of the operators collected before parsing
def parseStmt(ops, env) {
    r = parseWhileStmt(ops, env)
    if (r is Success) return r
    else backtrack
        // parse by the other control flow rules similarly
    r = parseVarDecl(ops, env)
    if (r is Success) return r
    else backtrack
        // parse by the other state rules similarly
    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(t, n, v)
    else return Failure
}
// VarDecl → TypeName<T> Identifier "=" Expr<T>
def parseVarDecl(ops, env) {
    t = parse the identifier rule
    n = parse by the identifier rule
    e = scan("=")
    v = parseExpr(get a type whose name is t, ops, env)
    if ((e 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
}
// Stmt → Expr<Void> ":;"
def parseStmtVoid(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<T>(typ, ops, env) {
    // operators have been sorted by parsing precedence
    operators = get operators returning typ from ops
    for (op in operators) {
        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 = parseExp(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 by the name of r from env
    if ((v is Success && v's type is typ) return VarAccess(v)
    else backtrack
        // parse by any other predefined rules }
    return Failure
}
```

Figure 6. the parsing algorithm for statements
Then we add an additional operator if the operator associativity is left-assoc. Here, the two different protean operators return the same type.

<table>
<thead>
<tr>
<th>PEG</th>
<th>protean operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>parsing rule</td>
<td>A ← e</td>
</tr>
<tr>
<td>terminal</td>
<td>a</td>
</tr>
<tr>
<td>non-terminal</td>
<td>T</td>
</tr>
<tr>
<td>empty string</td>
<td>ε</td>
</tr>
<tr>
<td>sequence</td>
<td>( e_1 ; e_2 )</td>
</tr>
<tr>
<td>ordered-choice</td>
<td>( e_1</td>
</tr>
</tbody>
</table>

Table 2. The translation from PEGs to protean operators

\[
\begin{align*}
[\_; Regex \; \_+^*] & : Regex \\
[\_; Regex \; \_+^\*] & : Regex \\
Expr<Regex> & → Expr<Regex> \; "^+^*" \\
| 1 | Expr<Regex> & "^+^*"
\end{align*}
\]

Figure 7. The definition of the regular expression literals without operator precedence or associativity (the translation from Figure 4)

Note that the ordered choice \( | \) chooses the left operand first and then the right operand. So the operator with a higher precedence is the left operand.

On the other hand, any PEG rule can be translated into protean operators. Table 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 \( op_2 \). Terminal symbols in PEG are translated into an operator-name of protean operator. Non-terminal symbols at the left-hand side of \( \rightarrow \) are translated into the return types while non-terminal symbols at the right-hand side are translated into the operand types. The left and right operands of an ordered choice are translated into distinct two protean operators. The operator for the left has a higher parsing precedence than the operator for the right.

Operator precedence and associativity

We show below how to translate the protean operators with operator precedence and associativity into the protean operators without them. Assume that operator precedence is represented by a non-negative integer number and the larger number indicates the higher precedence. We show the translation from the protean operator \( [S] : t \) having operator precedence \( P \) and associativity \( A \). The operator syntax \( S \) involves \( n \) operands and each operand has the type \( T^i \). First, the return type \( T \) is translated into the type \( T_P \). Here, the subscript \( P \) is a non-negative integer number that is equivalent to the operator precedence. Second, each operand \( S^i : T^i \) in the operator syntax \( S \) is translated into the operand \( S^i : T_{P+1} \) if the operand is not the left-or-right-most element in the syntax. The left-most operand \( S^i : T^1 \) is translated into \( S^i : T_{P+1} \) if the operator associativity \( A \) is left-assoc. Otherwise, it is translated into \( S^i : T_{P+1} \) like the other operand. The right-most operand \( S^i : T^n \) is also translated into \( S^i : T_{P+1} \) if the operator associativity \( A \) is right-assoc. Otherwise, it is translated into \( S^i : T_{P+1} \). For example, the following operator:

\[
[\_; Regex \_; Regex \_; Regex \{ left-assoc \}] : Regex
\]

with the operator precedence 0, is translated into:

\[
[\_; Regex \_; Regex \_; Regex] : Regex_0
\]

Then we add an additional operator \( [S; T_0] : T_{P-1} \) for each return type \( T_0 \) if \( P \) is not 0. This operator converts a given argument to the operand to a value of type \( T_{P-1} \) and returns it. Note that the parsing precedence of the added operator \( [S; T_0] : T_{P-1} \) is set to the lowest among the operators with the return type \( T_{P-1} \). Finally, we add the

4. Implementation: ProteaJ

We have developed ProteaJ, which is a subset language of Java and supports protean operators. ProteaJ recognizes a single character as a token. It enables protean operators to express user-defined literals. For convenience, a white space is recognized as a token separator by default, however, it can be recognized as a token by using a special keyword \texttt{reads}. ProteaJ provides a module system called operator modules to implement and export user-defined operators. Programmers can use these operators by importing the modules. 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 implemented the compiler of ProteaJ in Java. ProteaJ does not support generics since there is a problem when protean operators and generics use together (see section 3.1). ProteaJ also does not support inner classes because they make the compiler complicated. For the same reason, ProteaJ does not support annotations and the other facilities introduced in Java 1.5 or above.

4.1 Definitions of Protean Operators

The definitions of protean operators in ProteaJ are similar to the class and method definitions in Java. Figure 8 shows the definition of protean operators that express regular expressions. This code defines an operator module named \texttt{RegexOperators}. This module defines four protean operators. For example, the third one of them defines the greedy quantifier operator \( [\_; Regex \; ^*] : Regex \). The keyword \texttt{reads} 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 \texttt{reads} are mentioned later (see 4.2). ProteaJ next to \texttt{reads} 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 the operator-name of the operator. The parameter type of the operand \( r \) is described in the following part enclosed in parentheses (\texttt{Regex r}). It denotes that the type of the operand named \( r \) is \texttt{Regex}. The following : \texttt{priority} = 250 represents the operator precedence. The remaining part enclosed in curly braces is the operator body. It is equivalent to the method body of a method declaration.

\footnote{The keyword \texttt{reads} means that the parser \texttt{reads} the next input as an instance of a specified type.}
operators RegexOperators {
  readas Regex rs* (Regex... rs): priority = 200 {
    return new RegexList(rs);
  }
  readas Regex r "*+" (Regex r): priority = 250 {
    return new RegexPlus(r);
  }
  readas Regex r "*" (Regex r): priority = 250 {
    return new RegexPossessivePlus(r);
  }
  readas Regex l (Letter l): priority = 300 {
    return new Regex(l);
  }
}

Figure 8. The definition of protean operators expressing regular expressions

Figure 9 is the syntax of the declarations of protean operators 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 of a declaration 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 operator associativity is left-associative. ProteaJ provides several notations like PEG notations for describing the syntax of the operator. For example, the second operator in Figure 8 is an example using operators more concisely.

readas are called expression operators. Readas operators must be expressions of readas operators. Readas operators are mainly used for defining literals. Readas operators are inconvenient for user-defined expressions since token separators are automatically recognized as a separator of tokens. On the other hand, when parsing a readas operator, a white space is recognized as a separator. ProteaJ allows programmers to define an operator returning multiple arguments. Multiple operator modules can be imported in one source file by writing multiple using-declarations. For example, GrepOperators, RegexOperators, and FilePathOperators are used together in the following code:

```java
using Regex Operators;
using FilePath Operators;
using Grep Operators;

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

4.2 Readas Operators, Operator Precedence, Parsing Precedence

In ProteaJ, protean operators can be divided into two categories: expression operators and readas operators, which begins with 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. The operands of readas operators must be expressions of readas operators. Readas operators are mainly used for defining literals. Readas operators are inconvenient for user-defined expressions since token separators must be explicitly inserted into the definition of the syntax.

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.

The operator precedence of protean operators are specified by integer values. In ProteaJ, the value of a precedence is larger, the binding of an operator is tighter. For example, the third operator in Figure 8 has higher parsing precedence than the first operator in the figure.

```
Figure 9. The syntax of the protean operator declarations in ProteaJ
```

Parsing precedence of protean operators are specified by the order of definitions in ProteaJ. The precedence of an operator defined earlier is higher. For example, the second operator in Figure 8 has higher parsing precedence than the third operator.

Operator precedence and parsing precedence are closed in each operator module. The entire operator precedence and parsing precedence are finally determined by the order of using-declarations. Operators in a module that is imported earlier have lower parsing precedence. Operators imported earlier binds tighter than operators imported later.

4.3 Case Study

The rest of this section, we show several internal DSLs implemented in ProteaJ.

Ruby-like print statement

In ProteaJ, programmers can define a new statement since ProteaJ allows programmers to define an operator returning void. Programmers can use such an operator as if an expression of the operator is a user-defined statement since a statement expression is
Another usage of protean operators is performance optimization.

Simple Optimization

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, Nat n): priority = 250
readas Regex \[[ ]+ \[ ]\] (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 10. regular expression literals as an internal DSL

considered as an expression that expects void type. The following code is a definition of an operator returning void:

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

and we can use this as follows:

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

In the above code, the last line is a statement expression. We can use p statement, which takes a string argument and prints the 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 10. 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 of line 4 in the above code. This regular expression literal consists of many operators: [ _+:Regex ]: Regex, [ _+:Regex | _+:Regex ]: Regex, [ _+:Regex | _+:Nat ]: Regex, [ _+:ClsElm ]: Regex, and so on. Parentheses ( _ ) are an operator provided by ProteaJ. They reset the parsing precedence and the operator precedence of the expression within them.

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 be more efficient, we should instead use the StringBuilder class. Protean operators in ProteaJ can be used in this case.

The definition in Figure 11 is the operators module that defines the optimized string concatenation. When the operators module is used, the single string concatenation such as "foo" + "bar" is interpreted as "foo".concat("bar"), but the successive string concatenation such as "foo" + "bar" + "baz" is interpreted as the following:

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

Like this, protean operators enables us to optimize the expressions that conform to the typical patterns. An important fact is that the optimizations are defined by the library, not the compiler.

SQL

In ProteaJ, programmers can implement more complex internal DSLs. For example, they can implement a subset of SQL. We implemented two operator modules, FilePathOperators and SQL Operators. FilePathOperators module enables us to write a file path like "/Documents/file.txt". The definition of FilePathOperators is shown in Figure 12. SQL Operators module defines some SQL operators, for example, select, create table, and insert into. The definitions of these operator modules are available from our web site.2With these modules, programmers can write a program shown in Figure 13, for example.

5. Experiment

We have conducted an experiment for demonstrating that ProteaJ can efficiently parse expressions including user-defined literals even though a naive parsing method such as SGLR cannot parse them in pragmatic time. We used JSGLR parser [2], that is a well-known implementation of a SGLR parser in Java, as a parser of a naive parsing method for mixfix operators supporting user-defined literals. Since the parser of ProteaJ cannot be detached from the compiler, we compared a compile time (parse time + code generation time) by ProteaJ and a parse time by JSGLR. The machine used for the experimentation had 2.67GHz Core i5 processor and 8 GB memory. The installed operating system on the machine was OpenSUSE 12.1. We used openJDK 1.7.0.

2The source code of ProteaJ and DSLs introduced in this section is available from: http://www.csg.ci.i.u-tokyo.ac.jp/~ichikawa/ProteaJ.tar.gz
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 14. ProteaJ uses the two operator modules in Figure 12 as the grammar.

- Input: a/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 15 shows the input source for ProteaJ.

- Measurement: an average parse or compile time of ten executions.
  The grammar shown in Figure 14 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/a/a explode exponentially. The two operator modules shown in Figure 12 express the same grammar as in Figure 14. When the two modules are imported by using-declarations, ProteaJ can parse any expressions that can be expressed by the grammar in Figure 14. Note that the grammar in Figure 12 is more powerful than Figure 14 since identifiers are not only a. We have measured the parse or compile time by changing the input size.

Figure 16 shows the result of the experiment. It is a semilog graph. The vertical axis is the parsing time, and the horizontal axis is the input size. The diamond is an average parse time by JSGLR and the rectangle is an average compilation time (parse time + code generation time) by ProteaJ. This graph is plotted for the input size 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.

The compilation time by ProteaJ increases linearly with the input size. Figure 17 presents the compilation time by ProteaJ and the input size. The vertical axis is the compilation time and the horizontal axis is the input size. This figure presents the same data as Figure 16 but on a different scale. The graph is plotted with the input size from 0 to 1000. The vertical axis of Figure 16 is on a logarithmic scale, but one of Figure 17 is on a linear scale.
using FilePathOperators;

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

Figure 15. The input source for the experiment of ProteaJ

![Graph showing comparison between ProteaJ compiler and JSGLR parser](image)

Figure 16. Comparison between ProteaJ compiler and JSGLR parser

![Graph showing the compilation time by ProteaJ](image)

Figure 17. The compilation time by ProteaJ

6. Related Work

The idea of this paper is initially published as ACM Student Research Competition [13]. The detailed discussion and the experiments are new materials of this paper.

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 [4], MetaML [17], Template Haskell, Nemerle [20], and Scala [3] support syntactic macros. They have the same drawback as Lisp macros.

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. Reader macros are very powerful, however, they are not composable. Multiple syntax definitions in different read 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 [18] and Converge [22] have the same facilities.

Nemerle also provides another macro system like C/C++ lexical macros. It allows programmers to define new syntax, and the semantics of the syntax can be defined by a compile-time meta-program. The restriction on the syntax is that the first token of the syntax must be unique. User-defined literals are difficult to implement in Nemerle since the syntax must begin with an identifier in the host language.

Mixfix Operators with Empty Syntax

Isabelle [16] and Maude [6] are programming languages supporting mixfix operators with empty syntax. The empty syntax support a nameless operator syntax like the protean operator \[_:Regex\_]\[\_:Regex\]_:Regex. Arbitrary Context Free Grammar can be expressed by mixfix operators with empty syntax. Although the mixfix operators with empty syntax have good expressiveness, they cannot express user-defined literals. A naive extension to them by using a scannerless parser is not practical due to the efficiency of the parsing as we mentioned.

External Tools

JastAdd [8] and Silver [24] are language construction systems based on attribute grammar [14]. 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 [5] 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 on 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). It is not suitable in our case.

Type-Oriented Island Parsing

Type-oriented island parsing [19] is a parsing algorithm based on island parsing [21], 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 the type-oriented island parsing can be applied to scannerless parsers since the type-oriented island parsing uses heuristics for parsing tokens.
7. Conclusion

In this paper, 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 the efficient parsing: overloading by return type and 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 the overloading by return type. Since these features resolve all the grammar ambiguities at parse time, protean operators 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 considering type information. A drawback of protean operators is a limited kind of places where the operators are available. Protean operators are available only in the expressions whose expected type is statically determined before parsing the expression.

We have developed ProteaJ, which is a subset language of Java and supports protean operators. ProteaJ provides a module system called operator module to implement and modularize user-defined operators. We implemented the compiler of ProteaJ in Java. It is available from our web site mentioned in section 4.3. We have conducted an experiment for demonstrating that ProteaJ can efficiently parse expressions including user-defined literals even though a naive parsing method such as SGLR cannot parse them in pragmatic time. Currently, the entire operator precedence and parsing precedence are determined by the order of using-declarations; however, it is not clear that this means resolve conflicting operators in any case. To find better composable precedence rules is future work.

References