Modular and Extensible Parser Implementation using Mixins (DRAFT)

Yuuji ICHISUGI
Electrotechnical Laboratory

March 1, 1999

Abstract

This paper describes a method to construct highly modular and extensible recursive descent parser. This parser is used in an extensible Java preprocessor, EPP. EPP can be extended by adding plug-ins which extend Java syntax and add new language features. The EPP's parser consists of small mixins. A recursive descent parser class is constructed by composing these mixins. The syntax accepted by the parser can be extended by adding new mixins.

1 Introduction

The author has developed an extensible preprocessor for Java[GJS96], EPP[IR97, Ich] which can be extended by adding new modules which extend Java syntax, and added new language features. This paper describes a method of constructing a highly modular and extensible recursive descent parser, which is the parser of EPP.

In designing EPP, we aimed at a wide-range of extensibility, highly flexible implementation of extension modules, and simultaneous usability of multiple extension features - hereafter called composability.

Traditionally, extension of a system requires editing and direct modification of the source code. This has been considered the best method of realizing the highest extensibility and flexibility in implementation. With this method, however, it is difficult to simultaneously use more than one independent extension: the user cannot achieve composability.

Many extensible languages define a new syntax and its semantics with a declarative description. While this realizes high composability, descriptiveness of a descriptive language limits extensibility and flexibility.

EPP realizes as high extensibility as editing source code and realizes as high composability as declarative description. EPP's parser was implemented in such a way as to treat its extension modules as ordinary general-purpose programming language modules, using mixins, the feature of programming by difference. The parser has the following features:

Plug-ins which define new syntax, operators and others can be added afterward. Because syntax extension plug-ins are programmed by difference, multiple plug-ins can be combined at the same time.

Because of high modularity, plug-ins can be separately compiled one by one. Syntax extension plug-ins can be distributed without source codes.

Because of its general-purpose language description, it is easy to perform such ad-hoc processes as context sensibility and global escape, which are rather awkward with the BNF method.

Error recovery and line number managing can be implemented.

The EPP's architecture is applicable to any other programming languages if the implementation language has symbol and mixin features.

This paper consists of seven more sections. Section2 outlines EPP, Section3 presents symbols and mixins-language features necessary in parser implementation, Sections4 and 5 explain how to describe the extensible parser using mixins and to implement ad-hoc processes, Section6 evaluates the parser implementation, Section7 describes related studies, and Section8 is the conclusion.

2 Outline of Extensible Java Pre-processor EPP

EPP is an extensible Java source-to-source preprocessor which can introduce new language features. The user can specify EPP plug-ins at the top of the Java source code by writing `#epp name` in order to incorporate various extensions of Java. Multiple plug-ins can be retrieved simultaneously as

long as they do not collide with each other. Emitted source codes can be compiled by ordinary Java compilers and debugged by ordinary Java debuggers.

EPP works not only as an extensible Java preprocessor but also as a language-experimenting tool for language researchers; a framework for extensible Java implementation; and a framework for a Java source code parser/translator.

The EPP's source code is written in Java extended by EPP itself, and it was bootstrapped by EPP written in Common Lisp[Ste90]. The byte-code is available in any platform where Java is supported.

3 symbol and mixins

This section describes symbols and mixins, which are the language features necessary in implementing the extensible parser.

3.1 Symbol Implementation on Java

Symbols, a data-type as in languages like Lisp, have the following features:

Symbols are similar to constants defined by C language's `enum` statement. Unlike C constants, however, the user can use symbols having arbitrary names (strings) as required in the source codes, without defining a finite number of elements beforehand.

Symbols are similar to string literals, but unlike strings, the mere pointer comparison efficiently judges equality of symbols.

By specifying a name, a symbol can be generated dynamically.

Fig.1 shows a program using the Symbol plug-in. A Symbol literal is expressed by a colons followed by an identifier or a string literal.

The Symbol plug-in implements symbols on Java as follows: The symbol literals in the program are translated into references to each individual private static final variable. The variable is initialized by the return value of a static method invocation : `Symbol.intern("name")`. `Symbol.Intern("name")` looks up a hashtable and returns an instance of Symbol class having the specified name if it already exists. If it does not, a new instance is generated, registered in the table and returned. As a result, symbol literals having the same name are guaranteed to reference the instances having the same identity. The execution does not produce the overhead of hashtable retrieval because only a static constant value is referenced.

```
#epp jp.go.etl.epp.Symbol
import jp.go.etl.epp.epp.Symbol;

public class TestSymbol {
  public static void main(String args[]){
    Symbol x = :foo;
    Symbol y = :"+";
    System.out.println(x == :foo); // true
    System.out.println(y == :foo); // false
  }
}
```

Figure 1: A program using Symbol plug-in

3.2 Mixin Implementation on Java

3.2.1 What is mixins?

Usually, in an object-oriented language, a particular super class name is specified when defining a subclass; a mixin is a subclass defined with no particular super class specified. A mixin is defined on the presumption that multiple inheritance and linearization by another class will determine a super class afterward.

Bracha[BC90] showed that the mixin mechanism can simulate the same inheritance mechanisms as SmallTalk, BETA and CLOS. VanHilst[VN96] suggested a variation of mixin-based inheritance which enhances reusability of object-oriented programs.

C++ provides multiple inheritance, but does not linearize super classes. Therefore, it is impossible to do mixin-based inheritance with the C++'s multiple inheritance mechanism. A class which does not make a shared super class by multiple inheritance is sometimes called a mixin by C++ programmers, however, it should not be confused with the mixin used in this paper.

A mixin is largely similar to decorator pattern in Design Pattern[GRV95], with two differences: (1) a mixin does not allow modules to be exchanged during execution and (2) a mixin enables a new method to be added to a class, while a decorator can only have pre-fixed interface.

3.2.2 Program example using mixin

EPP's "SystemMixin plug-in" provides mixin-based inheritance [1]. The following describes the mixin

[1] Unlike intrinsic mixins, SystemMixin plug-ins actually provide programming by difference in the entire system consists of multiple classes, not in each class. However, this paper regards SystemMixins and intrinsic mixins as the same thing because EPP's parser consists of only one class.

```
class Foo {
  void m(char c){
    if (c == 'B') {
      doB();
    } else if (c == 'A') {
      doA();
    } else {
      doDefault();
    }
  }
}
```

Figure 2: A method definition which uses nested if statements.

features using the syntax extended by SystemMixin plug-in. First, Fig.2 shows a common method definition using nested `if` statements.

Next, Fig.3 shows the method definition divided into three mixins. Here, the method-invocation expression, `original`, introduced by SystemMixin plug-in, corresponds to the super method-invocation in traditional object-oriented languages.

Thus, mixins enable the user to divide a method, which used to be an inseparable unit, into multiple "method fragments"; and afterward the whole class can be constructed by combining multiple mixins.

3.2.3 Mixin implementation

EPP's SystemMixin plug-in implements mixins by translating all the method fragments incorporated in mixins into small Java classes. The method-invocation expression searches the hashtable of the receiver object for the object to implement a method segment and is translated into an expression to invocate its nested `call`. The problem with this implementation is that, with low efficiency, it is impossible to treat the mixin-defined class and the intrinsic Java class equally. However, this implementation is adopted because it allows separate compilation of the mixins.

C++ can also implement mixin-based inheritance by parameterizing the super class using the template mechanism. Fig.4 shows mixins defined by the template. Combining Skeletons A and B provides the class B¡A¡Skeleton¿¿. Note that mixins defined by the template do not allow separate compilation.

3.3 Mixins Composing EPP

The EPP's parser is defined as the only class named `Epp`, with the class definition divided into multiple

```
SystemMixin Skeleton {
  class Foo {
    define void m(char c){
      doDefault();
    }
  }
}
SystemMixin A {
  class Foo {
    void m(char c){
      if (c == 'A') { doA(); }
      else { original(c); }
    }
  }
}
SystemMixin B {
  class Foo {
    void m(char c){
      if (c == 'B') { doB(); }
      else { original(c); }
    }
  }
}
```

Figure 3: A method definition by mixins.

```
class Skeleton {
public:
  void m(char c){ doDefault(); }
};
template<class Super>
class A : public Super {
public:
  void m(char c){
    if (c == 'A') { doA(); }
    else { Super::m(c); }
  }
};
template<class Super>
class B : public Super {
public:
  void m(char c){
    if (c == 'B') { doB(); }
    else { Super::m(c); }
  }
};
```

Figure 4: Mixins defined by the template mechanism of C++.

mixins. Starting EPP combines all the mixins composing the standard Java parser and mixins composing the plug-in specified at the top of the source code to construct one parser (the class named `Epp`). EPP then generates the class instance and invokes the starting method to begin processing the input source code.

4 Implementation of extensible parser

4.1 Representation of tokens

One problem about extensible lexical analyzer is how the programmer extends the definition of data type returned by the lexical analyzer. Two possible solutions are as follows:

To provide the means of extending the token data type definition along with the means of extending the lexical analyzer.

To provide a general purpose data type in advance for all possible tokens.

Possibility (1) will require modification and recompilation of the source code of the parser because the lexical analyzer extension changes the data type, affecting the parser which processes the data type. Therefore, EPP implements possibility (2). The EPP's token data types, constructed so as to handle a wide range of extension, apply to almost any language extension without modifying data types. More specifically, all the tokens are expressed either in *literal* data types or *symbol* data types.

A literal is composed of a tag representing the kind of the literal and a string representing the content of the literal. For instance, an integer literal 123 is expressed as a tag `int` and a string `123`. In this way, a literal which was not included in the original syntax can be expressed with a new tag assigned.

All the tokens except literals-identifiers, key words such as `if` and `while`, operators, and special characters such as semicolons and parentheses-are expressed as symbols. EPP does not distinguish between keywords and identifiers; therefore, a new syntax can easily be added by simply extending the parser without modifying the lexical analyzer at all.

4.2 Recursive Descent Parser

This chapter describes the conventional implementation of the method of parsing non-terminals without using mixins. (Chapter 4.3 describes the implementation split into mixins.)

The following is an example of the production with alternatives of a prefix operator, parentheses, a right associative binary operator, a left associative binary operator, and a postfix operator [2].

```
Exp → ++ Exp | ( Exp ) | Term += Exp | Term
        | Exp + Term | Exp ++
```

Rewriting this production provides the form that can be parsed by the recursive descent parser: LL(1) grammar. (See the appendix for the details.) The recursive descent parser[ASU87] consists of functions which parse corresponding non-terminal and return the parsed abstract syntax trees.

Fig.5 shows a part of a recursive descent parser (without mixins) for the non-terminal `Exp` in the example production, where the three methods return abstract syntax trees as follows:

- `expTop` parses alternatives that are neither right recursion nor left recursion. For example, a prefix operator or parentheses.
- `expRight` parses alternative of right associative operators.
- `expLeft` parses alternatives that are left recursion. For example, postfix operators or left associative operators.

The roles of methods invoked from the program are as follows:

- `lookahead` returns the token currently being noticed.
- `match` reports an error if the current token differs from the argument value. Otherwise, it discards the current token and reads the next token.
- `matchAny` discards the current token unconditionally.

The program generates the abstract syntax trees as is expected from the production. For example, a += b += c generates (+= a (+= b c)); a + b + c generates (+ (+ a b) c).

4.3 Extensible Recursive Descent Parser

[2] In fact, in general language grammar, a non-terminal never mingles alternatives of parentheses and a binary operator; or a right associative binary operator and a left associative binary operator. Such grammar cannot generate expressions like `a + (b)` or `a + b += c`, and then, the grammar conflicts with human intuition.

```
Tree exp() {
  Tree tree = expTop();
  while (true){
    Tree newTree = expLeft(tree);
    if (newTree == null) break;
    tree = newTree;
  }
  return tree;
}
Tree expTop() {
  if (lookahead() == :"++"){
    matchAny();
    return new Tree(:"preInc", exp());
  } else if (lookahead() == :"("){
    matchAny();
    Tree e = exp();
    match(:")");
    return new Tree(:"paren", e);
  } else {
    return expRight(exp1());
  }
}
Tree expRight(Tree tree) {
  if (lookahead() == :"+="){
    matchAny();
    return new Tree(:"+=",tree,exp());
  } else {
    return tree;
  }
}
Tree expLeft(Tree tree) {
  if (lookahead() == :"+"){
    matchAny();
    return new Tree(:"+",tree,exp1());
  } else if (lookahead() == :"++"){
    matchAny();
    return new Tree(:"postInc", tree);
  } else {
    return null;
  }
}
Tree exp1() { return term(); }
```

Figure 5: A part of a recursive descent parser.

```
SystemMixin Exp {
  class Epp {
    define Tree exp(){
      Tree tree = expTop();
      while (true){
        Tree newTree = expLeft(tree);
        if (newTree == null) break;
        tree = newTree;
      }
      return tree;
    }
    define Tree expTop(){
      return expRight(exp1()); }
    define Tree expRight(Tree tree){
      return tree; }
    define Tree expLeft(Tree tree){
      return null; }
    define Tree exp1(){
      return term(); }
  }
}
```

Figure 6: A skeleton of a extensible parser.

```
SystemMixin Plus {
  class Epp {
    Tree expLeft(Tree tree) {
      if (lookahead() == :"+") {
        matchAny();
        return new Tree(:"+",tree,exp1());
      } else {
        return original(tree);
      }
    }
  }
}
```

Figure 7: A mixin which defines a left associative binary operator.

Splitting the program shown in Fig.5 into mixins makes it more modular and extensible. Removing *if-then* clauses and leaving *else* clauses in Fig.5 provides a skeleton as shown in Fig.6. `Exp` defined by the mixin is a method of parsing the following production:

`Exp` → `Term`

New alternatives can be added to non-terminals by extending the methods expTop, expRight and expLeft in this program using mixins. Fig.7 shows a mixin which defines a left associative binary operator.

EPP defines dozens of kinds of non-terminals as a set of the mixins which define the skeleton as shown in Fig.6 and the mixins which add alternatives as shown in Fig.7. With a macro facilitating these definitions, the mixins in Fig.6 can be defined by the following one line:

```
defineNonTerminal(exp, term());
```

Also, the mixins which add the left associative binary operator in Fig. 7 can be defined by the following one line:

```
defineBinaryOperator(Plus, :"+", exp);
```

4.4 Lexical Analyzer Extension by Difference

A recursive descent lexical analyzer is extensible by difference. The EPP's lexical analyzer mainly consists of the following methods, whose behavior can be extended by mixins.

```
readToken
readId
readNumber
readOperator
readStringLiteral
readCharLiteral
readTraditionalComment
readEndOfLineComment
```

For example, Fig.8 shows the mixin that does not regard `//` as a beginning of a comment if it is followed by `:`. (This feature preserves extensibility and compatibility with Java. This is an example of a program using the mixin.

```
//: assert(predicate);
```

This line is simply regarded as a comment by the standard Java compiler, but works as an assert macro if the file is processed by EPP.)

```
SystemMixin CommentPragma {
  class Epp {
    Token readEndOfLineComment
                (EppInputStream in){
      if (in.peekc() == ':'){
        in.getc();
        return readToken(in);
      } else {
        return original(in);
      }
    }
  }
}
```

Figure 8: A mixin which extends the lexical analyzer.

4.5 Parser Module Deletion and Redefinition

EPP also provides a means of extending grammar other than programming by difference.

Grammar extension by programming by difference is somewhat limited in that (1) only new parser module addition is possible; current module deletion is impossible and (2) extension works only with prepared "hooks" (methods).

One way to perform extension without programming by difference is not to execute the original method invocation in adding mixins: i.e., to disregard the `original` method. For example, redefining exp1 in Fig.6 modifies the precedence of operators.

In addition to this, plug-ins have a mean of removing some parser definition modules (mixin) when plug-ins are loaded. By the mean, the grammar can be arbitrary modified.

The drawback of realizing the plug-ins through the above two means is that they are much less composable than extension only by programming by difference. A plug-in programmer/user has to trade-off extensibility against composability.

5 Ad-hoc Processes

5.1 Backtrack

EPP provides explicit backtrack with the lexical analyzer and the parser. Fig.9 shows a mixin which defines a new token "`**`" .

The argument EppInputStream is an input stream that backtracks at an arbitrary length by having the whole input file as a character array on the memory.

```
SystemMixin NewOp {
  class Epp {
    Token readOperator(EppInputStream in){
      if (in.peekc() == '*') {
        int p = in.pointer();
        in.getc();
        if (in.getc() == '*') {
           return :"**";
        }
        in.backtrack(p);
        return original(in);
      } else {
        return original(in);
      }
    }
  }
}
```

Figure 9: A mixin which defines a new token.

5.2 Context Sensitivity

A recursive descent parser easily implements context-sensitive processes in the lexical analyzer and the parser: the user just has to input the context information into the global variables (static variables, in Java terminology).

5.3 Error Recovery

Error recovery during parsing is easily implemented by Java's exception handling feature. The error handler just has to skip tokens till a particular token appears.

5.4 Line Number Managing

It is desirable to have information on "the line number at which the syntax began" in the Tree that is generated by parsing. The information helps generate clear error messages when errors occur during semantic analysis afterwards. EPP also uses the information to output each line of the source code at the same line after translation.

Lisp and Java easily implement line number managing. Lisp uses variables with dynamic scope, and Java uses static variables, stacks and try-finally syntax.

Fig.10 shows an example definition of the method exp with Java. All the non-terminal methods are defined in the same way. The Tree constructor obtains the line number at which it began by checking

```
Tree exp(){
  LineNumber.stack
    .push(currentLineNumber());
  try {
    ... Same as Fig.6 ...
  } finally {
    LineNumber.stack.pop();
  }
}
```

Figure 10: Managing line number information.

the top of the stack of the static variable LineNumber.stack.

6 Evaluation

6.1 Java Grammar Description

EPP incorporates a complete Java parser for JDK1.1 implemented as described in this paper. The Java grammar definition part consists of 105 mixins, 29 of which define the skeleton of non-terminals as shown in Fig.6.

Explicit backtrack was executed at the following points during implementation:

1. Distinction between constructor and method or field.
2. Distinction between static method/field and static initializer.
3. Distinction between method and field.
4. Distinction between local variable declaration and statement.

Some of the above can be parsed with LL(1) by rewriting grammar, but that lowers extensibility and modularity. Therefore, backtrack was adopted.

Field access syntax and cast syntax are implemented with lower modularity; it is impossible to make these elements highly modular unless type information is obtained during parsing.

6.2 Efficiency

The author tested the speed of EPP for processing of source code of EPP itself, consisting of 7218 lines, and obtained the following results.

- MMX Pentium 233MHz, Windows95, Microsoft SDK2.01: approximately 30 seconds

- UltraSPARC 200MHz, Solaris2.5.1, JDK1.1.3: approximately 40 seconds

The process consists of three parts: source code parsing, macro expansion of extended syntax, and source code emission after translation. The most time-consuming part is parsing. That is no problem for practical use, but a parser should work much faster. Three factors cause EPP's low speed:

1. Java interpreter overhead,
2. Mixin method invocation overhead, and
3. intrinsic low speed of parser description methods proposed by this paper (e.g., sequential search of alternatives with if-then-else and unnecessary invocation of method having no content).

The overhead of the current mixin method invocation is approximately 10 times slower than that of the standard Java class invocation. This is because EPP implements a mixin-defined fragment with a small Java object, invoking the method by searching the hashtable during execution. Mixin implementation speed should be improved in the future.

The overhead of sequentially searching for alternatives with if-then-else takes up much time. In order to improve that, one possible solution is to implement an optimized translator that is specialized to the EPP's parser source code. For example, if-then-else should be translated into a table search.

In general, having too many backtracks reduces parsing efficiency, but Java grammar never causes backtracks that will seriously reduce the speed, as proven by the following experiment. When the source code of Java.uil.Vector in JDK1.1.1 was parsed by EPP, 1214 tokens were appeared. There were 86 backtracks which caused 172 times extra invocation of `readTokens` method. The result shows that the invocation of `readToken` increases approximately 14

6.3 Ease of Debugging

Standard declarative description parser generators detect collisions and ambiguity of grammar to warn the user. Unfortunately, EPP's parser does not have this feature. Since it is intended for extending the grammar of already completed languages by difference, the parser is not intended as a tool for designing a new language grammar.

Nevertheless, it would be possible to construct a parser generator which generates mixins described in this paper.

Local debugging of a recursive descent parser is easy; the standard Java debugger and print statements work as in common programs.

7 Related work

ANTLR[ANT] and JavaCC[Jav] are recent top-down parser generators based on LL(k). According to the creators of these tools, the advantages of a top-down parser include ease of debugging and passing attribute values downward or upward during parsing. A bottom-up parser like LALR(1) does not have such features. Also, ANTLR can extend the existing grammar by difference through inheritance. JavaCC enables direct writing in part of the production, making writing easy with declarative description.

MPC++[Ish94], OpenC++[Chi95], JTRANS[KK97] and OpenJava[Tat] are extensible systems which can introduce new language features by providing compile time MOP (Meta object Protocol[KdRB91]) during compilation. Like EPP, their task is to perform complicated translation on an abstract syntax tree after parsing. Also, the grammar is extensible in a limited range. For example, MPC++ allows addition of new operators and statements.

Eli[GHL+92] is a compiler-generator which modularizes the grammar definition. It automatically generates a language processor using grammar and semantics definitions based on attribute grammar, and defines a new language by a kind of inheritance using existing definition modules.

Many "extensible languages" for grammar modification have been created, and most of them, including Lisp and C macros, define new grammar extension in the *on-the-fly* style, i.e. in the program to be processed. The problems with on-the-fly extension are that (1) it does not allow pre-compilation of the grammar extension code and therefore lacks efficiency, and (2) modifying or extending "the syntax to define syntax extension" itself often causes confusion. EPP has no such problems because it does not work in the on-the-fly style.

Camlp4[Rau] is an Objective Campl pre-processor whose grammar can be extended by adding modules. The extension can be done by difference with declarative description, and the modules can be compiled separately.

8 Conclusion

A method of constructing a highly modular and extensible parser by splitting the recursive descent

parser into small mixins was described. The syntax accepted by the parser can be extended with high composability over a wide range by adding mixins implemented by programming by difference. Also, in principle, removing existing mixins arbitrarily modifies grammar.

acknowledgment

The author wishes to thank Makoto Matsushita of Osaka University for his discussion about the method of describing an extensible parser in the early stages of the study, and Yves Rouder who was a visiting researcher of the Electrotechnical Laboratory for his helpful suggestions on EPP.

References

[ANT] Antlr home page. "http://www.ANTLR.org/".

[ASU87] A.V. Aho, R. Sethi, and J.D. Ullmann. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1987.

[BC90] G. Bracha and W. Cook. Mixin-based Inheritance. In *Proc. of E-COOP/OOPSLA'90*, 1990.

[Chi95] S. Chiba. A Metaobject Protocol for C++. In *Proceedings of OOPSLA'95*, volume 30(10) of *ACM Sigplan Notices*, pages 285–299, Austin, Texas, October 1995. ACM Press. "http://www.softlab.is.tsukuba.ac.jp /~ chiba/openc++.html".

[GHL+92] R.W. Gray, V.P. Heuring, S.P. Levi, A.M. Sloane, and W.M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–131, February 1992.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[GRV95] Helm Gamma, E., R. R., Johnson, and J. Vlissides. *Design Patterns.* Addison-Welsley, 1995.

[Ich] Y. Ichisugi. EPP home page. "http://www.etl.go.jp/~ epp".

[IR97] Y. Ichisugi and Yves Roudier. The extensible java preprocessor kit and a tiny data-parallel java. In *ISCOPE'97, California*, LNCS 1343, pages 153–160, December 1997.

[Ish94] Y. Ishikawa. Meta-level Architecture for Extendable C++, Draft Document. Technical Report Technical Report TR-94024, Real World Computing Partnership, 1994. "http://www.rwcp.or.jp/lab/mpslab/mpc++ /mpc++.html".

[Jav] Javacc home page. "http://www.suntest.com/JavaCC/".

[KdRB91] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of Metaobject Protocol.* MIT Press, 1991.

[KK97] A. Kumeta and M. Komuro. Meta - Programming Framework for Java. In *The 12th workshop of object oriented computing WOOC'97, Japan Society of Software Science and Technology*, March 1997.

[Rau] D. Rauglaudre. Camlp4 home page. "http://pauillac.inria.fr/camlp4/".

[Ste90] G.L. Steele. *Common Lisp the Language, 2nd edition.* Digital Press, 1990.

[Tat] M. Tatsubori. Open java home page. "http:// www.softlab.is.tsukuba.ac.jp /~ mich/openjava/".

[VN96] M. VanHilst and D. Notkin. Using role components to implement collaboration-based designs. In *OOSPLA'96*, October 1996.

A Production Rewrite

The production defined as follow is rewritten so that it can be parsed by the recursive descent parser.

```
Exp → ++ Exp | ( Exp ) | Term += Exp | Term
      | Exp + Term | Exp ++
```

Split the production into two by introducing `ExpTop`.

```
Exp → ExpTop | Exp + Term | Exp ++
ExpTop → ++ Exp | ( Exp ) | Term += Exp | Term
```

Remove the left recursion of `Exp` by introducing `ExpLoop` and rewrite `ExpTop` by introducing `ExpRight`.

```
Exp → ExpTop ExpLoop
ExpTop → ++ Exp | ( Exp ) | Term ExpRight
ExpRight → += Exp | ε
ExpLoop → + Term ExpLoop | ++ ExpLoop | ε
```

Now this grammar can be parsed by recursive descent parser. Furthermore, by introducing `ExpLeft`, `ExpLoop` can be rewritten as follows:

```
Exp → ExpTop ExpLoop
ExpTop → ++ Exp | ( Exp ) | Term ExpRight
ExpRight → += Exp | ε
ExpLoop → ExpLeft ExpLoop | ε
ExpLeft → + Term | ++
```

Fig.5 shows a part of a recursive descent parser for non-terminals based on the above productions. Note that the tail recursion of ExpLoop is rewritten into Loop and embedded in the method `exp`; the method expLeft expresses returns a special value, null, if no matching alternatives.

Both the original and rewritten grammars are ambiguous. For example, the expression `++ a + 1` can be interpreted as either `(++ (+ a 1))` or `(+(++ a )1)` . The program in Fig. 5 parses this as `(+(++ a )1)`.