Extensible Java Pre-processor

Electrotechnical Laboratory

Yuuji Ichisugi

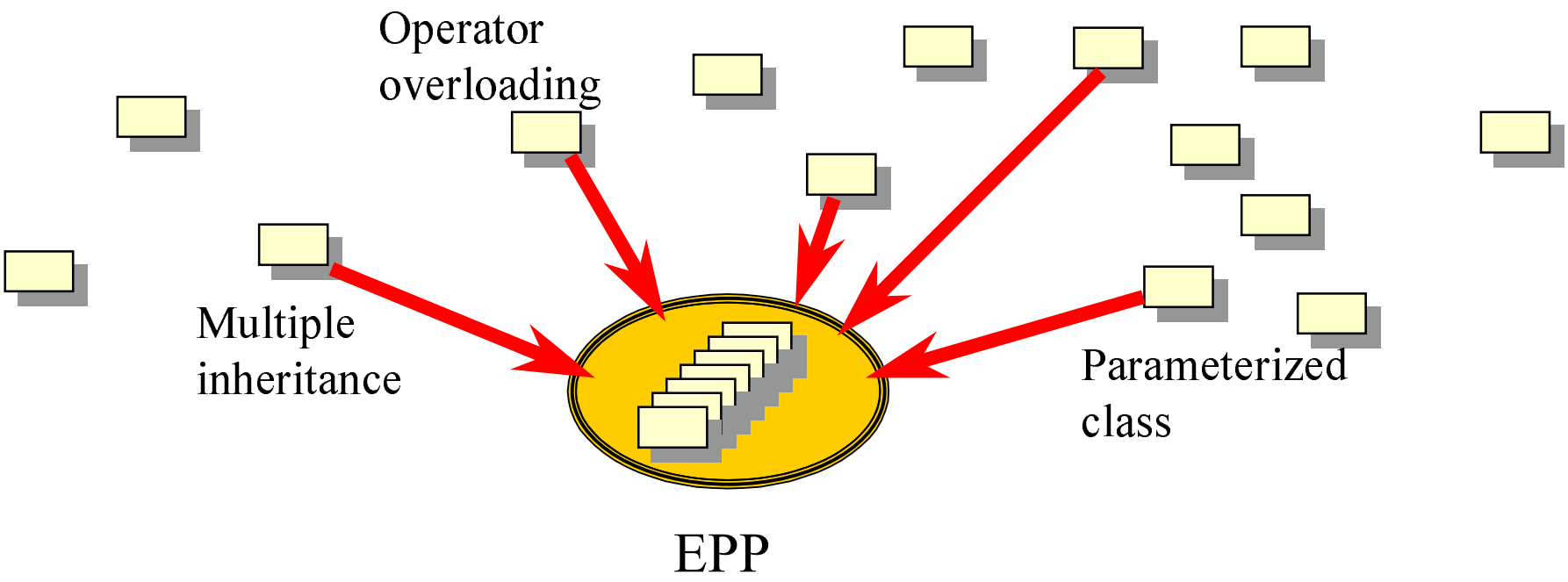
http://www.etl.go.jp/~epp/

Contents

- What is EPP ?
- Examples
- Mixin
- Extensible recursive-descent parser
- Extensible type checking system
- Conclusion

What is EPP ?

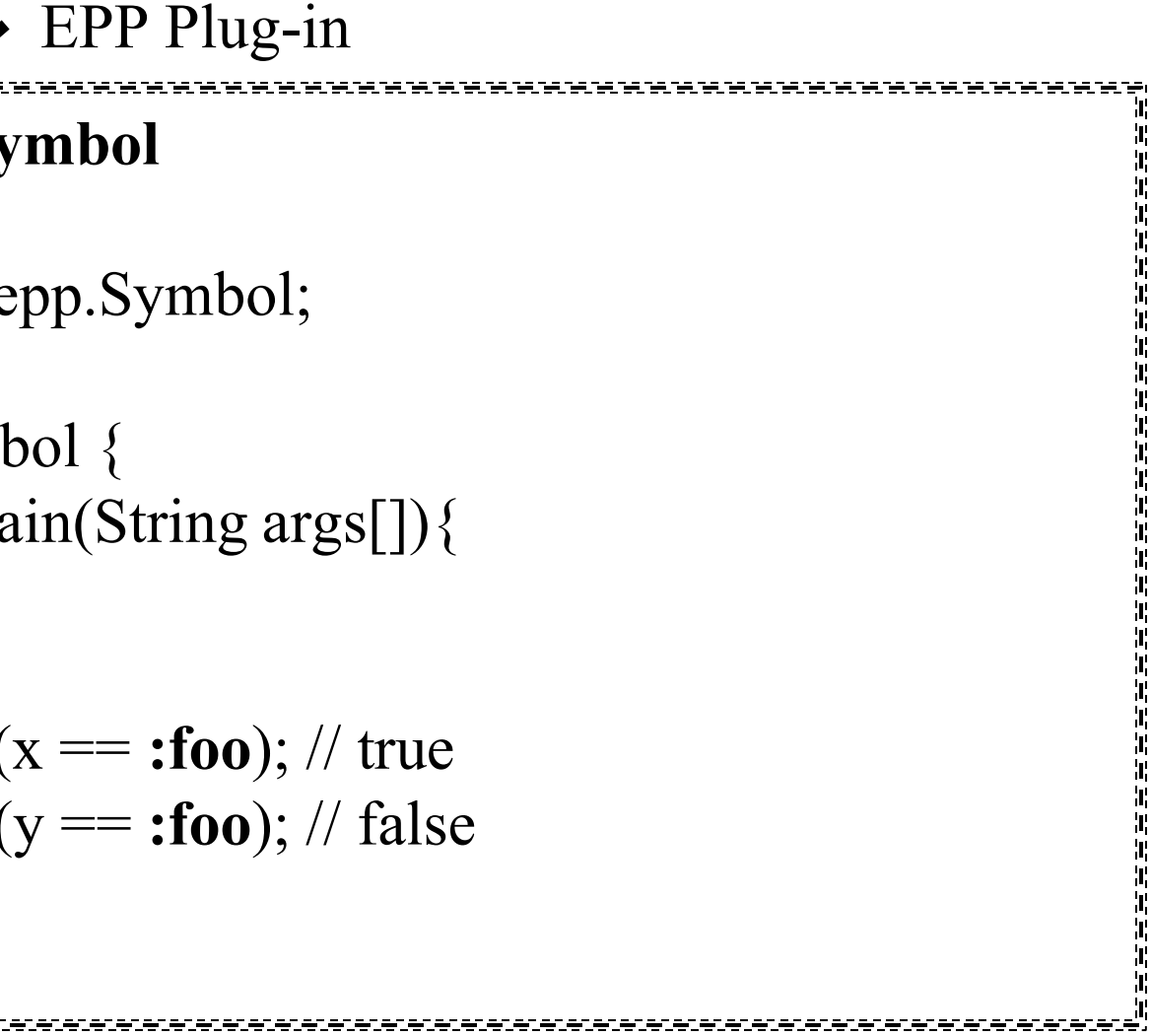
- A pre-processor which can be extended by adding “plug-ins”.



World-wide programming language

- EPP will produce various programming languages required for world-wide programming.
- Example: **Mobile agent system**
  - Collaboration with Tsukuba University.
  - Thread migration implementation on pure JavaVM.
- Future plan: **Security enhancement** for Java
  - taint Java, proof-carrying code for Java, …?

Example : Symbol plug-in



Translated program

```
/* Generated by EPP 1.0.2beta (by lisp-epp1.0.2beta) */
import jp.go.etl.epp.epp.Symbol;
public class TestSymbol {
  private static final Symbol _Sym0 = Symbol.intern("foo");
  private static final Symbol _Sym1 = Symbol.intern("+");
  private static final Symbol _Sym2 = Symbol.intern("foo");
  private static final Symbol _Sym3 = Symbol.intern("foo");
  public static void main(String args[]){
    Symbol x = _Sym0;
    Symbol y = _Sym1;
    System.out.println((x) == (_Sym2));
    System.out.println((y) == (_Sym3));
    }
  }
```

Actually, all line numbers are preserved by translation in order to support debugging.

Source code of Symbol plug-in

```
#epp jp.go.etl.epp.Symbol
#epp jp.go.etl.epp.SystemMixin
#epp jp.go.etl.epp.AutoSplitFiles
#epp jp.go.etl.epp.BackQuote
#epp jp.go.etl.epp.EppMacros

package jp.go.etl.epp.epp.Symbol;

defineNonTerminal(symbol, symbolOtherwise());
SystemMixin SymbolRep {
  class Epp {
    extend Tree primaryTop() {
      if (lookahead() == ':') {
        return symbol();
      } else {
        return original();
      }
    }
    extend Tree symbolTop() {
      if (lookahead() == ':') {
        matchAny();
        Token next = matchAny();
        if (next.isSymbol()){
          return new Tree(:symbol, new Identifier((Symbol)next));
        } else if (next.isLiteralToken()){
           String contents = next.literalContents();
          return new Tree(:symbol, new Identifier(Symbol.intern(contents)));
        } else {
          throw error("Symbol plug-in: "+ next+ "appeared after ¥":¥" .");
        }
      } else {
        return original();
      }
    }
    define implement Tree symbolOtherwise(){
      throw error("symbol is required here.");
    }
    extend void initMacroTable() {
      original();
      defineMacro(:symbol, new SymbolMacro());
    }
  }
}
class SymbolMacro extends Macro {
  public Tree call(Tree tree){
    checkArgsLength(tree, 1);
    Tree[] args = tree.args();
    String str = args[0].idName().toString();
    Tree var = new Identifier(genTemp("_Sym"));

    addTree(:beginningOfClassBody,
            `(decl (modifiers (id private) (id static)
                              (id final))
               (id Symbol)
               (vardecls
                (varInit ,var
                 (invokeLong (id Symbol) (id intern)
                  (argumentList ,(new LiteralTree(:string, str))))))));
    return var;
  }
}
```

Grammar extension

Macro expansion

Grammar extension

```
#epp jp.go.etl.epp.Symbol
#epp jp.go.etl.epp.SystemMixin
#epp jp.go.etl.epp.AutoSplitFiles
#epp jp.go.etl.epp.BackQuote
#epp jp.go.etl.epp.EppMacros

...

   extend Tree symbolTop() {
      if (lookahead() == :":") {
        matchAny();
        Token next = matchAny();
        if (next.isSymbol()){
          return new Tree(:symbol, new Identifier((Symbol)next));
        } else if (next.isLiteralToken()){
          String contents = next.literalContents();
          return new Tree(:symbol,
                          new Identifier(Symbol.intern(contents)));
        } else {
          throw Epp.fatal("");
        }
      } else {
        return original();
      }
    }
```

Macro expansion

```
class SymbolMacro extends Macro {
  public Tree call(Tree tree){
    checkArgsLength(tree, 1);
    Tree[] args = tree.args();
    String str = args[0].idName().toString();
    Tree var = new Identifier(genTemp("_Sym"));

    addTree(:beginningOfClassBody,
            `(decl (modifiers (id private) (id static)
                      (id final))
               (id Symbol)
               (vardecls
                (varInit ,var
                 (invokeLong (id Symbol) (id intern)
                  (argumentList ,(new LiteralTree(:string, str))))))));
    return var;
  }
}
```

GJ(Generic Java)

```
#epp jp.go.etl.epp.Generic
import java.util.Stack;

public class TStack<T> {
  private Stack stack = new Stack();
  void foo(){
    T x = null;
  }
  void push(T val){
    stack.push(val);
  }
  T pop(){
    return (T)stack.pop();
  }
}
```

Multiple inheritance

```
#epp jp.go.etl.epp.MultipleInheritance

class C1 {
  int m11() { return 11; }
  int m12() { return 12; }
}
class C2 {
  int m21() { return 21; }
  int m22() { return 22; }
}
class C3 {
  int m31() { return 31; }
  int m32() { return 32; }
}
class C4 extends C1, C2, C3 {
  int m41() { return 41; }
  int m42() { return 42; }
}
```

Composability

- Generic and MultipleInheritance plug-ins.

```
#epp jp.go.etl.epp.Generic
#epp jp.go.etl.epp.MultipleInheritance

import java.util.Stack;
class TStack<T> extends A1, A2 {
  private Stack stack = new Stack();
  void foo(){
    T x = null;
  }
  void push(T val){
    stack.push(val);
  }
  T pop(){
    return (T)stack.pop();
  }
}
```

Plug-ins currently implemented

- C, C++, g++ features
  - #ifdef, #include, enum, defmacro, assert macro, operator overloading, optional parameters, typeof
- Lisp features
  - Symbol, back quote macro, mixin, progn
- ML features
  - parameterized modules
- Perl features
  - association array

Design goal of EPP

- **High extensibility**.
  All extensions which can be done by "editing of original source-code" should be able to be done by EPP plug-ins.
- **High composability.**
  Plug-ins should work correctly even if they are used with other plug-ins.
- (C.f. Language extension systems based on attribute grammar or term rewriting system…)

EPP can be used as framework of ...

- Java extensions.
  - Operator overloading, parameterized modules, ...
- Source-code analyzing tools.
  - Metrics, cross reference, ...
- Source-code translation tools.
  - source level optimization, obfuscation, ...
- **Aim: A standard platform of Java related tools.**

Why EPP is useful as framework?

- **EPP will support all the following troublesome work.**
- Java grammar parser（Cf. JavaCC, ANTLR,...）
- Abstract syntax tree manipulation libraries.
- **Type checking**.

Now implementing the followings.

- Automatic re-compilation.
- ...

Aim: Making a programming language like a "**PC compatible machine**"

- EPP provides **the standard interface** for extension modules for language processors.
- Small venture companies or Universities can produce practical language parts.
- **Free competition**.
- EPP will accelerate the progress of programming languages.

For language researchers

- Before EPP (PhD languages)
  - More than half an year to implement an experimental programming language system.
  - Developing tools(debuggers, etc.) are future work.
  - Only a few users.
- After EPP
  - 3 hours ～ 3 month to implement.
  - Existing developing tools can be reused.
  - 100～10000 users can try the new language.
  - The idea becomes practical system immediately.

For programmers

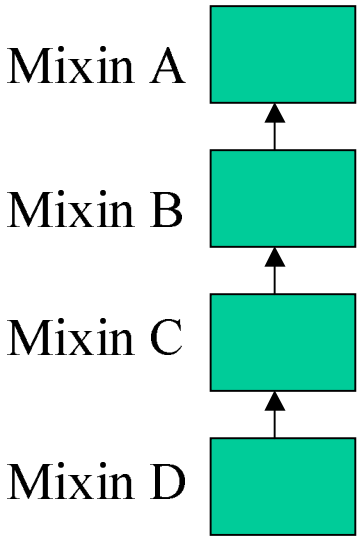
- Before EPP
  - Language features **selected by the designer** to suit his own tastes are imposed.
    (No free competition !)
  - The latest research results are not adopted to the user's language processor.
- After EPP
  - Programmers can select their **favorite** features.
  - The latest research results can be used immediately.

Technical characteristics

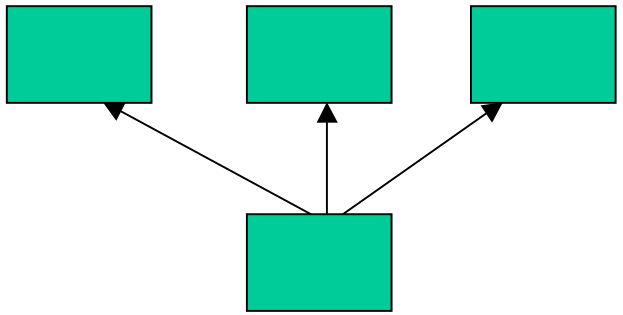
- **Mixin based design** which increase extensibility and re-usability of modules
  - Traditional object-oriented design is not enough.
  - Design pattern is still not enough.
- **Extensible architecture**
  - extensible lexical analyzer
  - extensible recursive descent parser
  - extensible type checking system

What is mixin ?

- “Abstract subclasses” which are not depending on specific super classes.
- A class can be constructed by composing and linearizing mixins.



NOTE: It is impossible to implement using C++ multiple inheritance.



Why mixins are important in EPP?

- Mixins supports programming-by-difference and modular programming.
  - Mixins increases composability of EPP plug-ins.
- Mixin mechanism is provided as an EPP plug-in. (Mixins can be separately compiled.)
- EPP itself and EPP plug-ins are written in Java with the “Mixin plug-in.”
  - And bootstraped by EPP written in Common Lisp.

A program NOT using mixins.

- Not flexible.

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

A program using mixins.

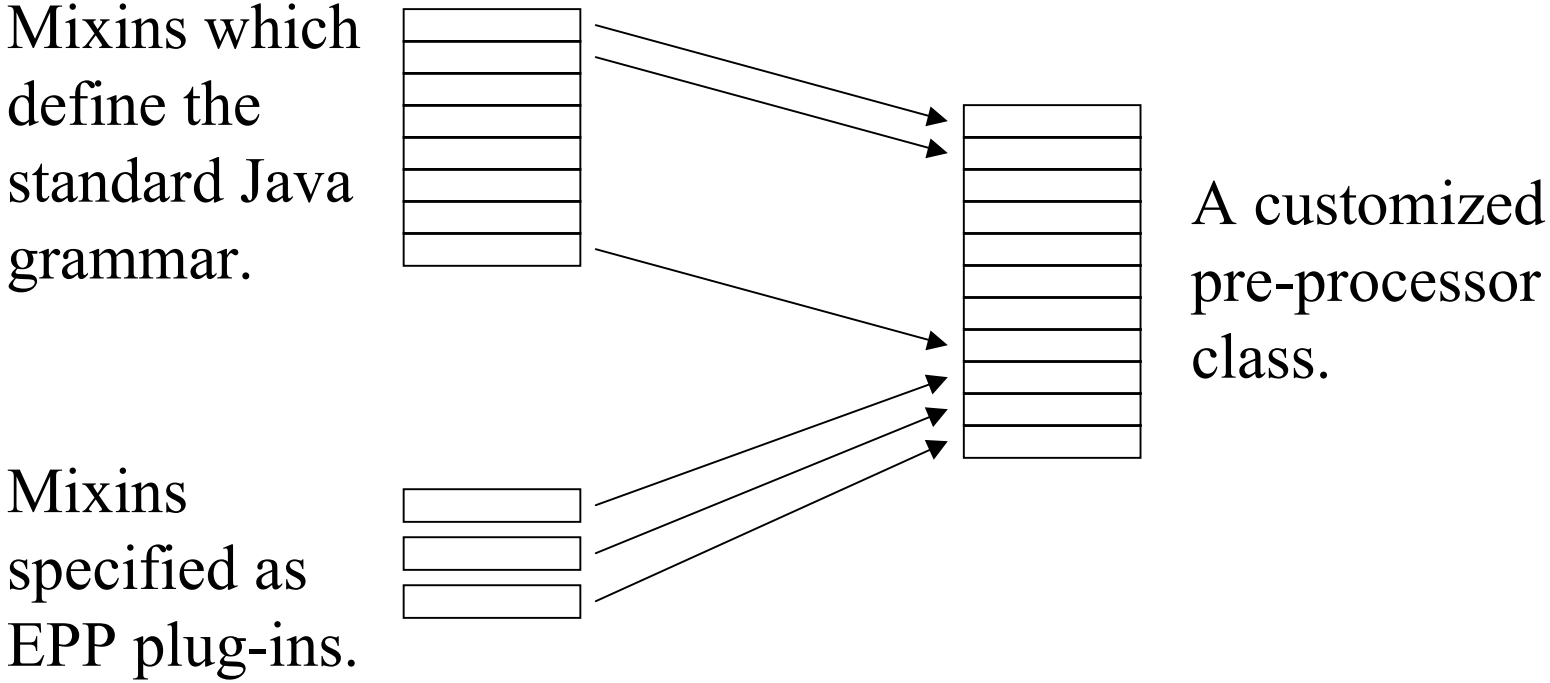
```
SystemMixin Skeleton {  class Foo {
    define void m(char c){ doDefault();   }
}}
```

```
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); }}
}}
```

How EPP starts pre-processing?

- The EPP main routine composes mixins, constructs one pre-processor class, and executes it.



How to write parser with modular and extensible way.

- Recursive descent parser consists of nested if statements.
- Nested if statements can be written with modular way using mixins.
- Each non-terminal and each alternative becomes a mixin.

An example of a non-terminal.

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

This can be rewritten and become LL(1).

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

Recursive Descent Parser(without mixins)

```
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(); }
```

Extensible parser skeleton.

```
SystemMixin Exp {
  class Parser {
    define Tree exp(){
      Tree tree = expTop();
      Tree newTree;
      while (true){
        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(); }
  }
}
```

Exp → Term

Adding a left associative operator.

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

Exp → Exp + Term

Adding a right associative operator.

```
SystemMixin Assign {
  class Parser {
    Tree expRight(Tree tree) {
      if (lookahead() == :"+=") {
        matchAny();
        return new Tree(:"+=", tree, exp());
      } else {
        return original(tree);
      }
    }
  }
}
```

Exp → Term += Exp

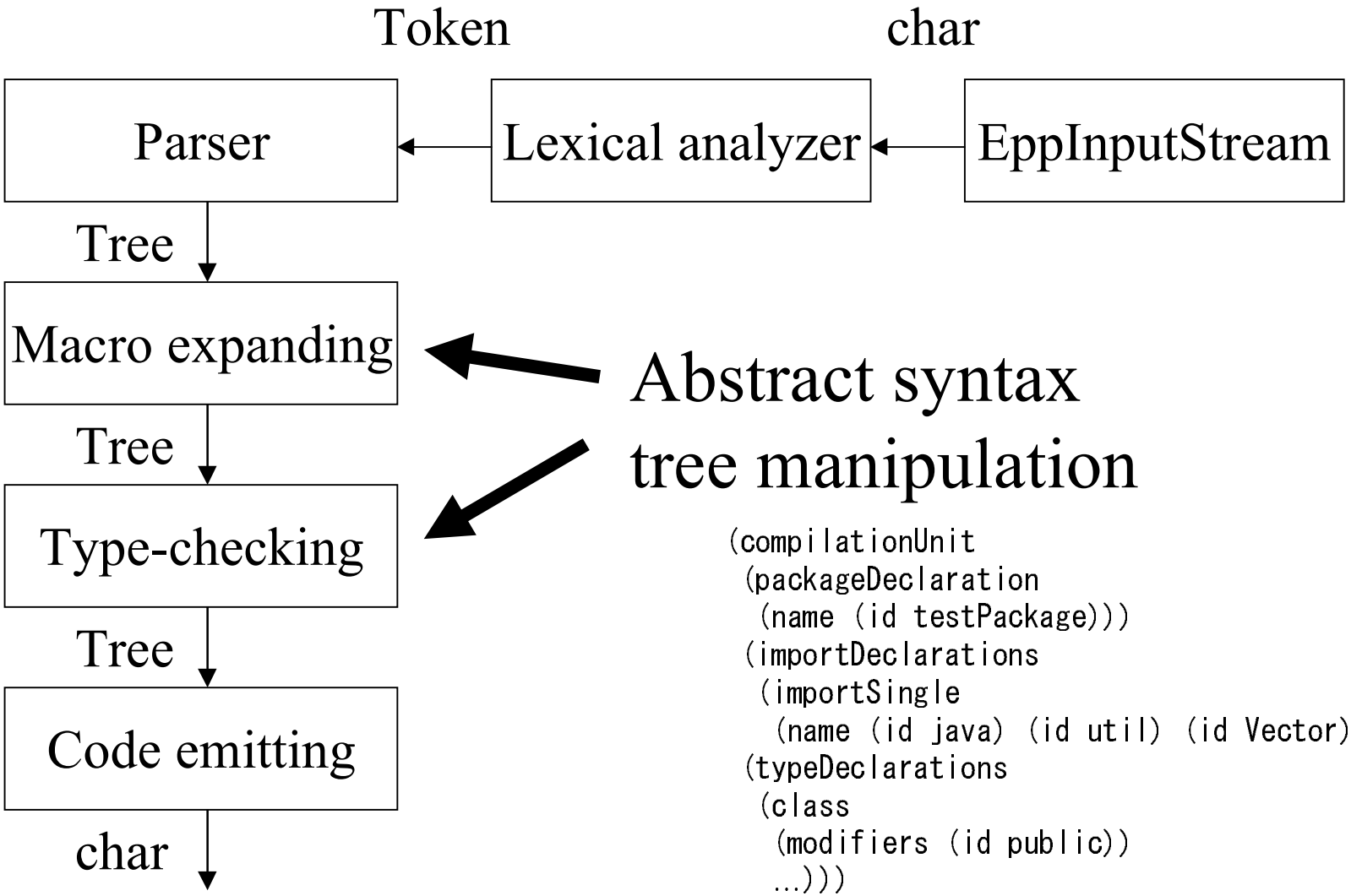
Java grammar

- 105 mixins.
  - 29 mixins define non-terminals and others define alternatives.
- I used 4 explicit backtrack because some production rules are not LL(1).

EPP architecture

- Simple.
  - Easy to understand its behavior.
- General-purpose.
  - Data structures (**Token**, **Tree** and **Type)** can express various programming languages.
- Extensible.
  - All parts of EPP are extensible.

4 passes

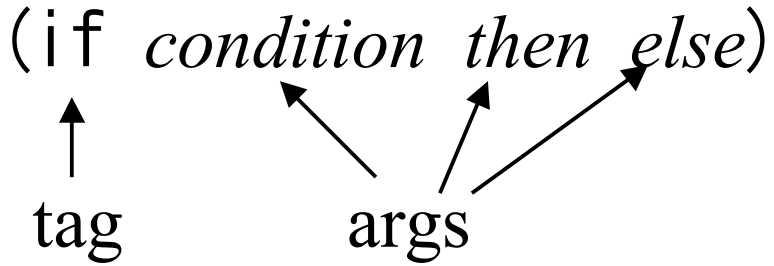


Abstract syntax tree manipulation

```
(compilationUnit
 (packageDeclaration
  (name (id testPackage)))
 (importDeclarations
  (importSingle
   (name (id java) (id util) (id Vector)))
 (typeDeclarations
  (class
   (modifiers (id public))
   ...)))
```

Abstract Syntax Tree

- Tree is an immutable object.
- Simple API:
  - Symbol tag();
  - Tree[] args();
  - Type type();
  - …

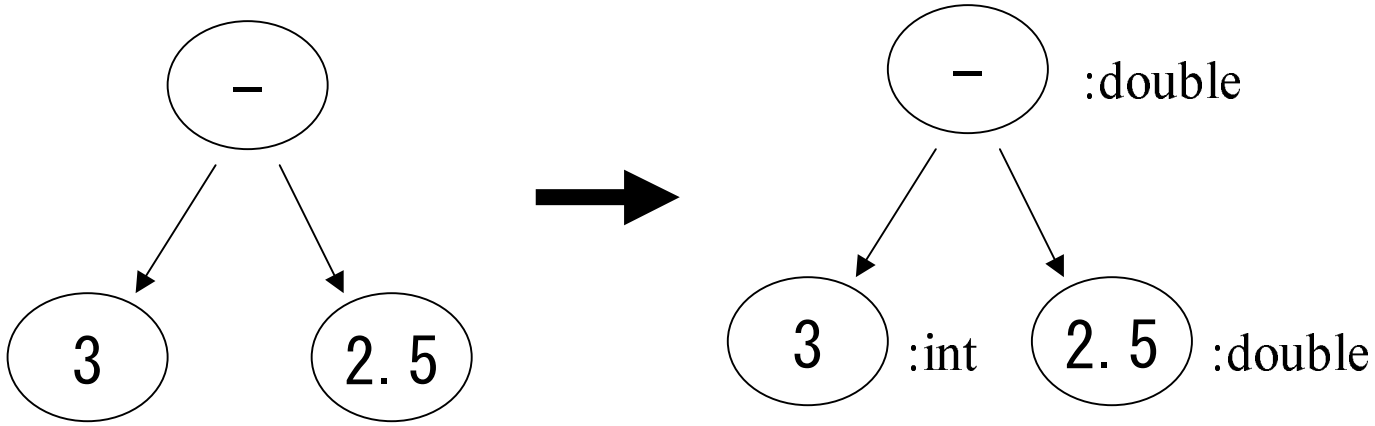


- “Back quote macro” plug-in is provided.

```
`(invokeExp ,target ,mehodName
            (argumentList ,@args))
```

Type checking pass

- Translate AST which does not have type information into AST which has.



Type checker object

- "TypeChecker" is associated with all tree tags.
- Extensible using *decorator pattern*.
  - Example: operator overloading

```
class ExpandOperatorOverloadingOfMinus extends TypeChecker {
  public Tree call(Tree tree) {
    Tree[] newArgs = typeCheckArgs(tree);
    Type t1 = newArgs[0].type();
    if (t1.tag() == :class){
      // Return the AST of "e1.minus(e2)" .
      return `(invokeExp ,(newArgs[0])
              (id minus) (argumentList ,(newArgs[1])));
    } else {
      return orig.call(tree.modifyArgs(newArgs));
    }
  }
}
```

Relationship between types

- Extensible using *mixin mechanism*.
  - **Not OO style.**
  - **But as modular and extensible as OO style.**
- API:
  - `boolean isSuperType(Type t1, Type t2);`
  - `Tree selectMostSpecificMethod(...);`
  - ...

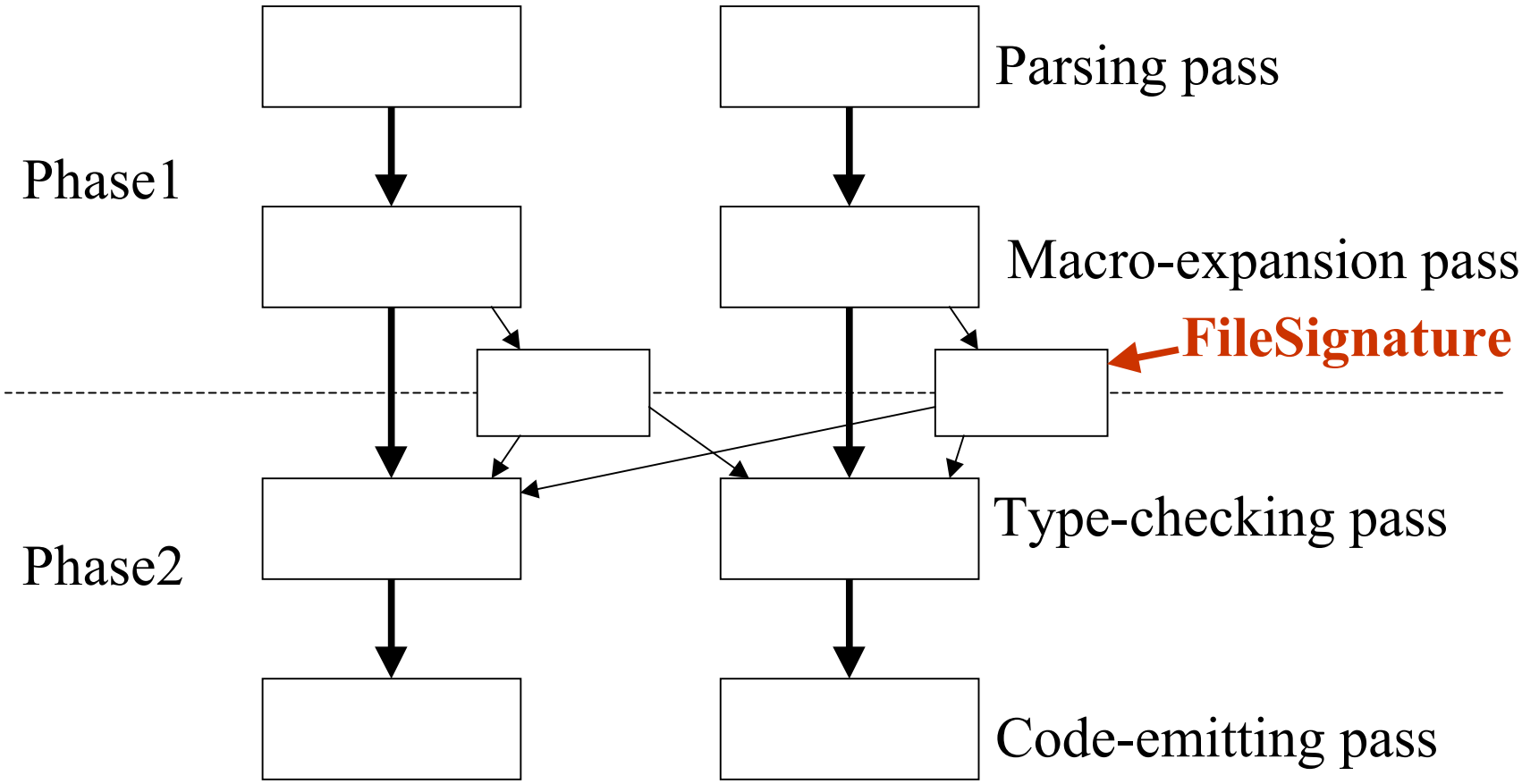
Example: extending isSuperType

```
boolean isSuperType(Type t1, Type t2){
    if (t1.tag() == :assocArray){
      if (t2.tag() == :assocArray){
        AssocArrayType a1 = (AssocArrayType)t1;
        AssocArrayType a2 = (AssocArrayType)t2;
        // Contra-variant role.
        return isSuperType(a1.getKeyType(),
                           a2.getKeyType())
           && isSuperType(a2.getValueType(),
                           a1.getValueType());
      } else {
        return false;
      }
    } else {
      return original(t1, t2);
    }
  }
```

Separate compilation

- Java language:
  - No forward declarations / header files.
  - No Makefiles.
  - These information is automatically generated by **javac** and written into class files.
- EPP's framework has generalized javac's mechanism.

Separate compilation of EPP



Files do not depend on each other directly.

Related work

- Nothing except for EPP achieve both of high extensibility and high composablity.
  - Parser Generator(Yacc,JavaCC,ANTLR, ...)
    Compiler Generator (Eli, Rie,...)
  - Extensible Language (EL1, lisp, ...)
  - Reflection (3Lisp, ABCL/R3,AL/1, ...)
  - Meta Object Protocol (CLOS-MOP)
  - Extensible translator (TXL, Sage++, Camlp4, …)
  - Compile Time MOP (MPC++, OpenC++, OpenJava, JTRANS)

Application 1. Mobile agent system

- PLANET project at Tsukuba University
- Pure Java implementation of thread migration.
  - **No JavaVM modification**.
  - **Almost 0% overhead** will be achieved.(?)
- Context saving/restoring code is added.
- Related work:
  - [Funfrocken98] 10% overhead.
  - Porch [Ramkumar, Strumpen97] For C language. More than 100% overhead.

Application 2. Java metrics tool

- JavaMetrics project by Electrotechnical laboratory, el.al.
- Enhance reliability of Java source code.
  - Code audit.
  - Code metrics.
  - Instrumentation for test coverage.

Conclusion

- EPP plug-ins can extend the pre-processor with programming-by-difference style.
- EPP achieves both of high extensibility and high composability.
- Separate compilation is not implemented yet.
- Distributed with source-code and examples.
  - http://www.etl.go.jp/~epp/