EPFL

DI − LAMP

SEMESTER PROJECT

Operator overloading in Java

Submitted by:
Student Yacine Saidji
6th semester
Student ID No.: 750959

Advisor:
Matthias Zenger

Ecublens, 30/06/2000

Table of Contents

1 Why Operator Overloading ?

The Java Grande Forum[JGF98] has identified the need for the operator over−loading feature, i.e. the capability to assign an operator name to a method, in the Java[GOS96] programming language. This feature, which is available in languages as Eiffel[MEY92], Sather[OMO93] and C++[STRO87], greatly enhance the readability and maintainability of code that makes extensive use of classes to represent numerical types by allowing a more natural invocation style. Therefore, if operator overloading is made available in Java, the scientific computing community would choose it for a greater number of projects.

In Eiffel, classes have features that can be attributes or routines, in which case the feature name can be an operator name (predefined or free). A programmer can define a meaning for binary (prefix) operators and unary (infix) operators. Note that an Eiffel class can not have two features with the same name (over−loading), thus preventing the implementation of a class representing complex numbers (for example) with one routine called "+" which takes another com−plex as a parameter and another routine also called "+" which takes a real num−ber as a parameter. Precedence rules can not be specified.

In Sather, operator overloading is mimicked by the use of syntactic sugar where predefined operators are rewritten to predefined routine calls without using any type analysis. For example, the following expression " r := (x^2+y^2).sqrt " gets transformed into " r := (x.pow(2).plus(y.pow(2))).sqrt ".

C++ offers a very complete operator overloading mechanism allowing the pro−grammer to define the meaning of arithmetic, logical and relational operators as well as call ( ), subscripting [ ] and dereferencing −>. It is however not possible to define functions for free operators, i.e. operators not part of the predefined operators set. As in Eiffel, precedence rules can not be specified.

Some attempts to extend Java with operator overloading have been made in the past by the use of pre−processors as Jfront[JF99] with a style very close to the Sather one, thus making it unsafe and not very flexible.

The approach followed in this project permits the definition of new (free) operators and let the programmer define if the operator takes a left or a right operand. In addition to that, extensive type checking is performed in order to resolve ambiguities and prevent runtime errors.

Once the operator overloading feature is supported by Java, it would enable writing code that looks like :

```
class complex {
    float re, im;
    complex(float re, float im) {
          this.re = re;      this.im = im;
    }
    complex this + (complex c) {
          return new complex(c.re + re, c.im + im);
    }
    complex (float re) + this {
          return new complex(this.re + re, im);
    }
    public static void main(String[] args) {
          complex c = new complex(1,1);
          c = c + c;
          c = 1 + c;
    }
}
```

Moreover, Java statements as c = a.times(b).plus(c) could then be expressed in a much more readable form, i.e. c = a*b+c.

This report describes how Java can be extended in order to support Operator Overloading and discusses some implementation issues.

2 Specification

This section summarizes the additions to Java's syntax and semantics [GOS96] in order to support binary operators overloading.

2.1 Lexical Grammar

The operators that can be overridden are described by the following lexical grammar:

OverridOperator:
    PredefinedOperator
    UserDefinedOperator

PredefinedOperator: one of
    * % + − < > & | ^ / << >> >>> <= >=

UserDefinedOperator:
    ConstructedOperator but not a *PredefinedOperator* or *ForbiddenOperator*

ConstructedOperator:
    OperatorLetter
    ConstructedOperator OperattorLetter

OperatorLetter : one of
    * % + − < > & | ^ ! ? ~ @ # =

ForbiddenOperator : one of
    ? : ~ ! = ++ −− && || += −= *= ^= %= <<=
    >>= >>>= == /= &= |=

Note that the '/' letter is not present in the *OperatorLetter* set in order to avoid ambiguity with the *EndOfLineComment* production.

2.2 Syntactic Grammar

The following terminal is added :

```
Name OPERATOR
```

The following non terminal are added :

```
Integer PredefinedOperator;
Integer Operator;
Tree TypeButNoSimpleName
```

The expansions of *PredefinedOperator* are :

```
PredefinedOperator : one of
        STAR SLASH PERCENT PLUS SUB LTLT GTGT
        GTGTGT LT GT LTEQ GTEQ AMP BAR CARET
```

The expansions of *Operator* are :

```
Operator :
        OPERATOR
        PredefinedOperator
```

The expansions of *TypeButNoSimpleName* are :

```
TypeButNoSimpleName :
        PrimitiveType
        QualifiedName
        ArrayType
```

The declaration of an operator method takes place in the *MethodHeader* pro−duction. Here are the added expansions :

```
MethodHeader :
        MethodModifiers_opt Type
                this Operator ( FormalParameter )
                Throws_opt
```

```
MethodModifiers_opt void
        this Operator ( FormalParameter )
        Throws_opt
MethodModifiers_opt TypeButNoSimpleName
        ( FormalParameter ) Operator this
        Throws_opt
MethodModifiers_opt void
        ( FormalParameter ) Operator this
        Throws_opt
```

Due to a shift−reduce conflict, an operator method declaration has to be added to *ConstructorDeclarator* expansions :

```
ConstructorDeclarator :
        SimpleName ( FormalParameter ) Operator this
```

Note that there are two different ways of declaring an operator method. The first two one declare a left−operand operator and the others declare a right−operand operator.

These operators can then be used in binary expressions. An expansion is added to the *BinaryExpression* production :

```
BinaryExpression :
        BinaryExpression OPERATOR BinaryExpression
```

The user defined operators (OPERATOR) have the lowest precedence com−pared to predefined operators and are syntactically left−associative (they group left−to−right).

2.3 Semantic

2.3.1 Operator Method Declarations

Operator methods can be put in interfaces and classes. An operator method defines a binary operator. There are two types of operator methods, left−operands and right−operands.

For example, in the following class, two operator methods are defined :

```
class foo {
          int this # (int i) { return 1;}
          int (int i) @ this { return 2;}
}
```

The # operator is a left−operand operator method,which means that the operand on the left of the operator, expressed by the *this* keyword, is of type *foo*.

The @ operator is a right−operand operator method,which means that the operand on the right of the operator, expressed by the *this* keyword, is of type *foo*.

The signature of an operator method consists of the operator, the type of the formal parameter and whether it is left−operand or not.

It is a compile−time error for the body of a class or an interface to have two operator methods with the same signature.

The example :

```
class complex {
          complex this + (float re) { ... }
          complex (float re) + this { ... }
          ...
}
```

compiles without errors, as one of the + operator method is a left−operand and the other a right−operand. The two + operator methods have different signatures.

However, the following example :

```
class complex {
          ...
          complex this + (float re) { ... }
          float this + (float re) { ... }
          ...
}
```

causes a compile−time error because it declares two + operator methods with the same signature.

A compile−time error occurs if two operator methods of a class C have the same operator name and a parameter of type C but one is a left−operand and the other a right−operand.

For example :

```
class foo {
          int (foo b) # this { return 1; }
          float this # (foo b) { return 2.0F }
}
```

causes a compile−time error to prevent ambiguities when the # operator is invoked as in the following code :

```
          foo a = new foo();
          foo b = new foo();
          System.out.println(a # b);
          ... // ambiguous ! Should it print 1 or 2 ?
```

Like all regular Java methods, operator methods can have method modifiers in front of their declaration (the same semantic applies). All the modifiers are allowed except *static*.

If two operator methods of a class have the same operator name and are both left or right−operands with different signatures, then the method operator is said to be overloaded. As with conventional Java methods, overriding is done on a signature−by−signature basis.

2.3.2 Operator Method Invocation Expressions

Operator methods are invoked in binary expressions. Resolving an operator is similar to resolving a method name.

Compile−time processing of an operator invocation is the following :

1. If both operands are numerical. Try to resolve the operator in the conventional Java way.
2. If the left operand is not−numerical. Check if its class has a method having the signature {operator, left−operand, type right−operand}.
3. Same thing but reversed. (for right−operand operators)
4. A compile−time error occurs if both left−operand and right−operand operator methods are available in order to report the ambiguity.
5. A compile−time error occurs if no matching operator method is found.

The order of evaluation remains left to right as guaranteed by Java.

At run time, operator method invocation is performed the same way than run−time evaluation of conventional Java method invocation.

The following example illustrates compile−time resolution:

```
class A {
          ...
          int (int i) # this { return 1; }
          int this # (int i) { return 2; }
          public static void main(String[] args) {
          A a = new A();
          System.out.println( 1 # a );
          System.out.println( a # 1);
          }
}
```

It should print 1 followed by 2.

The following example illustrates ambiguity detection performed at compile−time :

```
class A {
          int (B b) # this { return 1;}
}
class B {
          int this # (A a) { return 2;}
}
class Test {
          public static void main(String[] args) {
                    A a = new A();
                    B b = new B();
                    int i = b # a; // compile−time error
          }
}
```

This example produces an error at compile time. The problem is that there are two declarations of # that are applicable and accessible, and neither is more specific than the other. Therefore, the operator method invocation is ambigu−ous.

2.3.3 Predefined operator methods

Some Java classes already have operator methods. These operator methods have to be rewritten in order to be usable with the new scheme.

For example, the modifications in the class String should look like :

```
class String {
          ...

          String this + (int i) {
                    return concat.( valueOf(i) );
          }
          ... // same thing for all the primitive types

          String this + (Object o) {
                    return concat.( o.toString() );
          }

          ... // same as the previous ones, right−operand style

          String (int i) + this {
                    return valueOf(i).concat(this);
          }
          ... // same thing for all the primitive types

          String this + (Object o) {
                    return o.toString().concat(this);
          }

}
```

Once such a class is defined and made accessible, statements as

```
          System.out.println( "hello " + 1 + " world" + "!");
```

would be valid and computable.

3 Jaco

Jaco[ZEN98] is an extensible Java compiler that has been designed to resolve two major problems in conventional compilers : maintainability and reusability. Jaco is written in an extension of itself which is a programming language where the extensible algebraic types feature is added to Java.

The architecture of the Jaco compiler consists of about 30 components, most of them extensible, that can be organized in 4 families :

- Syntactic Analyzer, composed by the scanner and the parser
- Semantic Analyzer, composed by the modules responsible of the name and type checking
- Backend, composed by the modules responsible of the target code generation
- Basic System, composed by some helpers modules used by the other components

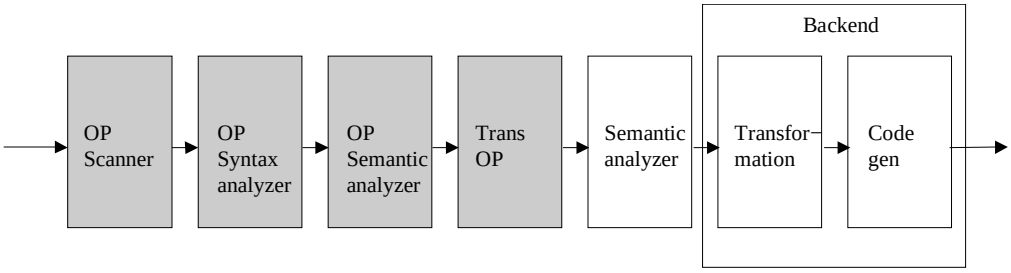
In most cases, Jaco can be extended by adding new components, the two main steps being :

- Producing and checking an extended AST (Abstract Syntax Tree)
- Transforming the AST so it can be processed by the back end of Jaco

The developer can focus on his extensions when writing new components as most of the existing code can be reused using the inheritance feature offered by Java. Typically, a new component is a subclass of an existing one, and, using the overriding mechanism in a very local fashion, is adapted to handle the extension specification.

4 Extending Jaco

This section describes the most important components that have been added to the Jaco compiler framework in order to support Operator Overloading.



The first step was to create a new syntactic analyzer. It was done by writing a new scanner that supports the user defined operators and reports them as OPERATOR tokens. A new parser has also been written in order to build a proper method declaration node in the AST when an operator method declara−tion in encountered. The parser is also responsible for producing a correct statement node in the AST when a user defined operator is used in an expres−sion.

Once a fresh AST is produced, a semantic check action has to be performed. A new component, in charge of the tree attribution pass and type checking, has been written. This component handles the binary operator expression nodes involving a user defined operator. Another component, responsible for the name checking, has also been written.

The final step was to transform the AST in order to enable the target code gen−eration by Jaco backend. To perform this transformation, a new component has been created. Its responsibility is to build the proper method call when a user defined operator is used in an expression.

More details are given in the following sections.

4.1 Scanner

A new terminal symbol called OPERATOR has been added in the grammar specified in the *Grammar.cup*. This file is used by the JavaCUP[HU96] tool to generate the parser (see section 4.2).

The new scanner inherits from the original Jaco *Scanner* component and over−rides the *getspecials* method and the *ispecial* method. The *getspecials* method is responsible for returning a token for a sequence of specials characters. This token is OPERATOR if the sequence is not a predefined Java operator.

4.2 Parser

The parser is generated by the JavaCUP tool which is provided the grammar definition expressed in the *Grammar.cup* file.

The extensions made to this file are the following :

- Handling of the Operator methods declarations
- Handling of the expressions involving user defined operators

The Operator method declarations are handled in the *MethodHeader* production and, in order to resolve a shift−reduce conflict, in the *ConstructorDeclarator* production. In the produced nodes, the method name is a mangled name generated by the operators repository component enabling in later stage method resolution using an unambiguous name reconstructed from the expression context. The mangled name expresses the fact the the method is an Operator method and specifies if it’s a left−operand or a right−operand operator.

The mangling scheme is the following :

- *OP$$L$xx* for a left−operand operator where *xx* is the operator
- *OP$$R$xx* for a right−operand operator where *xx* is the operator

The expressions involving a user defined operators are handled in the *BinaryExpression* production and are encoded as *Binop* nodes in the AST with an *opcode* generated by the operators repository. The predefined operator have various precedence levels defined in the Java specification; user defined operators all have the same precedence, lower than that of the predefined operators.

The operators repository component inherits from the original Jaco *Operators* component. The main extensions to this component is the name mangling (coding and decoding) and the opcode generation.

4.3 Pretty−printer

A new pretty−printer has been written. It inherits from the original Jaco *Pret−tyPrinter* and the method handling the method declarations has been overrid−den.

When the pretty−printer encounters a method declaration node while walking the AST, it derives the correct string of characters when the declaration is an Operator method declaration, otherwise the code of the original Jaco pretty−printer is executed.

Similarly, the correct string of characters is derived from *Binop* nodes. In this case, the operator "name" is retrieved from the operators repository . This mapping is performed by a method, called *toString*, which takes an opcode as an argument and returns the associated name. Note that the code of the original Jaco pretty−printer is reused to produce this derivation. No changes were nec−essary as the *toString* method of the operators repository handles the predefined operators as well as the user defined ones.

The operators repository is also responsible for providing the precedence of the operators. The predefined operators are given the the precedence specified in the Java language and the user defined operators are given the same precedence as the *add* operator one.

4.4 Semantic check

A new component, responsible of the tree attribution has been written. It inherits from the original Jaco *Attribute* component. In this component, the method handling the *Binop* nodes has been overridden.

The new method performs the attribution and the checks in two main steps :

- retrieving the method name associated with the opcode given the context
- attributing the correct method to the Binop node

The first step is performed by calling the *resolveOperator* method of the new *NameResolver* component. This component, responsible of the name checking, inherits from the original Jaco *NameResolver* component and overrides the *resolveOperator* method. In the new implementation, argument matching determines which Operator method is used. If two Operator methods are found, one being a left−operand operator and the other a right−operand operator, an error message reporting the ambiguity is outputted.

Once the correct method definition is returned to the new Attribute component, it is persisted in the AST. The result type of the expression is also set in the *Binop* node to enable type checking.

When both operands are of numerical type, the code of the original Jaco components (*Attribute* and *NameResolver*) is executed.

Another verification is performed when entering the Operator method declarations in the environment. This check ensures that no ambiguities that can be detected by just looking at the methods declarations are present.

The last described check enables error reporting when the following class, pro−vided as an example, is being compiled :

```
class A {
          ...
          A this + ( A a ) { ... }
          A ( A a ) + this { ... }
          ...
}
```

This prevents the following ambiguity : " Does a + b mean a.+(b) or b.+(a) ?", assuming a and b are instances of the class A.

4.5 AST transformation

The component in charge of the AST transformation inherits from the original Jaco component called *Translator*. The *translateExpr* method has been over−ridden in order to transform the *Binop* nodes where an Operator method is involved to *Apply* nodes.

This mapping into a method dispatch is performed in two steps:

- Building a *Select* node
- Building an *Apply* node

To build the *Select* node, the selected argument has to be determined. This is done by looking at the left or right operand property contained in the mangled name. The *Apply* node is then build by providing the other argument as a parameter to the method call.

In order to not reverse the original left to right order of argument evaluation, temporary variables are used when right operand operators are invoked.

So, for example, the test program :

```
class A {
          int i;
          A(int i) { this.i = i; }
          A this @ (int j){ i = (0!=i) ? j : i; return this; }
          int (A a) # this { return i; } // Right operand !!
          public static void main(String[] args) {
                    A a = new A(0);
                    System.out.println( (a @ 1) # (a @ 2) );
          }
}
```

prints : 1 .

It is not permitted for it to print 2 instead of 1 even if the expression will look like this pseudo−code (a.@(2)).#(a.@(1)).

Note that the Operator methods declarations remains unchanged as the AST stores them as Java class methods with mangled name.

For the curious reader, the *complex* class shown in the first section of this document is decompiled by Mocha[HPV96] as following :

```
synchronized class complex {
    float re;
    float im;
    complex(float f1, float f2) { re = f1; im = f2; }

    complex OP$$L$+(complex Complex) {
        return new complex(Complex.re + re, Complex.im + im);
    }
    complex OP$$R$+(float f) {
        return new complex(re + f, im);
    }
    public static void main(String astring[]) {
        float OP$$TMP$1;
        complex Complex = new complex(1.0F, 1.0F);
        Complex = Complex.OP$$L$+(Complex);
        OP$$TMP$1 = 1.0F;
        Complex = Complex.OP$$R$+(OP$$TMP$1);
    }
}
```

5 Conclusion

This project has demonstrated the fact that the Java language can be extended with the Operator Overloading feature. The Jaco compiler has been extended in a fairly straightforward manner thanks to its architecture.

The project implementation only supports binary operators although unary operators could be handled in a very similar way.

Some future directions involve the ability to specify the precedence of an operator as well as its associativity. The lexical specification of an operator is also an issue that needs to be addressed. On the method invocation side, it would be extremely useful to provide a *super* form.

This project has shown that the main effort is not to implement the compiler but to produce a usable specification. The one described in this document could be used as a starting point for a Java community wide discussion.

Acknowledgments

Many thanks to Matthias Zenger for his advices.

6 References

GOS96: James Gosling, Bill Joy, and Guy Steele, Java Programming Lan−guage, 1996

STRO87: Bjarne Stroustrup, The C++ Programming Language, 1987

OMO93: Stephen Omohundro, The Sather 1.0 Specification, 1993

MEY92: Bertrand Meyer, Eiffel: The Language, 1992

JGF98: Java Grande Forum, http://www.javagrande.org, 1998

JF99: , Jfront − Operator Overloading for Java, http://www.winternet.com/~gginc/jfront/index.html, 1999

ZEN98: Matthias Zenger, Erweiterbare Ubersetzer, 1998

HU96: Scott Hudson, JavaCUP, a Parser Generator for Java, 1996

HPV96: H.−P. V. Vliet, Mocha java bytecode decompiler, 1996