FLEXIBLE OPERATORS IN SPARROW

Lucian Radu Teodorescu[1], Vlad Dumitrel[2], Rodica Potolea[3]

[1]*Technical University of Cluj-Napoca, University Politehnica of Bucharest, Technical University of Cluj-Napoca*
[2]*Technical University of Cluj-Napoca, University Politehnica of Bucharest, Technical University of Cluj-Napoca*
[2]*Technical University of Cluj-Napoca, University Politehnica of Bucharest, Technical University of Cluj-Napoca*

Abstract

Operators are a fundamental tool in programming languages, allowing computations to be expressed in a natural and convenient manner. Custom operators are supported by many programming languages, however their underlying mechanisms typically suffer from various limitations. This paper describes a method of defining and customizing operators at the library level in the Sparrow programming language. We argue that our system is more flexible than what other languages have to offer, and that it allows operators to be defined and used in a straightforward and natural way.

Keywords: *Programming languages, operators, flexibility, Sparrow, C++, Scala*

---***---

1. INTRODUCTION

Operators originated in mathematical notations, where they are used to conveniently represent various laws of composition, functions, or relations. Unsurprisingly, these concepts are just as necessary in programming languages. Most languages provide the typical set of arithmetic operators, along with relational, logical, and bitwise operators.

As the range of applications for computer programs grew, so did the need for other kinds of operators, many of which applied to non-mathematical objects. Examples include string concatenation using the **+** operator, and iteration using **++** and **--**. Eventually, it became more convenient for some languages to offer programmers the ability to define their own custom operators for user-defined types, instead of hardcoding a fixed set of operators into the language itself. Consequently, from a semantic standpoint, operators became more similar to functions.

An operator has various properties, which are especially relevant in a programming language context. From a syntactic standpoint, an operator can be prefix, infix, or postfix. Arity refers to the number of operands on which an operator is applied. Precedence and associativity determine the order of evaluation in compound expressions.

In this paper we propose a flexible and natural mechanism for user-defined operators in a language called Sparrow [1], [2]. This is a general-purpose, imperative programming language, influenced by C++. Sparrow itself is designed to be flexible, natural, and efficient, and therefore having a powerful operator system is of significant importance.

Our approach is library-based, and supports binary infix, as well as unary prefix and postfix operators. Furthermore, precedence and associativity are fully customizable for each operator, also at the library level. We argue that our solution is more versatile than those used in other languages, and can be employed with minimal effort from the programmer's part.

2. RELATED WORK

Although not all programming languages support operators in the traditional sense, the vast majority of programming languages have support for unary and binary operators. However, support for operator customization varies greatly across the spectrum of languages.

In Lisp there is no distinction between functions and operators; they are all written as the first token in a list expression. Haskell [3] is more flexible, allowing the use of arbitrary symbol tokens as operators, infix and postfix (there is only one prefix operator: -), and the use of prefix functions with infix notation. In addition, the user can customize the precedence and the associativity of these operators.

For our purposes, we are interested in languages that use C-like syntax for expressions. Languages like C and Java [4] have a fixed set of operators, their precedence and associativity rules are defined by the language, and they do not allow the user to overload these operators; they are the less flexible from this point of view. In C++ [5], even though the operators and the precedence rules are fixed, the user can overload operators for custom types. Scala [6] is even more flexible, allowing the user to create operators from arbitrary symbol tokens; identifiers can be used as infix and postfix operators, however the set of prefix operators is fixed (-, +, ! and ~). On the downside, Scala's rules for computing the precedence and associativity are fixed; they are based on the first and last characters of the token, except for assignment operators (=, +=, /=, etc.) which are handled differently.

Defining custom operators in C++ is done via a special **operator** construct, similar to a function declaration. To distinguish between prefix and postfix operator calls, C++ requires a dummy **int** parameter to be added when defining a postfix operator. Overloading infix operators in C++ can be performed both in the namespace of the operand type (recommended) or inside the class of the left operand. In Scala an infix expression **A op B** is syntactic sugar for **A.op(B)**, and the symbol operators are just regular identifiers. Similarly a postfix expression **A op** is equivalent to **A.op()**. On the other hand, a prefix expression **op A** (where op can only be **-**, **+**, **!** and **~**) is treated as **A.unary_op()**.

3. FLEXIBLE OPERATORS IN SPARROW

3.1 Design

Operators in Sparrow are designed to meet several requirements:

- operators should be defined in the library, not by fixed grammar rules
- the user must be able to create custom operators (infix, prefix, and postfix)
- the names of operators can be (almost) any combination of symbols
- the user must be able to use identifiers as operators (infix, prefix and postfix)
- definitions of custom operators are identical to definitions of regular functions
- the user must be allowed to set the precedence of the operators and change their associativity rules

These requirements ensure that programmers will have a large range of possibilities at their disposal when defining operators. Additionally, the operators will be easy to create and use.

3.2 Grammar Rules

The implementation for Sparrow operators begins with a set of lexical rules [7]:

```
Letter        ::= [‘a’-’z’ | ‘A’-’Z’ | ‘_’]
OpChar        ::= [‘~’|‘!’|‘@’|‘#’|‘$’|‘%’|‘^’|‘&’|‘-’|‘+’|‘=’|‘|’|‘\’|‘:’|‘<’|‘>’|‘?’|‘/’|‘*’]
OpCharDot     ::= OpChar | ‘.’
Operator      ::= OpChar*
                | OpCharDot+ ‘.’ OpCharDot*
IdLetters     ::= Letter (Letter | Digit )*
IdLettersOp   ::= IdLetters ‘_’ Operator
Identifier    ::= IdLetters | IdLettersOp
IdOrOperator  ::= Identifier | Operator
```

Identifiers from languages like C++, containing only letters and digits, will be accommodated easily by this scheme.

In addition, the user can construct identifiers by appending symbols after a regular identifier ending in ‘_’. This allows the construction of names such as **pre_++** and **post_++**.

Almost any combination of symbols that are typically available on an ASCII US keyboard can form a lexical token corresponding to an operator, with minor exceptions which are presented below. In theory, the list of characters accepted as part of operators can be extended as much as needed (e.g., with Unicode symbols), but for simplicity we listed only the ones that are commonly used in programming languages.

The following symbols cannot be used as part of an operator: **{**, **}**, **[**, **]**, **(**, **)**, **;** (semicolon), **,** (comma), **`** (backquote) because they act as special separators in the Sparrow syntax. Also, a closer look at the way the operators are defined reveals that we cannot use the dot symbol as an operator if there is no other dot in the sequence; this ensures that a single dot can always be used as a compound expression separator.

The following are valid examples of operators: +, -, ++, ******, **#$**, =/=/=/=. Valid examples of identifiers include **foo**, **bar123**, **_123**, **oper_$#@**.

Based on the above lexical rules, we defined the following parsing rules [7] for operators:

```
PostfixExpr   ::= InfixExpr IdOrOperator?
InfixExpr     ::= PrexixExpr
                | InfixExpr IdOrOperator InfixExpr
PrefixExpr    ::= SimpleExpr
                | Operator PrefixExpr
                | ‘`’ Identifier ‘`’ PrefixExpr
SimpleExpr    ::= Identifier | ...
```

These simple rules provide a wide range of usage possibilities, and are similar to the ones in Scala [6]. The main difference is the fact that we allow any operator or backquoted identifier to be part of a prefix expression.

Here are some examples of expressions involving operators: **a + b** (add **a** and **b**), **a + b * c** (add **a** to the product of **b** and **c**), ++**a**++ (prefix ++ on **a**, then apply the postfix ++), **v1 dot v2** (dot product between **v1** and **v2**), **`length` a** (calling length on a with prefix call notation), **1..100** (the numeric range between 1 and 100 , inclusive), **@Int** (type representing a reference to **Int**).

Ignoring prefix operators, if we have a sequence of identifiers and operators, then the terms in odd positions are always operands, and the terms in even positions are always operators. If the sequence has an even number of terms, then the last term is a postfix operator. Note that a **SimpleExpr** cannot contain operators; therefore we cannot find operators in odd positions (if there are no prefix expressions in the sequence).

Prefix operators always have precedence over other types of operators. If the compiler finds an operator in an odd position in the sequence, it means that it has encountered a prefix expression. This way, there is no ambiguity in distinguishing between prefix, infix, and postfix expressions.

Although these simple grammars rules cover most of the required Sparrow functionality, there are two main cases that are not handled.

The first case concerns the ternary operator: **condition ? alt1 : alt2**. Not only does this not fit grammatically in our scheme, but there are also some additional semantic issues associated with it. While any prefix, postfix and infix operator can be implemented as a function call (macro or regular function), there is no way to implement the ternary operator in this manner. This is mainly due to the fact that one alternative is not evaluated at all, whereas for a regular function all the arguments need to be evaluated before the actual function call. Normally, the boolean short-circuit operators (**&&** and ||) would share the same problem, but they can be implemented in terms of the ternary operator.
The second case is related to the handling of the = operator. The way the Sparrow grammar is constructed, there are situations in which we want to prohibit the use of the = operator inside expressions. For example, let us look at a variable definition:

```
var name: typeExpr = initializer;
```

Here, **typeExpr** is actually an expression that can contain prefix, infix, and even postfix operators. For instance, denoting a reference type requires a simple prefix operator call: **@Int**. If the = operator were allowed inside the type expression, then variable declarations would be ambiguous. The compiler would not know whether you are using the infix = operator or separating the type expression from the initializer of the variable. To resolve this, the relevant grammar rules have two versions: one that allows the = operator, and one that does not. The latter is used for variable declarations. If the symbol = had not been used by Sparrow as a syntactic separator, this issue would not have existed.

3.3 Defining Operators

In Sparrow, defining an operator is identical to defining a regular function:

```
class Complex {...}
fun + (x, y: Complex): Complex { return
Complex(x.re + y.re, x.im, y.im); }
fun - (x: Complex) = Complex(-x.re, -x.im);
```

Here we defined a binary operator and an unary operator. The second definition showcases a convenient method of defining a function that simply returns an expression.

With these definitions, one can actually use the two operators as infix, prefix, and postfix:

```
var a, b: Complex;
cout << a+b << endl;
cout << -a  << endl;
cout << a-  << endl;
```

As the reader may have noticed, the definition of the unary minus operator makes it possible for it to be used both as a prefix and a postfix operator. Sometimes this is not desirable; for example, in our case the **a-** notation is confusing. Another important example is the **++** operator, where the postfix and prefix versions have different semantics. To distinguish between the two types of operator calls, one can use the **pre_** and **post_** prefixes when defining the operators, as shown below:

```
fun pre_++ (x: Complex): @Complex { x += 1;
return x; }
fun post_++(x: Complex): Complex  { var old = x; x
+= 1; return old; }
```

3.4 Operator Lookup

Although operators behave like functions, there are some differences in terms of the way they are looked up. For a function call **f(a1, a2, …)** a name lookup is initiated for the name **f** from the context of the function call, going upward until one or more definitions with the name **f** are found. Afterwards, a function overload selection algorithm is applied to determine which definition should be called with the current arguments.

For operators, the algorithm is different. There are three contexts in which the operator is looked up:

- inside the class of the first operand (left operand for infix expressions, and the only operand of the prefix and postfix expressions)
- in the package that contains the class of the first operand
- starting from the context of the operator call and going upward

For unary operators, the compiler can search for two operator names:

- with a name prefix (**pre_** for prefix operators, and **post_** for postfix operator)
- without any prefix, just the operator name

Putting it all together, the compiler performs the following searches in order:

- in the class of the first operand, with a name prefix (for unary operators)
- in the class of the first operand, with the actual operator name
- in the package that contains the class of the first operand, with a name prefix (for unary operators)
- in the package that contains the class of the first operand, with the actual operator name
- upward from the context of the operator call, with a name prefix (for unary operators)
- upward from the context of the operator call, with the actual operator name

If a matching definition is found in any of these steps, the algorithm will not search any further.

Typically, operators for a new type are written inside or near the class. By choosing the definition that is closest to the class of the first operand, the algorithm maximizes the odds of finding the most suitable operator for the given argument type. This is similar to Scala [6], where **a + b** actually means **a.+(b)**, and to some extent to Argument-Dependent Lookup (also called Koening lookup) form C++ [5], [8].

3.5 Precedence and Associativity Rules

For infix operators, we also need to consider precedence and associativity. Precedence determines the order in which different infix operators inside the same expression are called. Associativity determines whether for an expression containing only operators of the same type the order of applying the operator is from left to right, or from right to left.

For each infix operator we can associate a numeric value such that we can compare the precedence of two operators. Let us denote by p_1 the precedence of the operator op_1 and by p_2 the precedence of the operator op_2. Then, the expression **A op1 B op2 C** would be interpreted as **(A op1 B) op2 C** if $p_1 \geq p_2$, and as **A op1 (B op2 C)** if $p_1 < p_2$. For example, multiplication and division have a higher precedence than addition and subtraction.

For an infix operator **op**, an expression like **A op B op C** would be interpreted as **(A op B) op C** if op has left associativity, and **A op (B op C)** if op has right associativity. Most of the mathematical operators have left associativity, but an operation like assignment makes sense to have right associativity. Also, if one were to define an exponentiation operator, it should also have right associativity.

Because Sparrow operators are defined in the library, there should be a possibility to define precedence and associativity in the library. There should be some kind of definition that the user can write, and the compiler can search for, when determining the precedence and associativity for infix operators. The easiest way this can be achieved in Sparrow is with using directives:

```
using oper_precedence_default = 100;
using oper_precedence_+       = 500;
using oper_precedence_*       = 550;
using oper_assoc_=            = -1;
```

Such a directive associates a name with a value; it is similar to defining a constant. Whenever the **+** operator is used in the language, the compiler searches for the name **oper_precedence_+** and, if successful, evaluates the expression to get the precedence value for the operator. If the name cannot be found, the compiler will use the precedence value denoted by **oper_precedence_default**. For associativity, if the returned value is negative, then the operator is considered to have right associativity.
Although these using declarations are convenient to write, there is a better method of getting and setting the precedence rules, by using function calls:

```
setOperPrecedence("*", 550);
setOperPrecedence("**", getOperPrecedence("*") +
1);
setOperRightAssociativity("**");
```

By using these functions, the user is not required to know the underlying details of operator precedence and associativity. Moreover, the **getOperPrecedence** function will check whether a value is defined for the given operator, and if not, the default value will be returned.

Sparrow is powerful enough that it allows these functions to be defined at library level, without changing the compiler. The implementation for these functions involves using macros and the Compiler API, which allow the programmer to manipulate the internal structure of the program at the Abstract Syntax Tree (AST) level [7].

4. EVALUATION

We presented the method through which one can construct flexible operator schemes in Sparrow. All Sparrow operators are defined in this manner, and their precedence and associativity values are set using the mechanism described above. In this section we evaluate the benefits and limitations of this approach.

Let us start with an example of an operator for raising a number to a given power:

```
fun pow(x, y: Double) = Math.pow(x, y);
fun **(x, y: Double)  = pow(x, y);
setOperPrecedence("**", getOperPrecedence("*") +
1); // higher precedence than multiplication
setOperRightAssociativity("**");              // right
associativity
cout << 4 * 3 ** 2 << endl;              // will print 36,
interpreted as 4 * (3**2)
cout << 4 ** 3 ** 2 << endl;         // will print 262144,
interpreted as 4 ** (3**2) == 4 ** 9
```

As illustrated, it is easy to define the operator and to set its precedence and associativity. Furthermore, the newly created operator can be integrated seamlessly with standard arithmetic operators.

Another example involves the use of **map** and **filter** operations, heavily used in the context of functional programming:

```
fun filter(range: Range, pred: AnyType) =
mkFilteredRange(range, pred);
fun map(range: Range, f: AnyType) =
mkTransformedRange(range, f);
fun .. (start, end: Number) = mkNumericRange(start,
end, true);
printRange( 1..10 filter isOdd map (fun n = n*2) );
// will print: 2 6 10 14 18
```

The last line of code contains some interesting features that are worth mentioning. Firstly, the expression **1..10** represents a numeric range [9] constructed using the operator **..**. Secondly, **(fun n = n*2)** is an anonymous function that returns the double of its argument. Finally, we can chain a sequence of infix operations (**..**, **filter**, **map**) that will make the entire expression natural to the programmer. If we had written the same expression without operators, similar to what one writes in a language like C++, it would have become:

```
printRange(map(filter(mkNumericRange(1,10),
isOdd), (fun n=n*2)));
```

Although this version is valid, it is much harder to read than what we obtained by using operators.

A representative example for this operator system concerns reference types (a concept similar to C++ references). In Sparrow, if one wanted to declare a reference to a type, say **Int**, it would write **@Int**. Even though this pattern is pervasive in Sparrow, this is actually a prefix call to the operator **@** that acts on a type. Due to its advanced metaprogramming capabilities (a feature called hyper-metaprogramming) [2], [10], [11], Sparrow allows the user to define functions (and operators) that act on types and can return other types. Thus, expressing a reference type becomes an operator call. Other examples of type operators in Sparrow are: **-@** (remove reference), **!@** (ensure one at least one reference) and **#$** (get a value of the given type).

All these examples show that by using this method of defining operators we can actually enhance the naturalness of the programming language.

4.1 Comparison with Other Languages

In order to compare Sparrow to other languages in terms of the capabilities of their operators, we have identified several specific features. The degree of support for these features provided by Sparrow and several mainstream programming languages is outlined in the following table:

	C++	Java	Scala	Haskell	Sparrow
Overloading existing operators	yes	-	yes	yes	yes
Infix and postfix operators with custom symbols	-	-	yes	yes	yes
Prefix operators with custom symbols	-	-	-	-	yes
Prefix operator with identifier	-	-	-	-	yes
Postfix operator with identifier	-	-	yes	-	yes
Infix operator with identifier	-	-	yes	yes	yes
Custom precedence and associativity	-	-	-	yes	yes

As we have explained in previous sections, Sparrow supports the complete set of features. The two functional languages come closest to supporting the full set, whereas C++ only supports overloading existing operators. Finally, Java does not allow any form of operator customization.

4.2 Limitations

Although this method of defining operators is flexible, extensible, and can improve code naturalness, it has some drawbacks.

The first drawback comes from the fact that the lexer allows all symbol sequences as valid tokens. For example the sequence of two characters ***-** is a valid token. This means that an expression like **a*-b** will be parsed as the infix operation ***-** applied to terms **a** and **b**. In languages like C++ and Java, this would be parsed as "**a** times the negative of **b**". Because the lexer always takes the longest possible sequence of characters when forming a token, ***-** will never be interpreted as two separate tokens. To solve this problem, the user must enter a space between the two symbols, transforming the expression into **a* -b**; in this case, the parsing will be similar to the one used in C++ or Java. In most cases however, even in C++, the user will place a space between the two symbols, or use parentheses, just to make the code easier to read.

Another limitation is related to postfix operators. In Sparrow, these can be used either as the last element in an expression, or inside parentheses. Expressions such as **a++ * b**, which is valid in C++ (post increment **a** and multiply the old value by **b**), are not valid in Sparrow. This expression is interpreted as "infix **++** between **a** and *****, and then postfix with an operation named **b**; this is invalid because the second operand must not be a symbol token. There is an easy fix for this, by placing the postfix call in parentheses: **(a++) * b**. Again, in typical scenarios, the programmer rarely applies the postfix increment inside expressions.

The presented method has an additional limitation, but this time not with respect to languages like C++, but to our initial goals. We wanted to allow the possibility of defining different precedence values for operators, depending on the types of the arguments. For example, one may desire a matrix multiplication to have a different precedence than an integer multiplication. Because applying infix operators yields different types based on the order in which the operations are applied, the choice of the highest precedence operator quickly becomes ambiguous. For example let us say that we have the expression **a + b * c**, with all three operands of different (class) types; let us assume that the **+** operation between **a** and **b** has precedence 10 and the ***** operation between **b** and **c** has precedence 15. Moreover let us assume that the precedence of **+** between **a** and the result of **(b*c)** is 20 and the precedence of ***** between the result of **(a+b)** and **c** is 5. In this case, no matter the order in which the operations are executed, it violates the precedence rules.

5. CONCLUSION

We presented a method of constructing flexible operator schemes at the library level. All Sparrow operators, their precedence, and associativity, are defined using this mechanism. Not only is this true even for basic operators (e.g. arithmetic), it also applies to the operator that creates a reference type: **@**.

The grammar rules for operators and expressions are simple and non-restrictive, allowing for a large variety of definable operators. Precedence and associativity can be easily configured using compile-time function calls.

In our evaluation we have provided several examples, showing that custom operators can be useful in a variety of scenarios. Furthermore, using our proposed method, they can be defined and used in a natural way. Despite some limitations, our approach is more powerful and flexible as it provides users with more customization options for their operators compared to other languages.

Finally, as mentioned before, one of our initial goals was to allow type-dependent precedence values. We intend to further investigate this issue and attempt to integrate this type of functionality into our current solution.

REFERENCES

[1] L. R. Teodorescu, A. Suciu, and R. Potolea, “Sparrow: Towards a New Multi-Paradigm Language,” *Analele Univ. Vest din Timisoara*, vol. 48, no. 3, 2011.

[2] L. Teodorescu and R. Potolea, “Compiler Design for Hyper-metaprogramming,” in *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2013 15th International Symposium on*, 2013, pp. 201–208.

[3] S. Marlow, “Haskell 2010 language report,” 2010.

[4] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*. 2005, p. 688.

[5] “Standard - the C++ language. Report ISO/IEC 14882:2011.” .

[6] M. Odersky, “The Scala Language Specification,” 2011.

[7] A. V Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[8] A. Koenig, “A Personal Note About Argument-Dependent Lookup,” *Dr. Dobb’s Bloggers*, 2012. [Online]. Available: http://www.drdobbs.com/cpp/a-personal-note-about-argument-dependent/232901443.

[9] A. Alexandrescu, “Iterators Must Go,” 2009.

[10] L. R. Teodorescu and R. Potolea, “Metaprogramming can be Easy, Fast and Fun: A Plea for Hyper-Metaprogramming,” *ACAM J. Autom. Comput. Appl. Math.*, vol. 21, 2012.

[11] L. R. Teodorescu, V. Dumitrel, and R. Potolea, “Moving Computations from Run-time to Compile-time: Hyper-metaprogramming in Practice,” in *Proceedings of the 11th ACM Conference on Computing Frontiers*, 2014, pp. 17:1–17:10.