Add The Map Literal Feature to Java Compilers by JavaCC

Shaoqing Yu , *syu702*

***Abstract*—The map literal feature, which is commonly used by some script languages, is on demand by more and more Java coders, who believe it can improve the code readability and coding convenience. This report is written for demonstrating an approach to add the map literal recognition ability to a Java compiler.**

***Index Terms*—Map literal, semantic analysis, symbol table, Java complier.**

I. Introduction

JAVA is a popular but still on-growing programming language, which has long been integrating new features, good or not, from others. Recently, as Java has become one of the main web development language, more and more coders, especially those who wrote scripts mainly before, hope the map literal can be supported by Java compiler. This report will analyse the reasons for this demand first, and then show an approach to add this feature by JavaCC.

Shaoqing Yu
April 11, 2016

II. Motivation

Compared with the map initialization in some script languages, such as Perl [3] and PHP [1], the map initialization in Java used to be viewed as a weakness as it is laborious and syntactically inefficient [6], [7]. The latest release, Java 8, tries to improve the map in-line initialization from JDK level by some new features called double brace initializer or static initialization [2], [4], [7], [8]. However, it does reduce a little bit work on map initialization but probably brings more performance trade-off and even potential errors in real practices [5], [9]. If the map literal feature can be supported from the compiling level, the optimized code will be pure Java, which has neither performance penalty nor potential fatal bug. Therefore, the map literal is a demanding feature which should be implemented on compiling level.

III. implementation

There are 2 steps in implementing the Map literal feature: the common semantic analysis and the map literal feature integration.

A. Semantic Analysis

The semantic analysis mainly consists of tracking class, method and variable declarations and their type checking. It can be roughly divided into 2 parts: the in-scope existence checking and the type compatibility checking. To determine the accessibility of identifiers and then analyse them in logics, a scope stack properly filled with symbol tables is required to be defined and generated first.

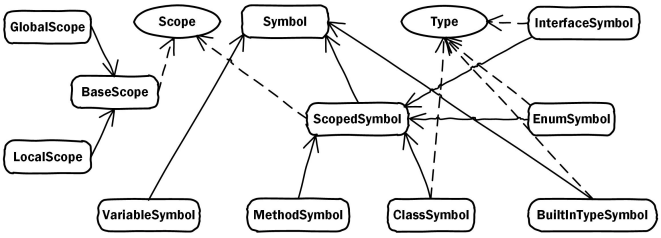

Fig. 1. The UML class diagram of the symbol table.

1) Symbol Table: The scope stack need to be properly organised by symbol tables filled with different symbols at each scope point. The classes required in the scope stack creation are showed in Figure 1. When a node associated with built in type, variable, enum, method, interface or class is visited during AST traversing, a symbol need to be created and loaded to the current symbol table. Sometimes if the symbol itself is a scope, for example the ClassSymbol, it will be pushed in to scope stack as a scope and be expanded at this point.

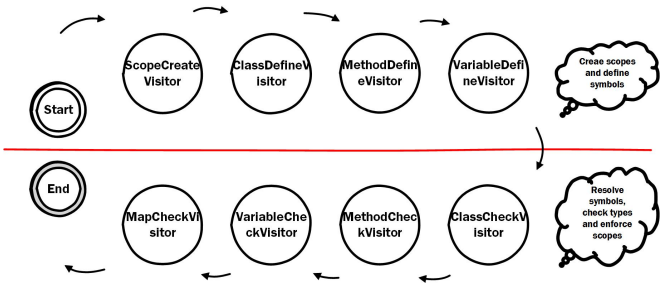

Fig. 2. The work flow diagram of visitors newly added.

2) Visitors: The actual operations on symbols in current scope are performed by visitors, which inject a particular action to each node in AST. The visitors will be executed

in the sequence showed in Figure 2. Accordingly to the 2 main parts of semantic analysis we mentioned before, they can be categorized into 2 groups to, firstly, create the scope stack (ScopeCreateVisitor) and define symbols (ClassCollectVisitor, MethodCollectVisitor, VariableCollectVisitot) when traversing the AST, and then check all the symbols in scope stack logically from different perspectives (class, method, variable and map).

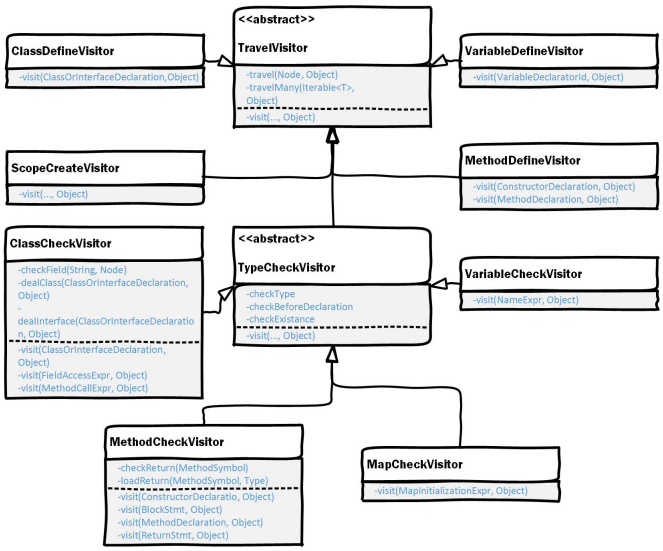

Fig. 3. The UML class diagram of visitors newly added.

The visitor class inheritance is designed as the Figure 3 demonstrates. To improve the re-usability of code, an abstract class named TravelVisitor is created to provide the AST traversing ability to all its child classes, unless they override their own visit functions. Each visitor inheriting from TypeCheckVisitor shares the basic type checking function to check the type compatibility of identifiers, while, applying it in different points (class, method, variable, map). The detailed descriptions of each visitor are listed below:

TravelVisitor It traverses through the AST by visiting all the nodes.

ScopeCreateVisitor Based on the traversing ability from TravelVisitor, it initializes the current scope of each node in AST, meanwhile creating and expanding a new scope if necessary.

ClassCollectVisitor Based on the traversing ability from TravelVisitor, it defines every class symbol and puts them to the proper scope when visiting ClassOrInterfaceDeclaration node in the AST.

MethodCollectVisitor Based on the traversing ability from TravelVisitor, it defines every method symbol and puts them to the proper scope when ConstructorDeclaration and MethodDeclaration are visited in the AST.

VariableCollectVisitor Based on the traversing ability from TravelVisitor, it defines every variable symbol and puts them to the proper scope when VariableDeclaratorId is caught in the AST.

TypeCheckVisitor Based on the traversing ability from TravelVisitor, it checks if the type of the target symbol matches the expecting type.

ClassCheckVisitor With the type checking ability from TypeCheckVisitor, it checks all the members (methods, fields and inheritance) in a class when traversing the AST.

MethodCheckVisitor With the type checking ability from TypeCheckVisitor, it checks return statements, constructors and methods when traversing the AST.

VariableCheckVisitor With the type checking ability from TypeCheckVisitor, it checks all the variable names in declarations and assign statements when traversing the AST.

MapCheckVisitor With the type checking ability from TypeCheckVisitor, it checks the size and types consistency of key-value pairs in the Map nodes when traversing the AST.

B. Map Literal checking

After the common semantic analysis has been achieved, the next is to add the new map literal feature.

```
Expression MapItem():
{
  Expression ret;
}
{
  (
    ret = Literal()
    | ret = Name()
  )
  {return ret;}
}
```

Fig. 4. The detailed code of MapItem.

1) Lexicon Level: At this level, firstly, a new production named MapItem, which is able to catch a literal or variable name and then return the expression, should be created like Figure 4.

```
//syu702 17:14
|
  "{"(key = MapItem()":"value = MapItem()) {    keys.add(key); values.add(value); }
  (","key = MapItem(){keys.add(key);}":"value = MapItem(){values.add(value);})* "}"
  {
      ret = new MapInitializationExpr(line, column,  scope,
      (ClassOrInterfaceType)type, typeArgs==null?new LinkedList():typeArgs,
       keys, values);
  }
//syu702 17:14
```

Fig. 5. The detailed code of modification about AllocationExpression.

Then, the AllocationExpression should be modified by adding a new option to parse out the map literal when the tokens associated with map literals are caught after ”new” key word. See main code in Figure 5.

2) Syntactic Level: Once the map literal is caught, a new AST node named MapInitializationExpr will be created and added to the AST. We can see from Figure 6 that it wraps a

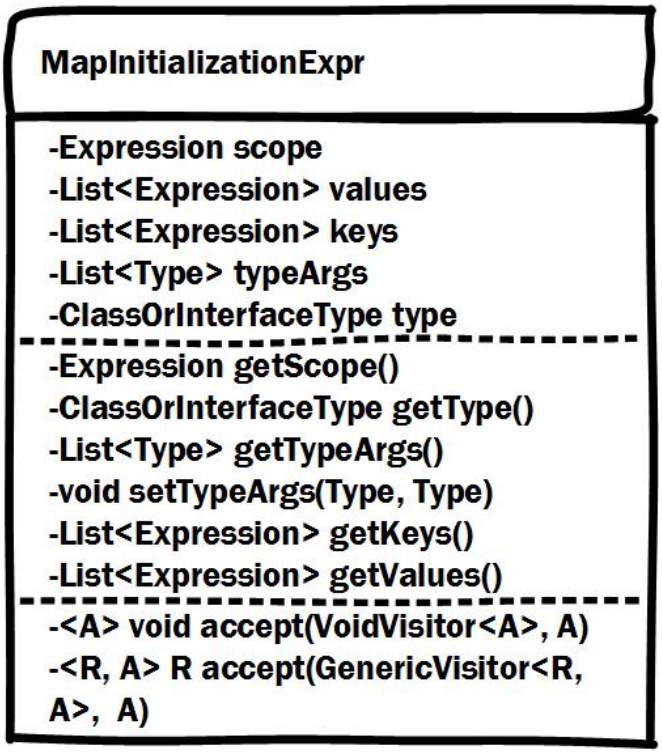

Fig. 6. The definition of MapInitializationExpr.

list of key and value expressions respectively, as well as its own expression type.

3) Semantic Level: As we claimed before, the type checking of HashMap is defined by TypeCheckVisitor and executed by VariableCheckVisitor as it inherits from TypeCheckVisitor and checks the variable type, no matter the variable is HashMap or something else. Even though, specific to the MapCheckVisitor, we still need to do some checking about the Map itself. This includes the size equality of values and keys, and the type consistency among either keys or values.

Here is still one more trick we need to play. The primitive type name can not be used directly in Map declaration, for example, the ”Integer” should be used instead of ”int” in the type argument of Map. We also need to convert the name of primitive type used in Map during type checking.

4) Display: In DumpVisitor, if MapInitializationExpr is recognised, the display way, including the printing of type, variable name and map literal, will be all delayed until calling the funtion ”visit(MapInitializationExpr , Object)” according to the original Java Grammar. The types of keys and values, no matter are explicitly declared before or implicitly detected here and filled in VarArgs, will be displayed as the map variable arguments in a pair of ”¡¿” after ”HashMap”. Then traverse through kay-value pairs to print Map.put(key, value) after the HashMap declaration.

5) Acheivement: Those points below are soundly tested by 13 cases in ”extentiontests” folder.

Map literal translation Map literals can be translated into pure Java code.

Multiple types Not ony primitive type, but also customized type can be used in map literals.

Division of map declaration and assignment Map can be declared as a member of a class and then assigned by a map literal.

Multiple map initalization inline Many variables of maps can be initialized by map literals inline once.

IV. Restriction

No in Class Members: The map literal cannot be used as a class member declaration because the compiler have no idea about where to put the callings of Map.put().

No in Return Statements: The map literal cannot be used in return expression directly without assigned by a variable.

Map & HashMap only: In this implementation, the map literal can only be used between Map and HashMap.

V. Conclusion

In this report, we have discussed the advantages of the Map literal, then described an approach to implement this new feature, as well as the partial semantic analysis in Java code. It is noted that the semantic analysis is a larger topic than what we discussed here and the real cases could be much more complicated especially if some object-orientation features are concerned. The map literal is a small but very useful feature which can reduce the repeated syntactic and improve code readability. By keeping integrating new features like this, Java will be more widely used as barriers of turning to Java from other script languages are being eliminating.

Acknowledgement

I would like to express my special thanks of gratitude to my lecturer Kelly, who is always patient to my stupid questions about JavaCC. If there exists any marker on this assignment, I think he/she definitely also deserves my thanks, because reading and testing such a massive code must be torturing and time-consuming.

References

[1] K. Tatroe, P. MacIntyre and R. Lerdorf, ”Arrays”, in *Programming PHP*, 7th ed. USA: O’Reilly Media, Inc., 2013, ch. 2, sec. 2, p. 26,27.

[2] K. Arnold, J. Gosling and D. Holmes, ”Static Initialization”, in *THE Java Programming Language*, 4th ed. Boston, USA: Addison Wesley Professional, 2005, ch. 2, sec. 5, p. 75.

[3] L. Wall, T. Christiansen and J. Orwant, ”Typing Hashes”, in *Programming Perl*, 3rd ed. USA: O’Reilly Media, Inc., 2000, ch. 14, sec. 3, pp. 378-384.

[4] www.c2.com, *Double Brace Initialization*, 2014, [online]. Available: http://www.c2.com/cgi/wiki?DoubleBraceInitialization. [Accessed: 11- Apr- 2016].

[5] jOOQ, *Dont be Clever: The Double Curly Braces Anti Pattern*, 2014, [online]. Available: https://blog.jooq.org/2014/12/08/dont-be-clever-the-double-curly-braces-anti-pattern/. [Accessed: 11- Apr- 2016].

[6] N. Buesing, *Inline initialization of Java Maps*, 2014, [online]. Available: https://objectpartners.com/2014/06/05/inline-initialization-of-java-maps/. [Accessed: 10- Apr- 2016].
[7] P. A. Minborg, *Java 8, Initializing Maps in the Smartest Way*, 2014, [online]. Available: http://minborgsjavapot.blogspot.co.nz/2014/12/java-8-initializing-maps-in-smartest-way.html. [Accessed: 10- Apr- 2016].
[8] N. Bansal, *Initializing Java Maps Inline*, 2009, [online]. Available: http://nileshbansal.blogspot.co.nz/2009/04/initializing-java-maps-inline.html. [Accessed: 11- Apr- 2016].
[9] StackOverFlow, *Efficiency of Java Double Brace Initialization?*, 2009, [online]. Available: https://stackoverflow.com/questions/924285/efficiency-of-java-double-brace-initialization. [Accessed: 11- Apr- 2016].