Implementing a Scala-like `match` statement in Java

Chang Liu cliu712 6752799

May 24, 2014

1 Motivation

It often occurs in Java programming that one has to perform certain tasks according to the class or interface type of a given variable. For example, the following is a snippet of the type checking code used in this assignment:

```
if (s instanceof MethodSymbol) {
    throw new SemanticsException();
} else if (s instanceof ClassSymbol) {
    return (ClassSymbol) s;
} else if (s instanceof VariableSymbol) {
    VariableSymbol v = (VariableSymbol) s;
    return v.get_type();
} else {    // unreachable code
    throw new InternalError();
}
```

We immediately discover a pattern here: shortly after checking that `s != null` (not shown in above code), we use a series of `if-else` statements and `instanceof` operators to check that the object given implements a certain class or interface type, and then proceed to process that object (under the assumption that it is of said class type). This is a common pattern in Java programming — idiomatic, one can say. It is however a rather clumsy one: one has to embark upon the dreadful labour of typing all the `if-else` clauses and even worse, manually performing the type casting (which is generally a discouraged programming practice).

However, we do have a better alternative than a series of `if-else` statements in Java: the `switch` statement. Wouldn't it be nice if we can replace the code listed above with a giant `switch` statement? Unfortunately, due to their C/C++ blood, the designers of the Java programming language decided to copy the `switch` statement, verbatim, as it is in C: `switch` can only match against integer variables. In other words, the variable symbol after the `switch` keyword has to be of an integer type, and the `case` labels have to be integer literals. This restriction is reasonable in the times of the C programming language, when the compiler techniques are not as advanced as they are today, and compilers generally have limited resources to perform overly complicated analysis. It is however an obsolete and destructive limitation today: by restricting the variable types in the `switch` statement, we placed artificial constraint that is hurting programmer productivity.

It is for exactly the same rationale that the designers of the Scala programming language decided to add a more liberal `switch` statement, term the `match` statement, that does generic pattern matching [1] in general, and typed pattern matching (what we want to replace the `if-instanceof-else` pattern with) in particular. A typical example of typed pattern matching, drawing from an Internet tutorial of this feature [2], looks like this:

```
def f(x: Any): String = x match {
    case i:Int => "integer: " + i
    case _:Double => "a double"
    case s:String => "I want to say " + s
}
```

Our goal in this assignment is to implement a similar `match` statement for Java.[1]

2 Syntax

The syntax for our `match` statement is as follows:

```
MatchStatement: "match" <IDENTIFIER> "{"
        ( MatchEntry )*
"}"

MatchEntry: ( ( "case" <IDENTIFIER> )
              | ( "default" )
            ) ":" BlockStatement
```

where `BlockStatement` is the non-terminal for block statements enclosed by `{}`.

The following is an simplified example of our feature. A more complete one can be found in the `StudentExample.javax` file.

[1] Note that the `match` statement in Scala can do much more than matching different types. What we implemented here is only a limited subset of the generic pattern matching in Scala.

```
Object b;
match b {
    case Window : {
        b.paint();
    }
    case Button : {
        b.set_inactive();
    }
    case RadioBox : {
        b.tune();
    }
    default : {
        System.out.println("Do nothing");
    }
}
```

When the above code is translated by our compiler, the following code is produced:

```
// GENERATED: Match statement
if (false) {}
else if (b instanceof Window){
    ((Window) b).paint();
}
else if (b instanceof Button){
    ((Button) b).set_inactive();
}
else if (b instanceof RadioBox){
    ((RadioBox) b).tune();
}
else {
    System.out.println("Do nothing");
}
// END OF MATCH STATEMENT
```

Note that all the reference to variable `b` is replaced by the appropriate type casting, say, `((Window) b)`.

3 Source-to-source Translation

The source-to-source part of the compilation process is done in the `TranslationVisitor`, after the semantics analysis of the input has been completed. This is the easier part of this assignment. Most of the visits done in the `DumpVisitor` are copied into the `TranslationVisitor` verbatim, with one important change: The visit functions for `MatchStmt` and `MatchEntryStmt` are modified to record the `(identifier, type)` pairs into a `HashMap`, and pass them into the visit function for `NameExpr`, which is similarly modified to print `((type) identifier)` when `identifier` is found within a match statement. The result of this translation process is that the `match` statement we implemented for `.javax` files are translated into standard java code, which is ready to be executed by a standard Java environment.

4 Semantics Analysis

The harder part of this assignment is the semantics analysis part of our compiler. As the Java language has a complicated and sometime subtle semantics, our analysis for the correctness of input is divided into four visitors: 1) `TypingVisitor`, which processes the class or interface definition, but leaves the inheritance declarations for, 2) the `InheritanceVisitor`, which processes the `extends` and `implements` keywords found within a class or interface definition. 3) `DefinitionVisitor`, which processes the field, method, and local variable definitions, and finally 4) `ResolvingVisitor`, which resolves accesses to member and local variable symbols.

Note that as soon as one visitor has completed its part of the semantics analysis, the information it gathered is readily available to the next stages of the compilation process. Therefore, `InheritanceVisitor` and `DefinitionVisitor` can check for undefined class or interface symbols. In other words, if the input has invalid reference to a class or interface name, it will be caught either in a `InheritanceVisitor` (if the reference happens in a `extends` or `implements` declaration), or in a `DefinitionVisitor` (if the reference happens in a member or local symbol definition).

The semantics work for `TypeingVisitor` is the easiest in terms of the semantics exceptions thrown: only one semantics exception is thrown in `TypingVisitor`. Besides building the symbol table for class or interface symbols, the only semantics it checks is multiple definitions of identifiers. In other words, if a class or interface name is defined twice in the input, this error will be caught in the `TypingVisitor`.

`InheritanceVisitor` builds upon the work done in the `TypeingVisitor`, and completes the class or interface symbol tables by adding inheritance information to the class or interface symbols. The semantics it checks are more complicated than `TypingVisitor`. In addition to checking the proper reference to class or interface symbols (i.e., cannot refer to undefined symbols), it check that classes type can only extend one superclass, as is required by the Java programming language (Java only allows single inheritance). It also checks that classes extend classes and interfaces extend interfaces, not the other way around. Finally, it checks that the `implements` keyword appears only in a class definition and not in an interface definition, and the name after it is an interface name

instead of a class name.

The bulk of the semantics checks are done in the `DefinitionVisitor` and the `ResolvingVisitor`. The `DefinitionVisitor` builds the symbol tables for method symbols and variable symbols. Method symbols are defined in `MethodDeclaration` nodes. Variable symbols may be defined in three places: in a field declaration, in the parameter list of a method, and in a variable definition expression as a local variable. However, these cases do not require separate treatment, as they all reuses `VariableDeclarator` nodes. To check for semantics errors, `DefinitionVisitor` only needs to check `MethodDeclaration` and `VariableDeclarator` nodes.

The primary semantics errors `DefinitionVisitor` checks are multiple definitions of method or variable symbols. For method symbols, a duplicated method has to not only match the method name, but also to match the method signature, as Java supports method overloading. For variable symbols, a local symbol can shadow a non-local symbol (symbols defined in class scope). However, it is a semantics error that a local symbol shadows another local symbol accessible from the current scope. The `DefinitionVisitor` handles these situations accordingly, by checking that `current_scope` implements `ClassSymbol` or not. If it does, this means we are in a class scope. Otherwise, we are in a local scope.

The fourth and final semantics visitor, the `ResolvingVisitor`, implements the interface `GenericVisitor<symtab.Type, Object>` instead of the `VoidVisitor<Object>` interface. The idea is that we want the visit functions for each expression to return a `symtab.Type`, so that when we process variable initialization or assignment statements, we can simply compare the left-hand side type and the right-hand side type. If the right-hand side type is not convertible to the left-hand side one, the input is in semantics error. The reason that we used this approach is that as the right-hand side expressions for assignment statements or variable initializations can be arbitrarily complicated, the type information have to be passed to parent nodes recursively to handle the most general cases.

The `ResolvingVisitor` does a similar processing for field access expressions and method invocation expressions, by processing and returning the type information recursively. In addition, `ResolvingVisitor` checks arithmetic expressions such as `+` and `-`. The checking here is however rather elementary: we only check that both sides of the binary expressions are compatible. We do not check for legitimacy of these operations. These checks can however be easily added: simply add more checks in the visit function for the `BinaryExpr` node. We see here the power of the visitor pattern: new features can easily be added to the existing code-base.[2]

All four visitors understands scopes. The `TypingVisitor` builds the basic class scopes: classes defined inside another class are placed inside the scope of said class. The `InheritanceVisitor` builds another kind of scopes: the inheritance scopes. The `DefinitionVisitor` places the field and method declarations in respective class scopes and the local variable definitions in the local scopes. It does so by creating a new `LocalScope` for each block statement, and add the scope object into the AST node. `ResolvingVisitor` retrieves this information, and descend into the scope to properly resolve symbol references.

The `ResolvingVisitor` also checks for proper semantics of our `match` statement: it checks that the types in each case labels are convertible to the type of the identifier provided after the `match` keyword. The reason for doing this checking is simple: if it is not convertible to the identifier's type, the case label will never be executed, and therefore may indicate a programmer's error.

5 Limitations

As this work is done in only half of the course work load, and therefore only one eighth of the total work load in the first term, our compiler is inevitably highly limited. For example, we ignore generic types such as `C<T>`. We also ignore enum types. We also ignore classes and interfaces defined inside a *method* (not another class), as they can only be final or abstract and this is a rare use-case anyway. However, these features can be easily added to our compiler. They are not implemented simply due to the lack of man-hours, not an inherent deficiency in our compiler design.

References

[1] The Scala Language Specification, M Odersky, 2013, Chapter 8, Pattern Matching

[2] Playing with Scala's pattern matching, `http://kerflyn.wordpress.com/2011/02/14/playing-with-scalas-pattern-matching/`

[2] Note that the `DefinitionVisitor` also implements a `GenericVisitor` instead of a `VoidVisitor`. However, it only makes limited use of the return values of the visit functions.