Individual Exercises - Lecture 11

1. (optional - but recommended) Do the exercises associated with the Tutorial on the Visitor Pattern (if you haven't already done so)
   You can consider looking at the ANTLR example included in the virtual machine. You can expand the grammar, and investigate the visitor classes that ANTLR generates from it. Try doing a simple visitor that just prints something for every node in the tree.

2. For the little language defined in Figure 7.14 design and implement in pseudo code AST classes based on the AST structure in Figure 7.15

```
 1  Start → Stmt $
 2  Stmt  → id assign E
 3        | if lparen E rparen Stmt else Stmt fi
 4        | if lparen E rparen Stmt fi
 5        | while lparen E rparen do Stmt od
 6        | begin Stmts end
 7  Stmts → Stmts semi Stmt
 8        | Stmt
 9  E     → E plus T
10        | T
11  T     → id
12        | num
```

Figure 7.14: Grammar for a simple language.

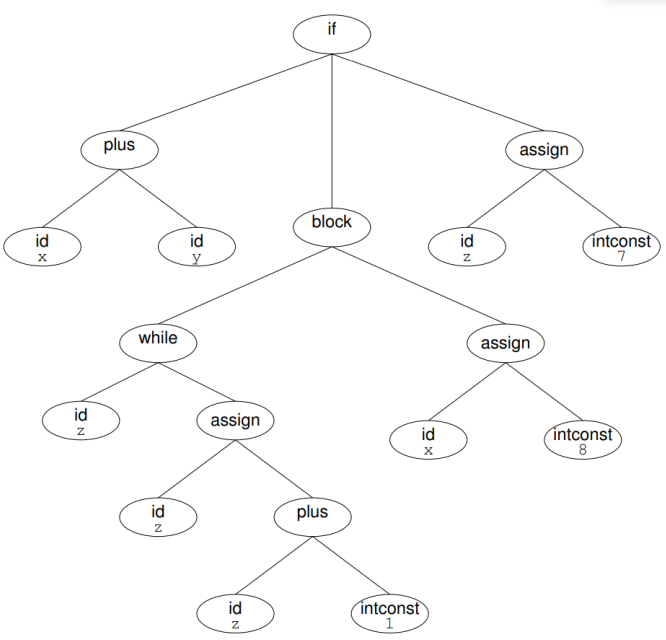
Example implementations for *AssigmentNode* and *WhileNode* in java.
Note that the classes could include an accept(); method to support use of the visitor pattern in java.

```
class AssignmentNode implements Node{
	private VariableNode variable;
	private ExpressionNode expr;

	public AssignmentNode(VariableNode variable, ExpressionNode expr){
		this.variable = variable;
		this.expression = expr;
	}
	@Override
	List<Node> getChildren(){
		return new ArrayList<>({predicate, loopBody})
	}
}
```

```
class WhileNode implements Node {
	Private PredicateNode predicate;
	Private LoopBodyNode loopBody;

	Public WhileNode(PredicateNode pred, LoopBodyNode loop) {
		this.predicate = pred;
		this.loopBody = loop;
	}
	@Override
	List<Node> getChildren() {
		return new ArrayList<>({predicate, loopBody});
	}
}
```

3. Construct an AST for the program in figure 7.18 (you may cheat by creating the AST based on figure 7.19)

   The program: If (x + y) { while (z) z = z + 1 od; x = 8 } else z = 7 fi $
   ('od' is just the symbol that marks the end of the while loop body in this grammar)



Important points:

- Structural symbols (such as '{', '(' ';' etc.) can be omitted as their meaning is represented implicitly in the structure of the tree.
- Several nodes can often be compacted into a single node (because we do not need information about how they were derived using the grammar)
  For example a series of nested Stmt nodes may be compacted into a single block node containing as many children as there are statements.
- You can use existing AST designs from other languages as inspiration for your own. For instance: https://astexplorer.net/

4. Add a pretty-printing method to each class in your AST structure from exercise 2 and pretty print the program from exercise 3

   One approach to pretty-printing is indenting nodes based on how deep into the tree they are nested. To print a tree in this fashion, one might pass an indentation string that increases in length for nested calls:

```
class WhileNode implements Node {
      ...
      public void prettyPrint(String indentation){
            printTabs(indentation);
            System.out.println("While Node : ")
            this.predicate.prettyPrint(indentation + "    ")
            System.out.println(); // New line
            this.loopBody.prettyPrint(indentation + "    ")
      }
}
```

5. Design a Visitor for pretty-printing for the little language defined in Figure 7.14 using the AST structure in Figure 7.15 and use it on the program from Figure 7.18

```
 1  Start  → Stmt $
 2  Stmt   → id assign E
 3         | if lparen E rparen Stmt else Stmt fi
 4         | if lparen E rparen Stmt fi
 5         | while lparen E rparen do Stmt od
 6         | begin Stmts end
 7  Stmts  → Stmts semi Stmt
 8         | Stmt
 9  E      → E plus T
10         | T
11  T      → id
12         | num
```

Figure 7.14: Grammar for a simple language.

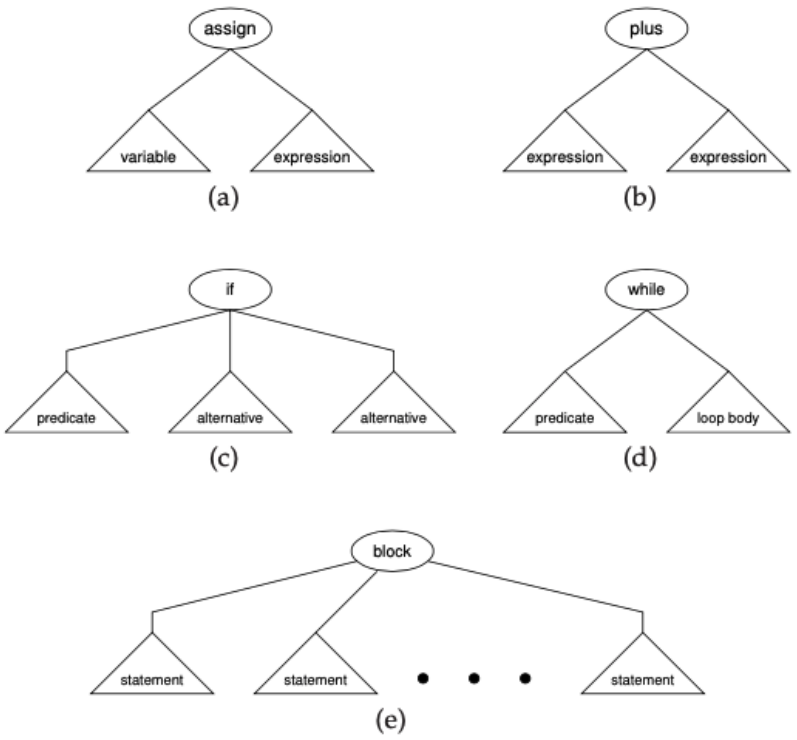


Figure 7.15: AST structures: A specific node is designated by an ellipse. Tree structure of arbitrary complexity is designated by a triangle.

This visitor might include a global variable to keep track of indentation. Upon visiting a node the indentationCounter is incremented, and upon finishing a visit method, the count is decremented. Below is a partial implementation of such a visitor:

```
class PrettyPrintVisitor extends ASTVisitor{
      private int indentation = 0;
      public void visit(IfNode ifNode){
             printTabs(indentation);
             System.out.println("IfNode");
             indentation++;
             visit(whileNode.predicate); // Visits
             visit(whileNode.loopBody); // Visits block node
             indentation--;
      }
      public void visit(WhileNode whileNode){
             printTabs(indentation);
             System.out.println("WhileNode");
             indentation++;
             visit(whileNode.predicate); // Visits
             visit(whileNode.loopBody); // Visits block node
             indentation--;
      }
      public void visit(BlockNode block){
            printTabs(indentation);
            System.out.println("BlockNode");
            indentation++;
            for(Node n : block.getChildren())
                visit(n);
            indentation--;
      }
      ...
}
```

The print result should be something like this:

IfNode
- PlusNode
  - x
  - y
- BlockNode
  - WhileNode
    - IdNode *(This is the while predicate)*
    - AssignNode *(This is the while body)*
      - IdNode (z)
      - PlusNode
        - IdNode(z)
        - Integernode(1)
  - AssignNode
    - IdNode(x)
    - IntegerNode(8)
- AssignNode
  - *IdNode(z)*
  - *Integer(7)*

6. Do Fischer et al exercise 18, 21 on pages 339-340 Note the word VISIT is missing! (exercise 18, 19 on pages 371-373 in GE)

   18. Write a visit method for handling the array type definitions represented by the AST in Figure 8.19(b), with upper and lower bounds included in array specification. Note that some type checking on the bounds expressions will be required in this method.

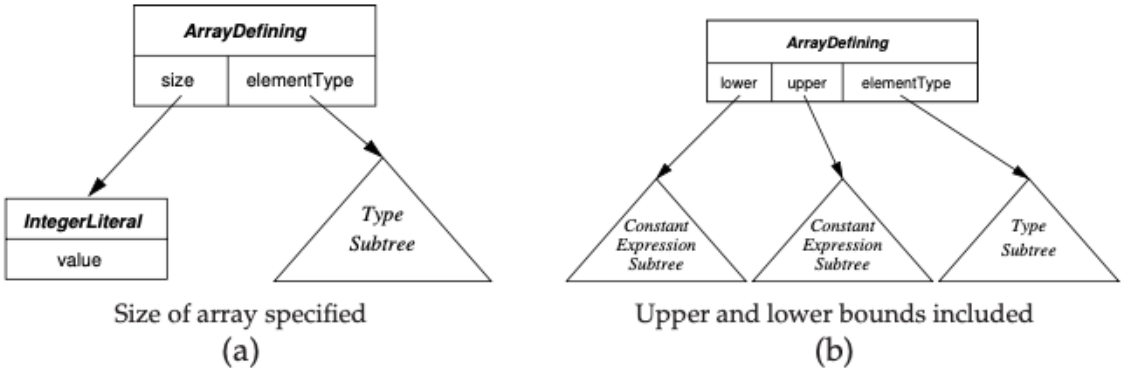


Figure 8.19: Abstract Syntax Trees for Array Definitions

```
class TypeVisitor extends ASTVisitor{
      ...
      Type visit(ArrayDefinition arrayDef){
            lowerBound = visit(arrayDef.lowerBound)
            upperBound = visit(arrayDef.upperBound)
            if(lower.type != integer || upper.type != integer)
                  throw new InvalidBoundTypeException();
             if(lower.value < 0 || upper.value < lower.value)
                  throw new InvalidBoundValueException();
             elementType = visit(arrayDef.elementNode)
             return new ArrayType(elemenType)
       }
       ...
}
```

21. Extend the visit method in Section 8.7.1 to handle the feature in Java that allows a list of interfaces implemented as part of a class declaration (as described at the end of that section). Interface declarations themselves are similar to class declarations. Write a visit method for processing interface declarations.

```
/*  Visitor code for Marker (14) on page 302                          */
procedure VISIT( ClassDeclaring cd )
    typeRef ← new TypeDescriptor( ClassType )                          (51)
    typeRef.names ← new SymbolTable( )
    attr ← new Attributes( ClassAttributes )
    attr.classType ← typeRef
    call currentSymbolTable.ENTERSYMBOL( name.name, attr )
    call SETCURRENTCLASS( attr )
    if cd.parentclass = null                                           (52)
    then cd.parentclass ← GETREFTOOBJECT( )
    else
        typeVisitor ← new TypeVisitor( )
        call cd.parentclass.ACCEPT( typeVisitor )
    if cd.parentclass.type = errorType
    then attr.classtype ← errorType
    else
        if cd.parentclass.type.kind ≠ classType
        then
            attr.classtype ← errorType
            call ERROR( parentClass.name, "does not name a class" )
        else
            typeRef.parent ← cd.parentClass.attributeRef               (53)
            typeRef.isFinal ← MEMBEROF( cd.modifiers, final )
            typeRef.isAbstractl ← MEMBEROF( cd.modifiers, abstract )
            call typeRef.names.INCORPORATE( cd.parentclass.type.names ) (54)
            call OPENSCOPE( typeRef.names )
            call cd.fields.ACCEPT( this )                              (55)
            call cd.constructors.ACCEPT( this )
            call cd.methods.ACCEPT( this )
            call CLOSESCOPE( )
    call SETCURRENTCLASS( null )
end
```

Figure 8.29: VISIT method in TopDeclVisitor forClassDeclaring

Extension to visit(ClassDeclaring cd):

```
...
cd.interfaces ← getRefToInterfaces()
for interface in cd.interfaces:
      if interface.type.kind != interfaceType
      then
         attr.classType ← errorType
      if not cd.methods.containsAll(interface.methods())
      then
         call ERROR("Class does not implement all methods from " +
         interface.name)
...
```

The visit(interfaceDecl) method is very similar to the visit method for class declaration. But recall that interfaces can only extend other interfaces. The following method skips the code duplicated from the visit(ClassDeclaring) Method, and shows only the check to verify that “parent classes” are interfaces:

```
procedure visit(InterfaceDecl iDecl)
       ...
       for parent in iDecl.parentClasses:
            if iDecl.parentclass.type.kind != interfaceType
            then
                attr.classtype ← errorType
                call Error(parentclass.name, "is not an interface")
        ...
```

7.Do Fischer et al exercise 1, 2 on page 385 (exercise 1, 2 on pages 417-421 in GE)

1. *Extend the semantic analysis, reachability, and throw visitors for if statements (Section 9.1.2) to handle the special case in which the condition expression can be evaluated, at compile time, to true or false.*

A detailed answer to this exercise is presented in Solutions Manual for Crafting a Compiler (can be found in general course materials on moodle):

```
procedure (IfTesting ifn)
    ifn.thenPart.isReachable ← true
    ifn.elsePart.isReachable ← true
    call C (ifn)
    constExprVisitor ← new ConstExprVisitor()
    call ifn.condition. (constExprVisitor)
    conditionValue ← ifn.condition.exprValue
    if conditionValue = true
    then
        ifn.elsePart.isReachable ← false
        ifn.terminatesNormally ← ifn.thenPart.terminatesNormally
    else
        if conditionValue = false
        then
            ifn.ifPart.isReachable ← false
            ifn.terminatesNormally ← ifn.elsePart.terminatesNormally
        else
            thenNormal ← ifn.thenPart.terminatesNormally
            elseNormal ← ifn.elsePart.terminatesNormally
            ifn.terminatesNormally ← thenNormal or elseNormal
end
```

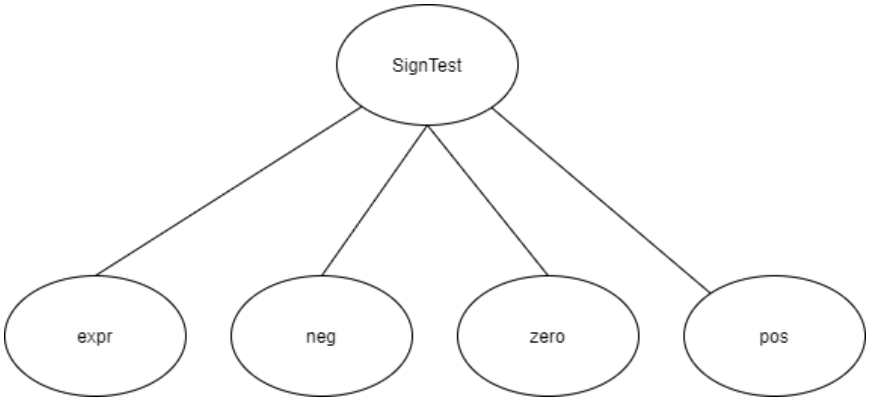
In summary, check if the boolean expression inside the if statement can be computed at compile time. For instance if the expressions consist only of constant values or literals. If that is the case branching is not required and can be optimized out.

2. *Assume that we add a new kind of conditional statement to C or Java, the signtest. Its structure is:*

Signtest (exp) {
neg: stmts
zero: stmts
pos: stmts
}

The integer expression exp is evaluated. If it is negative, the statements following neg are executed. If it is zero, the statements following zero are executed. If it is positive, the statements following pos are executed.

Show the AST you would use for this construct. Revise the semantic analysis, reachability, and throws visitors for if statements (Section 9.1.2) to handle the signtest.



A simple implementation of SignTest contains four children. One for integer sign expression and three for the stmts. It's possible to optimize this. For instance if the expression is a value we can compute at compile time, then we can leave out the 2 unused statement branches.

The SignTest is a special case of an if statement. Firstly, we must ensure that the expression, expr, evaluates to an integer. If the result of the expr is a signed integer, then we can mark all 3 statements paths as reachable. If the expr evaluates to an unsigned integer, we can mark the neg path as unreachable. Finally, we have the special case where expr can be evaluated at compile time, In this case we could determine which path would be taken at compile and mark the 2 others as unreachable.

The throws visitor remains the same as for an if statement.

Group Exercises - Lecture 11

1. Discuss the outcome of the individual exercises
   Did you all agree on the results?

2. (optional) Do Fischer et al exercise 26 on page 341 (exercise 27 on pages 373 in GE) (use the language you are defining in your project)

   26. Recall from Section 8.8 that special checking must be done for names that are used on the left-hand side of an assignment. The visitor class LHSSemanticVisitor was introduced for this purpose. Section 8.8.1 noted that since a parser will not allow a literal to appear as the target of an assignment, no visit method for handling literals was included in LHSSemanticVisitor.
   This example uses Java, you should use your own project language!

   a) *For a language of your choice, identify any other contexts in a program where an L-value is required.*

      Increment operators in Java, where for example 5++; is not a legal expression.

   b) *Do the syntactic rules of the language guarantee that a literal can not appear in these contexts?*

      No, the grammar actually allows literals(which are rvalues) in a postfix increment. The syntax rules specify that a postfix increment has the form:

      ```
      PostIncrementExpression:
        PostfixExpression ++
      ```

      Where a PostFixExpression is defined as:

      ```
      PostfixExpression:
        Primary
        ExpressionName
        PostIncrementExpression
        PostDecrementExpression
      ```

      And Primary is defined as:

      ```
      Primary:
        PrimaryNoNewArray
        ArrayCreationExpression

      PrimaryNoNewArray:
        Literal
        ClassLiteral
        this
        TypeName . this
        ( Expression )
        ClassInstanceCreationExpression
        FieldAccess
        ArrayAccess
      ```

      ^ From the java syntax specification found on:
      https://docs.oracle.com/javase/specs/jls/se15/html/jls-15.html#jls-PostIncrementExpression

c) *Write appropriate LHSSemanticVisitor visit methods for the AST nodes corresponding to any literals allowed by the language.*

In the **SemanticsVisitor** we create a method for visiting postfix increment nodes:

```
public void visit(PostFixIncrementNode incNode) {
	LHSSemanticVisitor lhsVisitor = new LHSSemanticVisitor();
	incNode.incrementTarget.accept(lhsVisitor);
}
```

Inside the LHSSemanticVisitor we create methods for the possible increment targets (considering only the increment operator):

```
public void visit(IdNode idNode){
	if (symbolTable.get(idNode.name).isFinal()) {
		throw new ConstantModifciationException(...);
	} else if (!symbolTable.get(idNode.name).supportsIncrement()) {
		throw new UnsupportedOperationException(...);
	}
	// You could also add more semantic checks,
	// fx. to check that the variable has been declared
}

public void visit(IntegerLiteralNode intLiteralNode){
	throw new InvalidLHSValueException(...);
}
// Repeat for other literal types such as float, doubles and strings
...
```

3. Do Fischer et al exercise 3, 7, 10, 13, 12 on pages 385-389 (exercise 3, 7, 11, 13, 14 on pages 417-421 in GE)

3. Assume we add a new kind of looping statement, the exit-when loop to C or Java. This loop is of the form:

loop

statements1

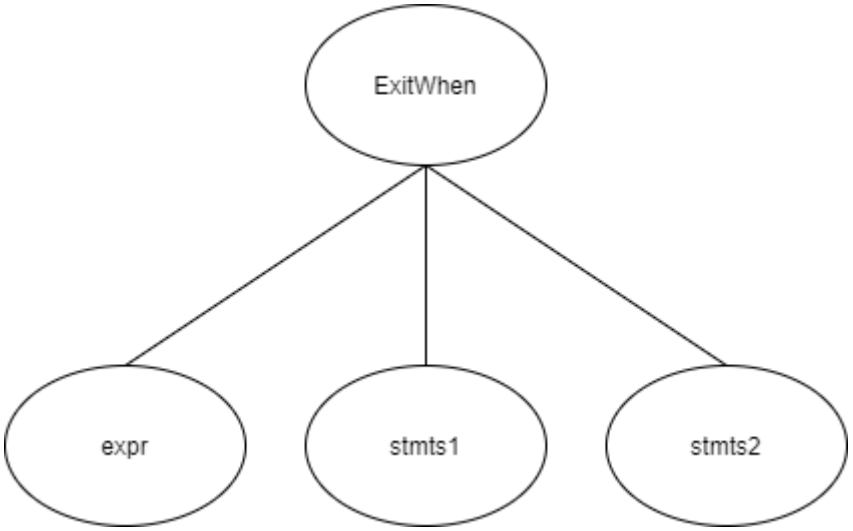
exit when expression

statements2

end

First, the statements in statements1 are executed. Then, expression is evaluated. If it is true, then the loop is exited. Otherwise, the statements in statements2 and statements1 are executed. Next expression is reevaluated and the loop is conditionally exited. This process repeats until expression eventually becomes true (or else the loop iterates forever). Show the AST you would use for this construct. Revise the semantic analysis,

reachability, and throws visitors for while loops (Section 9.1.3) to handle this form of loop. Be sure the special case of expression being constant-valued is handled properly



Once again we look at the special case where the expression can be evaluated at compile compile time. In this case where it evaluates to true, we can mark stmts2 as unreachable since it will always exit beforehand. If it is false at compile time, then both statements will be reachable.

7. Some programming languages, such as Ada, require that within a for loop, the loop index should be treated like a constant. That is, the only way that a loop index can change is via the loop update mechanism listed in the loop header. Thus (using Java syntax):

for (i=1;i<100;i++) print(i)

is legal, but:

for (i=1;i<100;i++) print(--i)

is not legal. Explain how to change the semantic analysis visitor of Section 9.1.4 to enforce read-only access to a loop index within a loop body.

Give the loop index variable a different type or flag in the symbol table (for example you can mark it as ‘read-only’). With such a flag it is easy to verify that no symbol marked ‘read-only’ is used on the left hand side. The semantic check has to make an exception by not checking the ‘read-only’ for the assignment operation to a loop index using the loop update mechanism in the loop where a specific loop index is declared.

10. One of the problems with the class structure used by Java and C is that field and method declarations (which are terse) are intermixed with method implementations (which can be lengthy and detailed). As a result, it can be hard to casually "browse" a class definition. As an alternative, assume we modify the structure of a class to separate declarations and implementations. A class begins with class declarations. These are variable and constant declarations (completely unchanged) as well as method headers (without method bodies). An "implemented as" section follows, which contains the bodies of each method declared in the class. Each method declared in the class must have a body defined in this section, and no body may be defined unless it has been previously declared. Here is a simple example of this revised class structure:

```
class demo {
     char skip = '\n';
     int f();
     void main();
implemented as
     f:    { return 10; }
     main: { print("Ans =",f(),skip); }
}
```

What changes are needed in the semantic analysis of classes and methods to implement this new class structure?

A detailed answer to this exercise is presented in Solutions Manual for Crafting a Compiler. (Can be found in general course materials on moodle)

In summary, the headers must be processed to create the symbol tables entries. then, the bodies can be processed with checks added to ensure the implementations match the definitions, that all declared methods are implemented, and that all implemented methods are declared.

13. As mentioned in Section 9.1.6, C, C++, and Java allow non-null cases in a switch statement to ”fall through” to the next case if they do not end with a break statement. This option is occasionally useful, but far more often leads to unexpected errors. Suggest how the semantic analysis of switch statements (Figure 9.21) can be extended to issue a warning for non-null cases that do not end in a break. (The very last case never needs a break since there are no further cases to ”fall into.”)

The SemanticsVisitor already visits all statements in each case by calling VisitChildren in visit(CaseItem cn). The method visit(CaseItem cn) could be extended to check if this is a non-null case (i.e. it contains statements) and if the more variable in the CaseItem AST node (Figure 9.20) is not null (i.e. this is not the last case). If both of these checks return true and the last statement in the case is not a break, a warning should be emitted that this is neither a none-null case nor is it the last case and as such should probably end with a break.

12. Most programming languages, including C, C++, C, and Java pass parameters positionally. That is, the first value in the argument list is the first parameter, the next value is the second parameter, and so on.

For long parameter lists this approach is tedious and error prone. It is easy to forget the exact order in which parameters must be passed. An alternative to positional parameters is keyword parameters. Each parameter value is labeled with the name of the formal parameter it represents. The order in which parameters are passed is now unimportant.

For example, assume method M is declared with four parameters, a to d. The call M(1,2,3,4), using ordinary positional form, can be rewritten as M(d:4,a:1,c:3,b:2). The two calls have identical effects; only the notation used to match actual parameters to formal parameters differs. What changes would be needed in the semantic analysis of calls, as defined in Figure 9.34, to allow keyword parameters?

A detailed answer to this exercise is presented in Solutions Manual for Crafting a Compiler. (Can be found in general course materials on moodle)

In summary, the primary task is to determine what method is being called given the method name and signature. For calls to methods with positional parameters only, their types and order is required, but for keyword parameters their names must also be taken into account. So to ensure a method can be called with a set of keyword arguments the semantic analysis must find all relevant methods (e.g., overloading allows multiple methods with the same name), and verify that the number of arguments, their types, and

their names match a unique method. Python is an example of a language that supports both positional and keyword parameters:

```
def pow(x, y):
    ...

positional = pow(2, 8)  # 256
keyword = pow(y=8, x=2) # 256
```