Chapter 11 Code Generation for a Virtual Machine

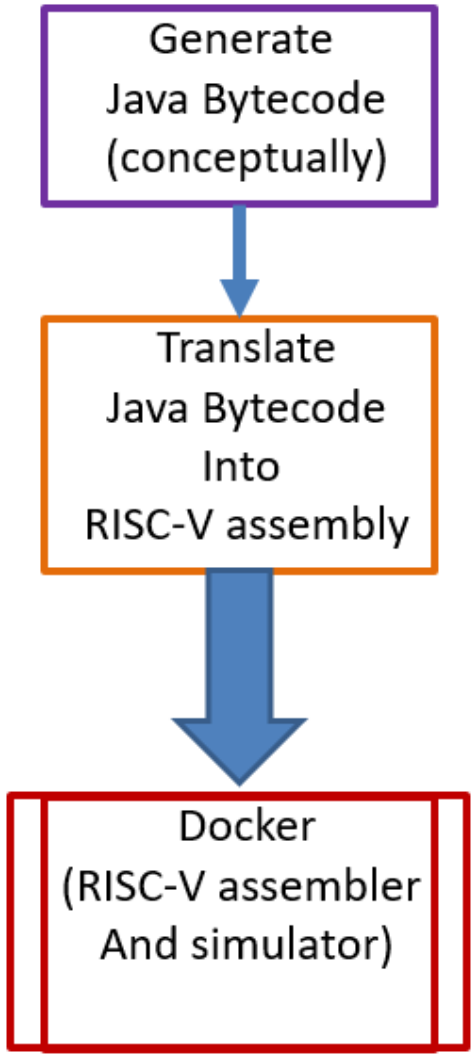
Wuu Yang

National Chiao-Tung University, Taiwan, R.O.C.

first draft: May 24, 2010

current version: June 16, 2022

RISC-V code geneartion



Example.

1. What is an exception?
2. How are exceptions are used by programmers?
3. How does exceptions interact with other language features?
4. The compiler needs to enforce all rules for exceptions.
5. The compiler needs to generate code according to semantics of exceptions.

Language features incldues: single ineritance, double inheritance, overloading, interface, inner class, anonymous class, type rules, implicit conversion, etc.

Transforming Java bytecode into RISC-V assembly

See the file “211-2CODE.GENERATION.pdf” for Java bytecode.

See the file “211-2JASMIN.pdf” for generating bytecode.

See the file “Java bytecode2RISCVasm20191031.pdf” (which has been posted in e3new).

Bytecode	RISC-V assembly

`emit(‘‘iadd’’);`	`emit(‘‘lw, a1, 0(s0)’’);`
	`emit(‘‘addi s0,s0,-4’’);`
	`emit(‘‘lw a2, 0(s0)’’);`
	`emit(‘‘addi s0,s0,-4’’);`
	`emit(‘‘add a3, a1, a2’’);`
	`emit(‘‘sw a3, 0(s0)’’);`
	`emit(‘‘addi s0,4’’);`

`emit(‘‘isub’’);`	`emit(‘‘lw a1, 0(s0)’’);`
	`emit(‘‘addi s0,s0,-4’’);`
	`emit(‘‘lw a2, 0(s0)’’);`
	`emit(‘‘addi s0,s0,-4’’);`
	`emit(‘‘sub a3, a1, a2’’);`
	`emit(‘‘sw a3, 0(s0)’’);`
	`emit(‘‘addi s0,4’’);`

Bytecode	RISC-V assembly

`emit(‘‘iload_#index’’);`	`emit(‘‘lw a3, 4*[#index](s1)’’);`
	`emit(‘‘sw a3, 0(s0)’’);`
	`emit(‘‘addi s0,s0,-44’’);`

`emit(‘‘istore_#index’’);`	`emit(‘‘lw a3, 0(s0)’’);`
	`emit(‘‘addi s0,s0,-4’’);`
	`emit(‘‘sw a3, 4*[#index](s1)’’);`

`emit(‘‘goto LABEL’’);`	`emit(‘‘j LABEL’’);`

`emit(‘‘ifeq LABEL’’);`	`emit(‘‘lw a2, 0(s0)’’);`
	`emit(‘‘addi s0,s0,-4’’);`
	`emit(‘‘beq a2,zero,LABEL’’);`

Chapter outline: Code Generation for a Virtual Machine

1. Visitors for code generators
2. Class and method declarations
3. The `MethodBody` visitor
4. `LHSVisitor`

Reference: `Jasmin Instructions.pdf`, `Jasmin User Guide.pdf`.

```
main() {
  AbstractSyntaxTree ast := parser(inputfile);

  SymbolTable symtab := buildSymbolTable(ast);

  SemanticsVisitor sv = new SemanticsVisitor();
  sv.Visit(ast);

  ReachabilityVisitor rv = new ReachabilityVisitor();
  rv.Visit(ast);

  ThrowsVisitor tv = new ThrowsVisitor();
  tv.Visit(ast);

  TopVisitor topv  = new TopVisitor();
  topv.Visit(ast);  // generate code
}
```

Figure 11.0 A very rough main program.

§11.1 Visitors for code generators

The code generator traverses the AST and generates code (usually for a virtual machine).

1. Generating code for a virtual machine is easier than for a real machine. A virtual machine could provide many high-level operations, such as virtual method calls, `String` data types, etc. Furthermore, we use labels instead of the actual addresses in Jasmin.
2. A virtual machine provides many resources, such as unlimited registers, and it may also help garbage collection.

Chapter 13 will discuss code generators for a real machine.

This code generator traverses the abstract syntax tree in 1 pass.

The code generator will make use of the visitor pattern.

We will have the following visitor classes:

1. `TopVisitor` processes class and method declarations.
2. `MethodBodyVisitor` generates code for constructs within a method body. In order to guarantee proper method termination, the visitor accepts a label where a method's postlude code will be put.
3. `LHSVisitor` generates code for the left-hand side of assignment statements. Passing reference parameters in Pascal and C++ bears some similarity to assignments. The `LHSVisitor` processes the portions of the AST that represent an address, rather than a value.
4. `SignatureVisitor` visits AST subtrees that represent a method definition or a method invocation and develops the signature of the method. The signature includes the name of the method and the types of all the parameters and the return value.

   When the code generator generates code to invoke a method, it needs the method's signature. The `SignatureVisitor` is activated

at this moment.

There is a `Visit` method for each AST construct (i.e., node) of interest.

In this chapter, the code generator generates JVM byte code (or JVM assembly code). A detailed explanation of JVM byte code is in §10.2.

§11.2 Class and method declarations

The `Visit` methods in the `TopVisitor` class for `ClassDeclaring` and `MethodDeclaring` process class and method declarations, respectively.

```
class NodeVisitor
   procedure VisitChildren(n)
      foreach c in n.GetChildren() do
         call c.Accept(this)
   end
end

class TopVisitor extends NodeVisitor
   procedure Visit(ClassDeclaring cd)
      // 11.2.1, p. 422
   end
   procedure Visit(MethodDeclaring md)
      // 11.2.2, p. 424
   end
```

```
end
```

Figure 11.1 Structure of the code-generation visitors, with references to sections in the textbook.

```
class MethodBodyVisitor extends NodeVisitor
   procedure Visit(ConstReferencing n)
      // 11.3.1, p. 425
   end
   procedure Visit(LocalReferencing n)
      // 11.3.2, p. 426
   end
   procedure Visit(StaticReferencing n)
      // 11.3.3, p. 427
   end
   procedure Visit(Computing n)
      // 11.3.4, p. 427
   end
   procedure Visit(Assigning n)
      // 11.3.5, p. 429
   end
   procedure Visit(Invoking n)
      // 11.3.6, p. 430
```

```
        end
        procedure Visit(FieldReferencing n)
            // 11.3.7, p. 432
        end
        procedure Visit(ArrayReferencing n)
            // 11.3.8, p. 433
        end
        procedure Visit(CondTesting n)
            // 11.3.9, p. 435
        end
        procedure Visit(WhileTesting n)
            // 11.3.10, p. 436
        end
    end
```

Figure 11.2 continued from Figure 11.1.

§11.2.1 Class declarations

There is an AST tree with root `ClassDeclaring` for each class declaration. The name of a class might be prefixed with the name of the package that contains the class. For anonymous and inner classes, the class name is generated systematically by the compiler, such as `myclass$001`. Sometimes the name of a class might also include the parametric type information, such as `Vector<int>` and `Vector<double>`. This is called *name mangling*.

The class names must be composed systematically and consistently since separately compiled classes might need to work together, i.e., referencing one another. Sometimes different compilers may need to follow the same naming rules so that they can work together.

Then the compiler emits information about the superclasses. This information is used for method and variable inheritance.

Java adopts single inheritance of classes while C++ adopts multiple inheritance. Code must be generated for inheritance.

The fields in a class are similar to those in a structure. A field includes a name, type, and access permissions. A compiler needs to calculate the size of the memory area for an instance of the class.

A class may also have static variables (i.e., class variables), which are shared by all instances of the class.

Finally, the compiler iteratively calls

```
node.Accept(this)
```

for each method in the class. Each node is a `MethodDeclaring` node. This `Accept` call triggers the following `Visit` call:

```
Visit(MethodDeclaring)
```

The compiler also needs to generate the code for initializing a class. Actions include allocating storage for static and instance variables, constructing the *virtual method table*, etc. See §13.1.3 on p. 496.

```
procedure Visit(ClassDeclaring cd)
   call EmitClassName( cd.GetClassName() )
   foreach superclass in cd.GetSuperClasses() do
      call EmitExtends(superclass)  // single/multiple inheritance
   foreach field in cd.GetFields() do
      call EmitFieldDeclaration(field)
   foreach static in cd.GetStatics() do
      call EmitStaticDeclaration(static)
   foreach node in cd.GetMethods() do  // node is MethodDeclaring
      call node.Accept(this)
end
```

§11.2.2 Method declarations

A `SignatureVisitor` will visit the method to create its signature. A signature includes the method name, types of parameters, and return type. A signature is used when a method is invoked. A method's name and signature is published.

Before code for the method's body is generated, a compiler would add *prelude code*. The prelude code is used to *establish the run-time context for executing the method*. This includes allocating space for a method's parameters and local variables and intermediate results. Note that there is a run-time stack because methods may call one another recursively.

(Note that in Fortran, there is no run-time stack. Why?)

Since JVM is a stack machine, a method also includes an *operand stack*. A compiler would need to calculate the size of the operand stack by analyzing the method's body. See Exercise 67 on p. 649.

A `MethodBodyVisitor` generates code for the method body. In doing so, the `MethodBodyVisitor` needs to know the end of method's body, which is denoted by the `postludeLabel`.

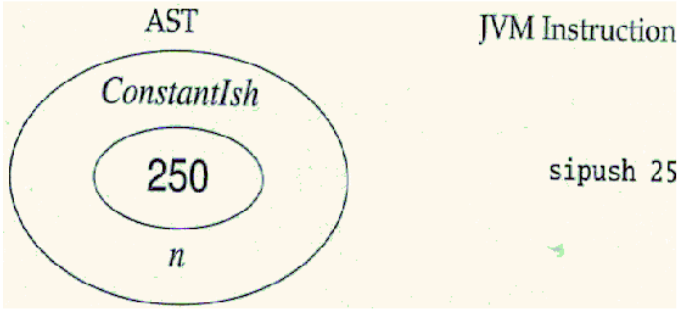
Before a method returns to its invoker, it may need to prepare for the return value, propagate an exception, or clean up the data structures for this invocation. Code for these actions are put in the method postlude.

```
procedure Visit(MethodDeclaring md)
    sigVisitor := new SignatureVisitor()
    call md.Accept(sigVisitor)
    signature := sigVisitor.GetSignature()
    call EmitMethodName(signature)
    call EmitMethodAlloc( md.GetLocals(), md.GetStack() )
    postludeLabel := GenLabel()
    bodyVisitor := new MethodBodyVisitor( postludeLabel )
    md.GetBody().Accept(bodyVisitor)
    call EmitMethodPostlude(postludeLabel)
end
```

§11.3 The `MethodBodyVisitor`

A `MethodBodyVisitor` generates code for each kind of construct in the method body.

§11.3.1 Constants



A program may reference a constant, such as 250. Small constants may be coded as an *immediate operand* in an instruction, such as

```
sipush 250
```

The `AllocLocal` call finds a place, such as a register or a cell in the run-time stack, to hold the constant. `AllocLocal` may involve *register allocation* in a one-pass compiler. (In JVM, we will use the stack.)

There might be multiple instruction sequences that could generate a given constant value. We will choose the one with the least execution time or the least space.

JVM includes four instructions for constants:

1. `bipush` for values that a single byte can hold.
2. `sipush` for larger values (2 bytes).
3. `ldc` and `ldc_w` (load constant)
4. `iconst_0`, `iconst_1`, `iconst_2`, `iconst_m1`, etc.

The `EmitConstantLoad` call will choose an appropriate instruction depending on the constant and the architecture of the machine.[a]

[a] All `emit` calls involve *instruction selection.*

```
procedure Visit(ConstReferencing n)
   loc := AllocLocal()         // destination of load
   n.SetResultLocal(loc)
   call EmitConstantLoad( loc, n.GetConstantValue() )
end
```

In Java, a method can reference three kinds of variables:

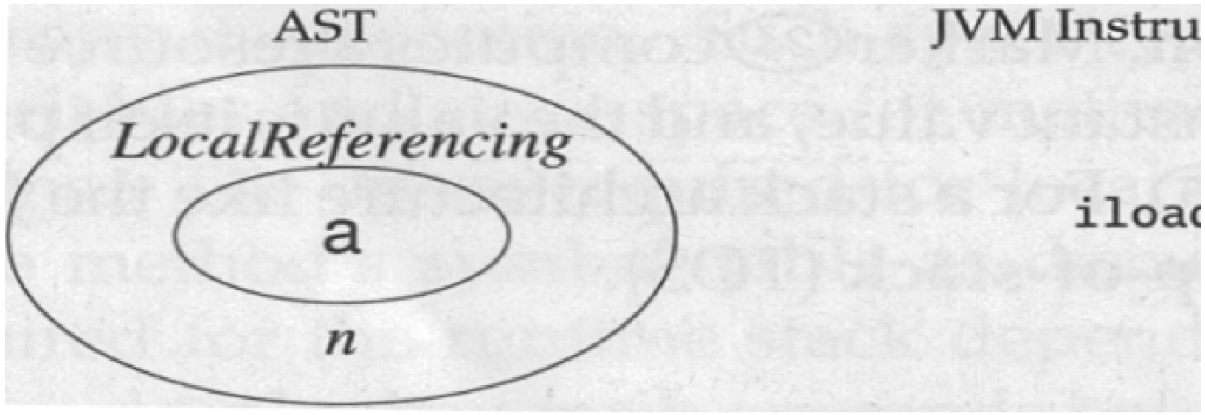
1. local variables (including parameters)
2. fields of an object (considered as global variables)
3. static variables of a class (considered as global variables)

Note that in Java as well as C, C++, and C#, procedures may not be nested. Variables are either local or global. Variable usage is simpler.

An example Java program with various variable usages:

```
class A {
  static int x;        // static variable
  float  y;            // field
  int foo(char z) {    // parameter
    double w;          // local variable
  }
}
```

§11.3.2 References to local storage



An instruction will be generated for fetching the value stored in the local storage into an accessible place, such as a register or a stack cell. In the above Figure, the location of the local variable `a` (which is also called a *virtual register*) is "5". The "`iload 5`" instruction loads its value into the top of the operand stack.

When building the symbol table, you need to find an empty slot for storing the local variable `a`. The empty slot could be on the stack frame or in the global area for RISC-V code. The compiler will remmber the offset of the slot in the frame or in the global area.

Additional Notes for RISC-V:

§1.

For code generation, start from methodbodyvisistor.

See the grammar.

We concentrate on
constant/literal
variable
expression
statement
calling a function
compiling a function

§2

variable addresss:

1. global varialbes: need the static address assigned by the compiler

2. local variables: need the offsets within the frame of the function in which the local variables are declared.

3. field of a structure: need the offset within the structure and the addresss of the structure.

4. element of an array: need the index and the address of the array.

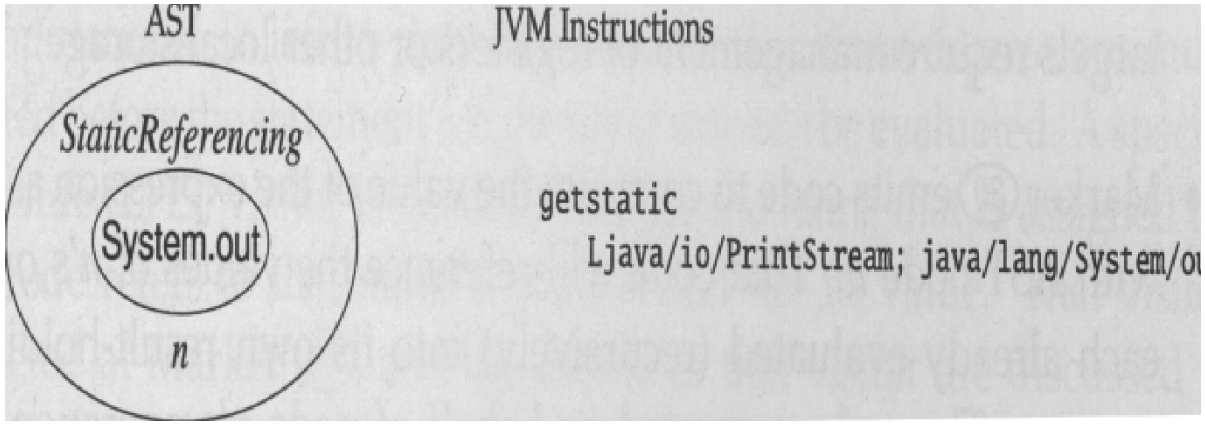
5. constant: need a static address

The static addressses (of global variables) and offsets of local variables and fields within a structure/array are computerd by the compiler.

The location of the local variable, “5”, is assigned by a visitor when it encounters a `LocalDeclaring` node. The size of the allocated space is usually 4 bytes in JVM, except that `double` and `long` need 8 bytes.

```
procedure Visit(LocalReferencing n)
   loc := AllocLocal()  // destination of the load instruction
   n.SetResultLocal(loc)
   call EmitLocalLoad( loc, n.GetLocation() )
end
```

§11.3.3 Static references



Here `StaticReferencing` means a more global variable. In Java, a static name is associated with a class type. For instance, the `java.lang.Sytem` class contains a static field `out`, whose type is `java.io.PrintStream`.

```
package java.lang;
class System {
    static java.io.PrintStream out;
    . . .
}
```

A Java compiler will generate a `getstatic` instruction:

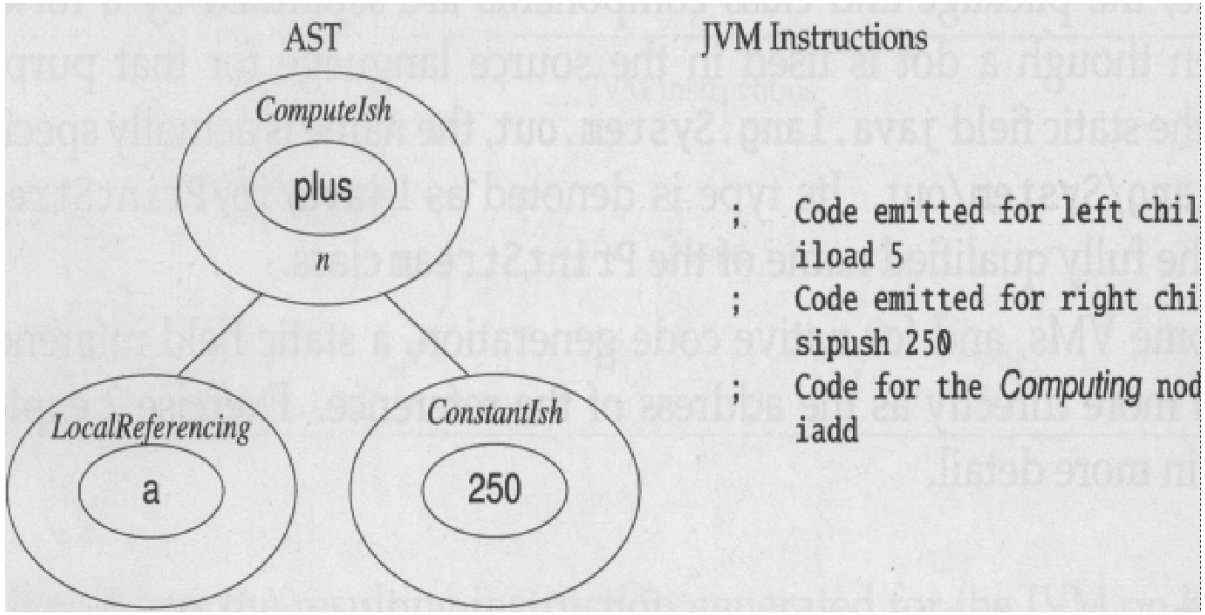
```
getstatic Ljava/io/PrintStream; java/lang/System/out
```

Note that the `getstatic` instruction uses the field name instead of generating code to calculate the address of the field. This saves a lot of trouble for the compiler. Address calculation is done by JVM.

When a compiler attempts to generate real machine code, it will need to compute the actual address of the static variable. Sometimes, the address calculation is done inside the compiler. Sometimes (e.g., an element in a static array), the compiler will generate code for address calculation.

```
procedure Visit(StaticReferencing n)
   call EmitStaticReference( n.GetType(), n.GetName() )
end
```

§11.3.4 Expressions



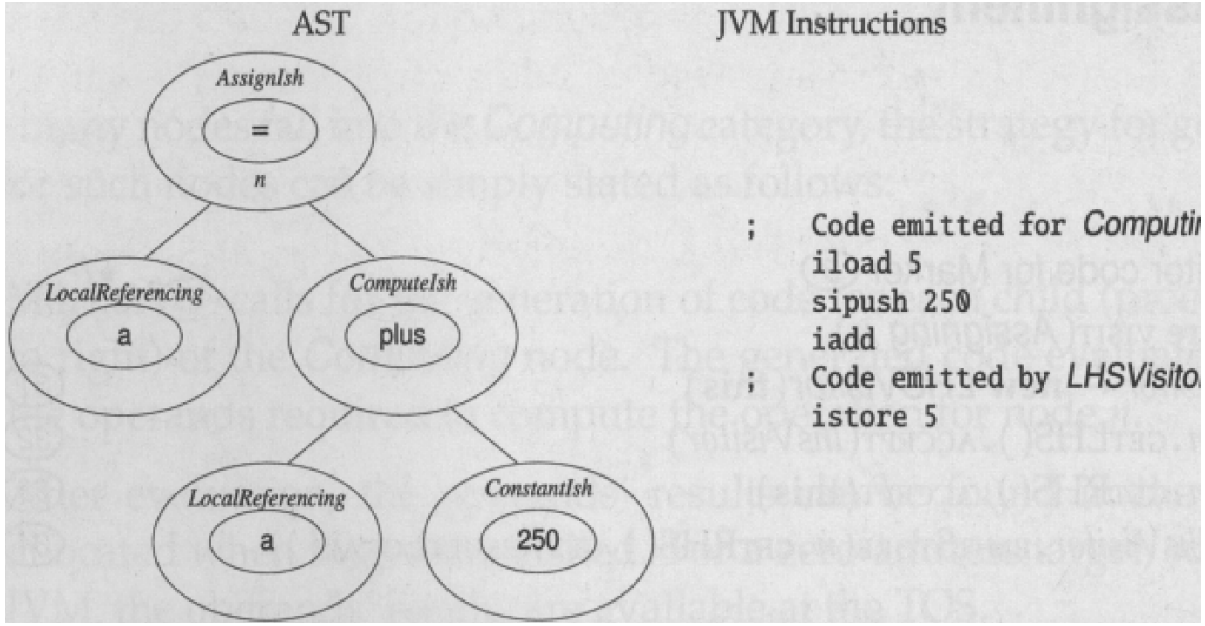
An expression consists of an operator and sub-expressions. A `Computing` node contains an operator. For an expression tree with a `Computing` node as the root, the `Visit` method first generates code for the left and right subtrees, allocates a memory cell (say in a register or in the run-time stack) for the result of the operation, and finally generates code for the operation at the `Computing` node.

Note that each `EmitOperation` call may generate one or more instructions.

In Java bytecode, the code for the two subtrees leaves the respective results on the top of the operand stack. Finally, an `iadd` instruction adds the two operands and leaves the final result on the stack top.

```
procedure Visit(Computing n)
   VisitChildren(n)          // two subtrees
   loc := AllocLocal()       // destination of final result
                             // this may need reg allocation
   n.SetResultLocal(loc)
   call EmitOperation( n )
end
```

§11.3.5 Assignment



For an assignment statement, the compiler first uses a `LHSVisitor` to generate code for the target of the assignment. The address of the target of the assignment is stored in the `LHSVisitor`.

Then a usual Visitor is used to generate code for the expression of the assignment. Finally, the call

```
lhsVisitor.EmitStore( n.GetRHS().GetResultLocal() )
```

generates a `store` instruction.

When the target of the assignment is an element of an array, we need to be careful that the index may fall out of the allowed range. Additional code may be generated to check the index during program execution.

```
procedure Visit(Assigning n)
   lhsVisitor := new LHSVisitor(this)
   call n.GetLHS().Accept(lhsVisitor)
   call n.GetRHS().Accept(this)
   call lhsVisitor.EmitStore( n.GetRHS().GetResultLocal() )
end
```

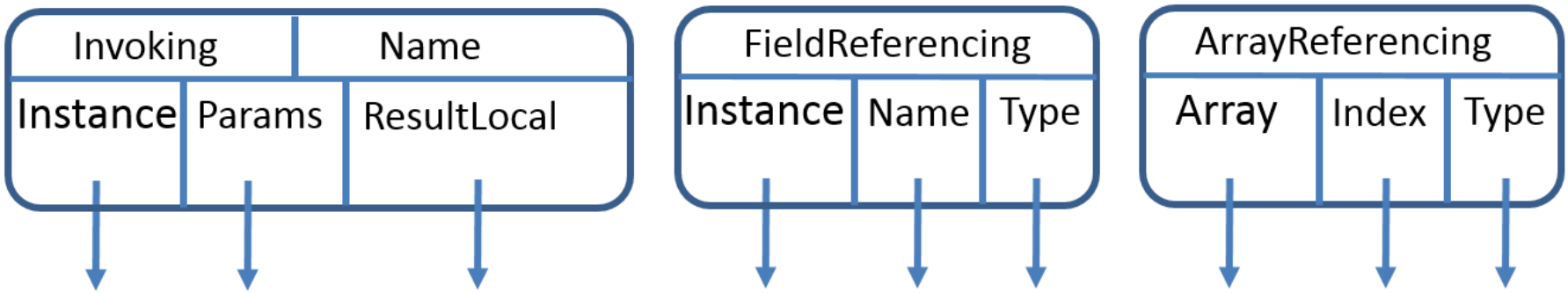


Figure 1: The Invoking, FieldReferencing, and ArrayReferencing nodes.

§11.3.6 Method calls

Calling a method is represented by an `Invoking` node. A `SignatureVisitor` is used to collect the arguments types and the return type.

The `FindSignature` call chooses an appropriate method to call. Overload resolution is applied in this `FindSignature` call.[a]

If the return value is not void, space is allocated for storing the result of this method invocation. For JVM bytecode, the result of a method invocation is left on the stack top. Thus, no space will be reserved in Java.

If the invoked method is not a static method, we need a *receiver*, such as

```
myCar.foo(x, y)
```

which serves as the first argument of the invocation. The

[a] Java adopts a two-stage dynamic dispatch. `FindSignature` in `Visit(Invoking)` is the 1st stage, which determines the signature of the invoked function. The 2nd stage is done during run time.

`n.GetInstance().Accept(this)` call finds out the receiver.

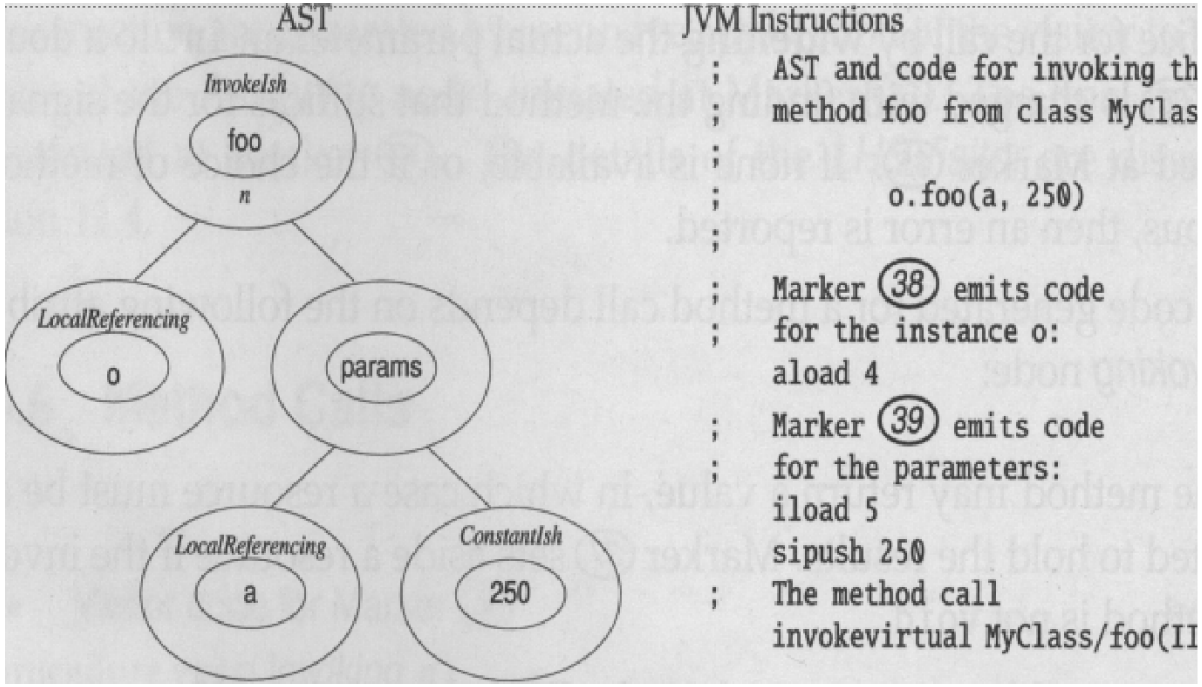
JVM provides different instructions for different kinds of invocations. The `invokevirtual` instruction is used to invoke virtual methods. (In contrast to C++, all Java methods are virtual methods.) Virtual methods involve the use of a *method table* for each class during dynamic dispatching.

```
procedure Visit(Invoking n)
   sigVisitor := new SignatureVisitor()
   call n.Accept(sigVisitor)
   usageSignature := sigVisitor.GetSignature()
       // overload resolution in FindSignature
   matchedSignature := FindSignature(usageSignature)
   if not n.IsVoid()  // return value is not void
   then loc := AllocLocal()          // destination of result
        call n.SetResultLocal(loc)
38 if not n.IsStatic() then call n.GetInstance().Accept(this)
39 foreach param in n.GetParams() do call param.Accept(this)
40 if n.IsVirtual()                 // Is it a virtual method?
   then call EmitVirtualMethodCall(n)
   else call EmitNonVirtualMethodCall(n)
end
```

The following figure shows the call:

```
o.foo(a, 250)
```

Note `o` is the first parameter to the invocation. The signature of `foo` is `(II)Z`.



§11.3.7 Field references

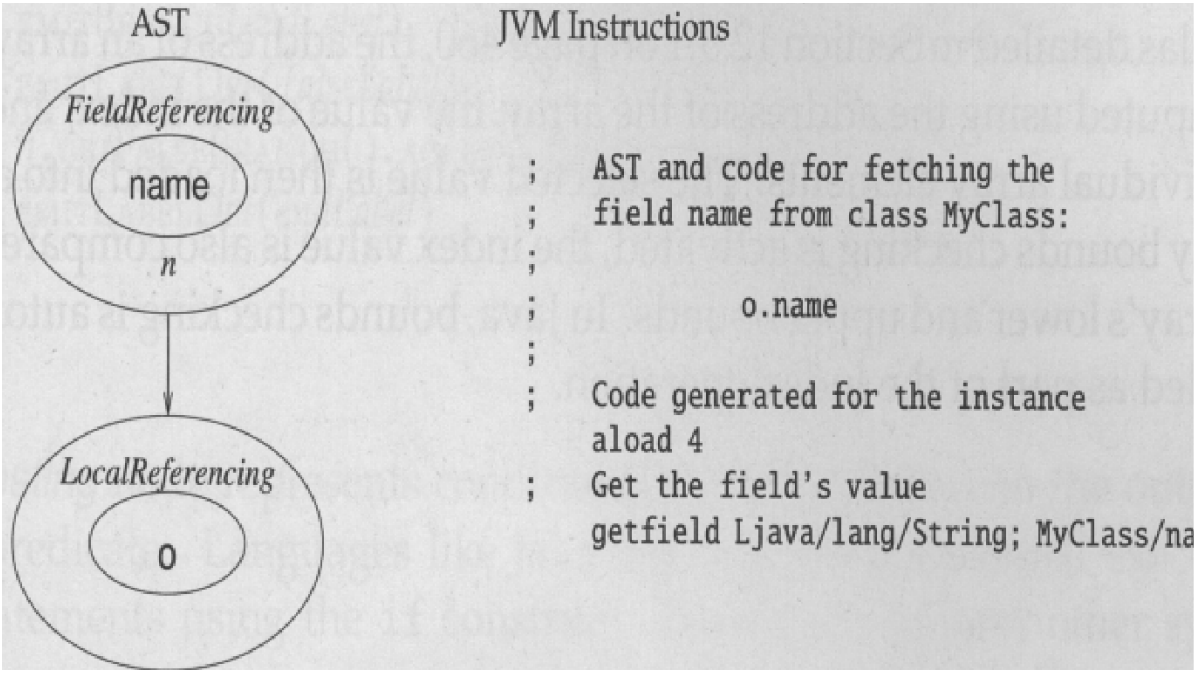
A reference to a field in an object resembles a static reference. The object is needed to resolve the reference. The object itself may be denoted as a simple local variable or a complex expression, which may include invoking other methods and referencing other fields and static references. The call

```
n.GetInstance().Accept(this)
```

resolves the subtree representing the object reference. Finally, a `getfield` instruction is generated, which is similar to the `getstatic` instruction.

```
 procedure Visit(FieldReferencing n)
41   call n.GetInstance().Accept(this)
     call EmitFieldReference( n.GetType(), n.GetName() )
 end
```

The following figure shows the example of `o.name`, where `o` is an instance of `MyClass` and `name` is of the `java.lang.String` type.



§11.3.8 Array references

An array reference consists of two portions: the name of the array and the index. In Java, the name is a reference to an array object while in C and C++, the name could be a global, local, or heap address. The call

```
n.GetArray().Accept(this)
```

resolves the name part of an array reference. For JVM, the object reference is pushed to the stack. For register-based machines, it is loaded into a register.

The index could be an expression. The call

```
n.GetIndex().Accept(this)
```

emits code to compute the value of the index. The value of the index is pushed into the stack (for JVM) or is loaded into a register (for register-based machines). Finally, an `iaload` instruction is emitted to load the value of the designated element of the array.

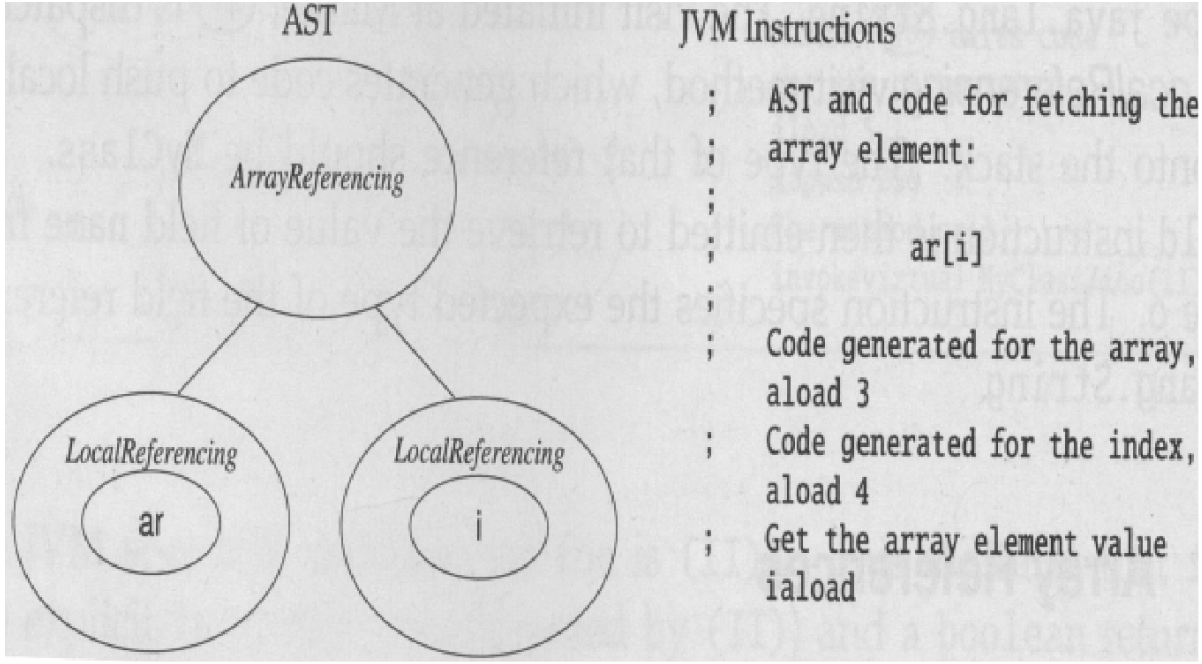
For other machine architectures, several instructions and various addressing modes may be needed to load the element. The compiler

needs to emit code that calculates the address of the element based on the address of the array, the element size, and the index value.

Additional instructions may also be needed for checking the index against the allowed range. An optimization in a compiler could try to eliminate unnecessary bound checks.

```
 procedure Visit(ArrayReferencing n)
42  call n.GetArray().Accept(this)
43  call n.GetIndex().Accept(this)
44  call EmitArrayReference( n.GetArray(), n.GetType() )
 end
```

In the following figure, code for `ar[i]` is generated. (Should be "`iload 4`".)



§11.3.9 Conditional execution

An `if` statement looks like

```
if exp then stmt1 else stmt2 end;
```

An `if` expression is similar:

```
(if exp then exp1 else exp2)
```

An `if` statement involves a predicate. The call

```
call n.GetPredicate().Accept(this)
```

generates code to evaluate the predicate. Then a conditional jump (to the *false* branch) instruction is generated.

The two calls

```
call n.GetTrueBranch().Accept(this)
call n.GetFalseBranch().Accept(this)
```

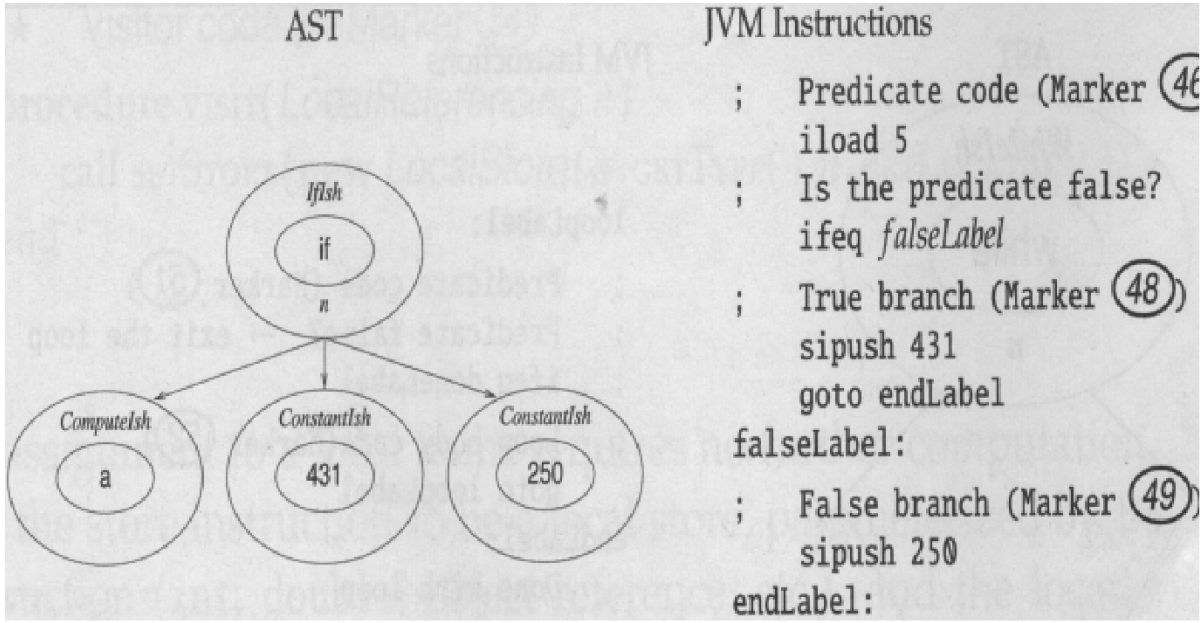
generate code for the *true* and *false* branches, respectively.

```
 procedure Visit(CondTesting n)
45  falseLabel := Genlabel()
    endlabel := GenLabel()
    call n.GetPredicate().Accept(this)   // evaluate predicate
46  predicateResult := n.GetPredicate().GetResultLocal()
47  call EmitBranchIfFalse(predicateResult, falseLabel) // jump
48  call n.GetTrueBranch().Accept(this)  //gen code for true branch
    call EmitBranch(endLabel)            //jump

    call EmitLabelDef(falseLabel)
49  call n.GetFalseBranch().Accept(this) //gen code for false branch
    call EmitLabelDef(endLabel)
 end
```

The following figure shows an example of an `if` expression:

```
if a then 431 else 250
```



The issue of *branch prediction* for modern computer architectures

The issue of *trace*

The placement of `true` and `false` branches

constant predicate

predicated instructions, such as ARM ISA

The code generated for an *if* statement is as follows:

```
    code for evaluating the predicate
    test the value of the predicate
    jump to falselabel if the predicate is false
    code for the true branch
    jump endLabel
falseLabel:
    code for the false branch
endLabel:
```

Note that we may switch the `true` and the `false` branches.

Note that we may even move the code for the `false` branch elsewhere to create a larger basic block.

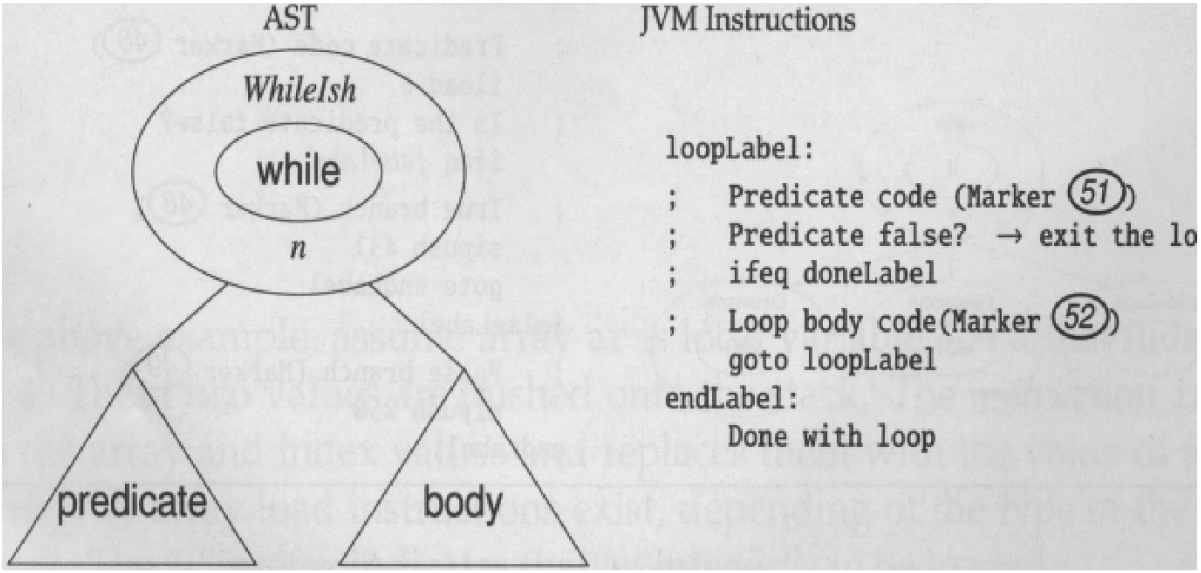
A compiler usually attempts to avoid code that jumps a lot (during run time) in order to take advantage of modern pipelined architectures. Thus, it carefully places the jump instructions.

§11.3.10 Loops

A loop is similar to an `if` statement. First code is generated for evaluating the predicate. Then a conditional jump (to the exit of the loop) instruction is generated. Then the code for the loop body is generated. Finally, an unconditional jump (to the predicate) instruction is generated.

```
 procedure Visit(WhileTesting n)
50  doneLabel := Genlabel()
    looplabel := GenLabel()
    call EmitLabelDef(loopLabel)
51  call n.GetPredicate().Accept(this)  // evaluate predicate
    predicateResult := n.GetPredicate().GetResultLocal()
    call EmitBranchIfFalse(predicateResult, doneLabel)  // jump
52  call n.GetLoopBody().Accept(this)  // code for loop body
    call EmitBranch(loopLabel)  // jump
    call EmitLabelDef(doneLabel)
 end
```

The following figure shows the AST for a `while` loop.



The code generated for a loop is as follows:

```
loopLabel:
   code for evaluating the predicate
   test the value of the predicate
   jump to doneLabel if the predicate is false
   code for loop body
   jump loopLabel
doneLabel:
```

§11.4 The LHSVisitor

For obtaining the left-value of an expression, the generated code will *not* load the value of the expression. Instead, it will generate code to evaluate the address of the result of the expression. However, it is not the case that all names appearing on the left side of an assignment statement are evaluated to an address. For example, in

```
a.b.c := 5
```

`a` and `b` will yield `load` instructions, while `c` will yield a `store` instruction.

The constructor of a `LHSVisitor` is passed the current `MethodBodyVisitor` (which is processing the current AST). The `MethodBodyVisitor` is used in the `LHSVisitor` when a right value is needed. The current `MethodBodyVisitor`, rather than a new instance, is used because it knows the current context of the method body, such as the postlude label.

For Java, the `LHSVisitor` will handle `LocalReferencing`, `StaticReferencing`, `FieldReferencing`, and `ArrayReferencing` nodes.

The `LHSVisitor` is used in the assignment (§11.3.5), for emitting a `store` instruction.

In contrast to the `MethodBodyVisitor`, the `LHSVisitor` does not issue a `load` instruction (nor a `store` instruction).

```
class LHSVisitor extends NodeVisitor
   constructor LHSVisitor(MethodBodyVisitor valueVisitor)
      this.valueVisitor := valueVisitor
   end
   procedure Visit(LocalReferencing n)
      // 11.4.1, p. 437
   end
   procedure Visit(StaticReferencing n)
      // 11.4.2, p. 438
   end
   procedure Visit(FieldReferencing n)
      // 11.4.3, p. 439
   end
   procedure Visit(ArrayReferencing n)
      // 11.4.4, p. 439
   end
end
```

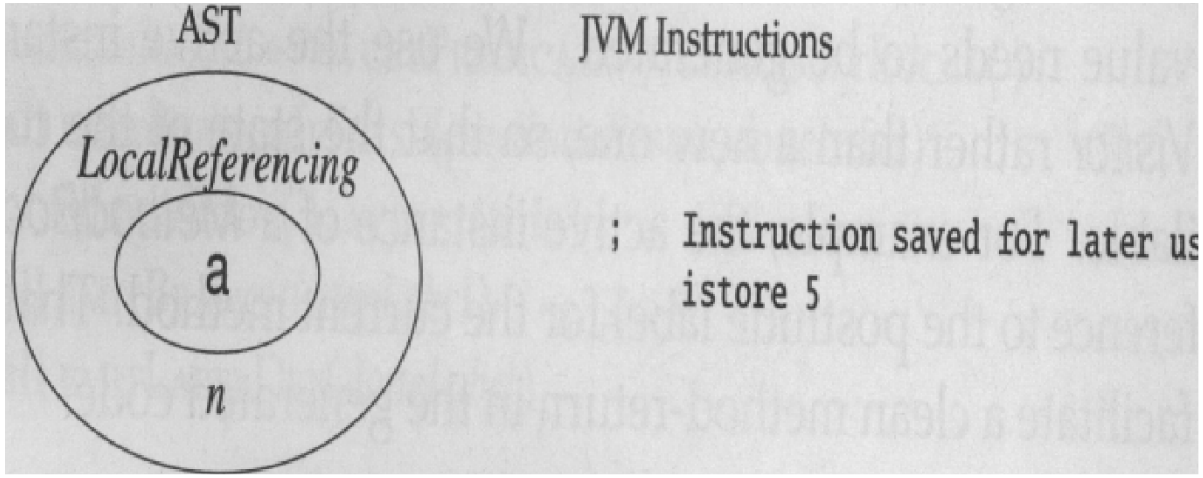
Figure 11.3 Structure of the LHSVisitor

§11.4.1 Local references

For an assignment to a local variable, the `SetStore` call remembers that a local `store` instruction, together with the type (int, double, or object reference, etc.) and the location of the target of the assignment will be issued later. The actual `store` instruction is generated in the `Visit(Assigning)` method (§11.3.5).

```
 procedure Visit(LocalReferencing n)
58  call SetStore( new LocalStore(n.GetType(), n.GetLocation()) )
 end
```

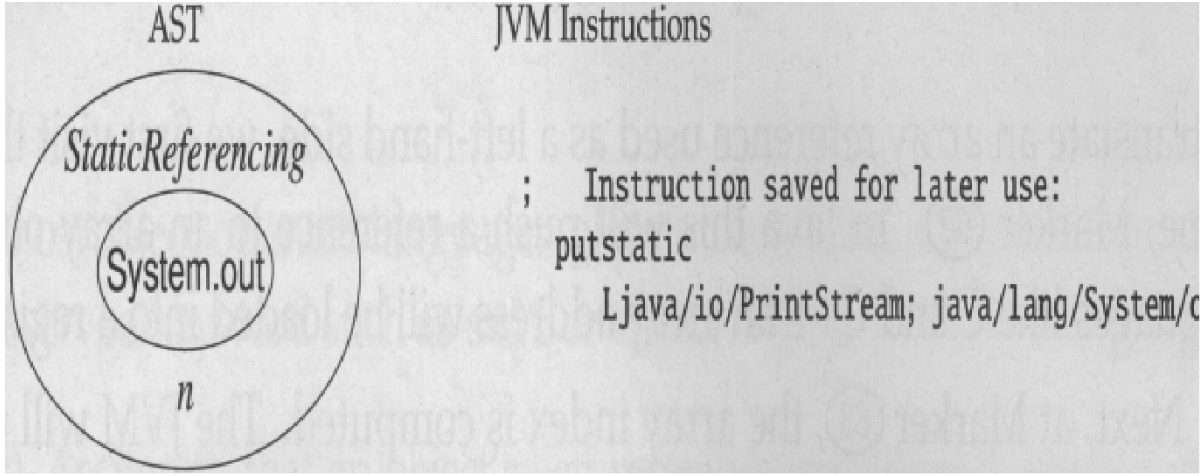
In the following example, an `istore` instruction will be issued later.



§11.4.2 Static references

Static references are similar to local references, except that a `putstatic` instruction will be issued. In Java, storing into a local variable and into a static variable will use different instructions. For other machine architectures, their addresses might be computed differently.

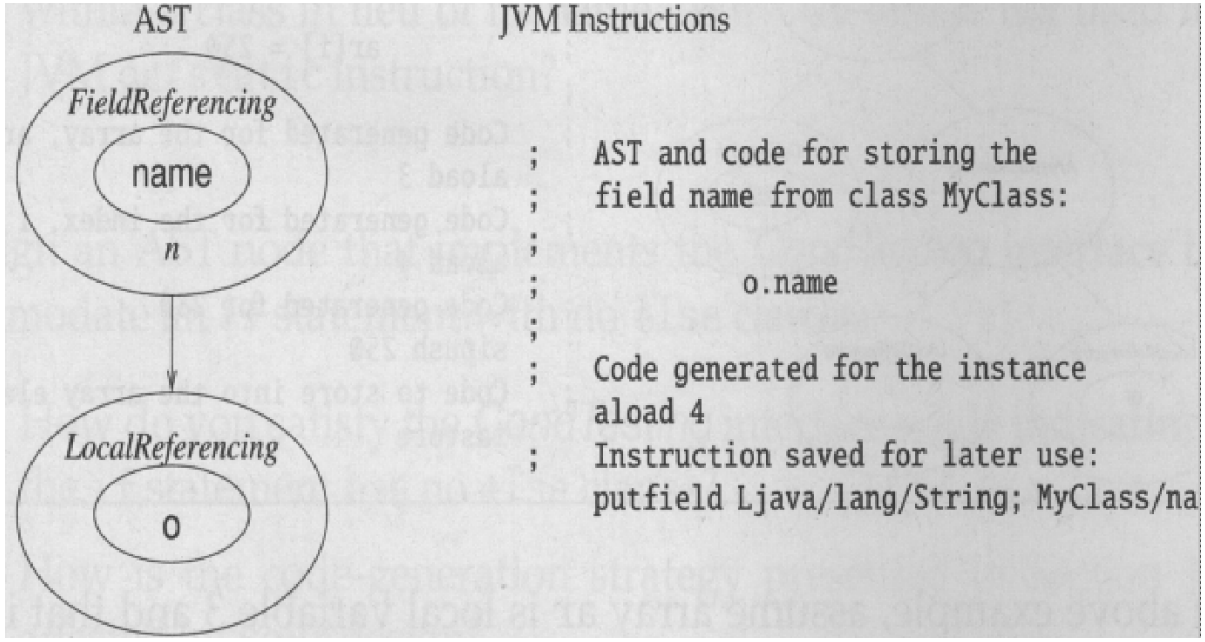
```
 procedure Visit(StaticReferencing n)
59  call SetStore( new StaticStore(n.GetType(), n.GetName()) )
 end
```



§11.4.3 Field references

A field reference will need a reference to the object first. It will emit a `putfield` instruction (for JVM).

```
 procedure Visit(FieldReferencing n)
60   call n.GetInstance().Accept(valueVisitor) //find the obj
61   call SetStore( new FieldStore(n.GetType(), n.GetName()))
 end
```



§11.4.4 Array references

The `Visit(ArrayReferencing)` method in the `LHSVisitor` class differs from that in the `MethodBodyVisitor` class in the last line of code:

```
call EmitArrayReference( n.GetArray(), n.GetType() )
```

vs.

```
call SetStore(new ArrayStore(n.GetArray(), n.GetType()))
```

JVM provides a special `iastore` instruction for storing an integer into an array element. This instruction implicitly includes bound checks of the index (by the JVM). For other processor architectures, several instructions are needed to perform similar tasks.

```
procedure Visit(ArrayReferencing n)
   call n.GetArray().Accept(valueVisitor) // r-value
   call n.GetIndex().Accept(valueVisitor) // r-value
   call SetStore( new ArrayStore(n.GetArray(), n.GetType()) )
end
```

The following figure shows an example of assignment to an array element.