SWEN430 - Compiler Engineering (2018)

Lecture 13 - Bytecode Generation

Lindsay Groves & David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

Program Translation

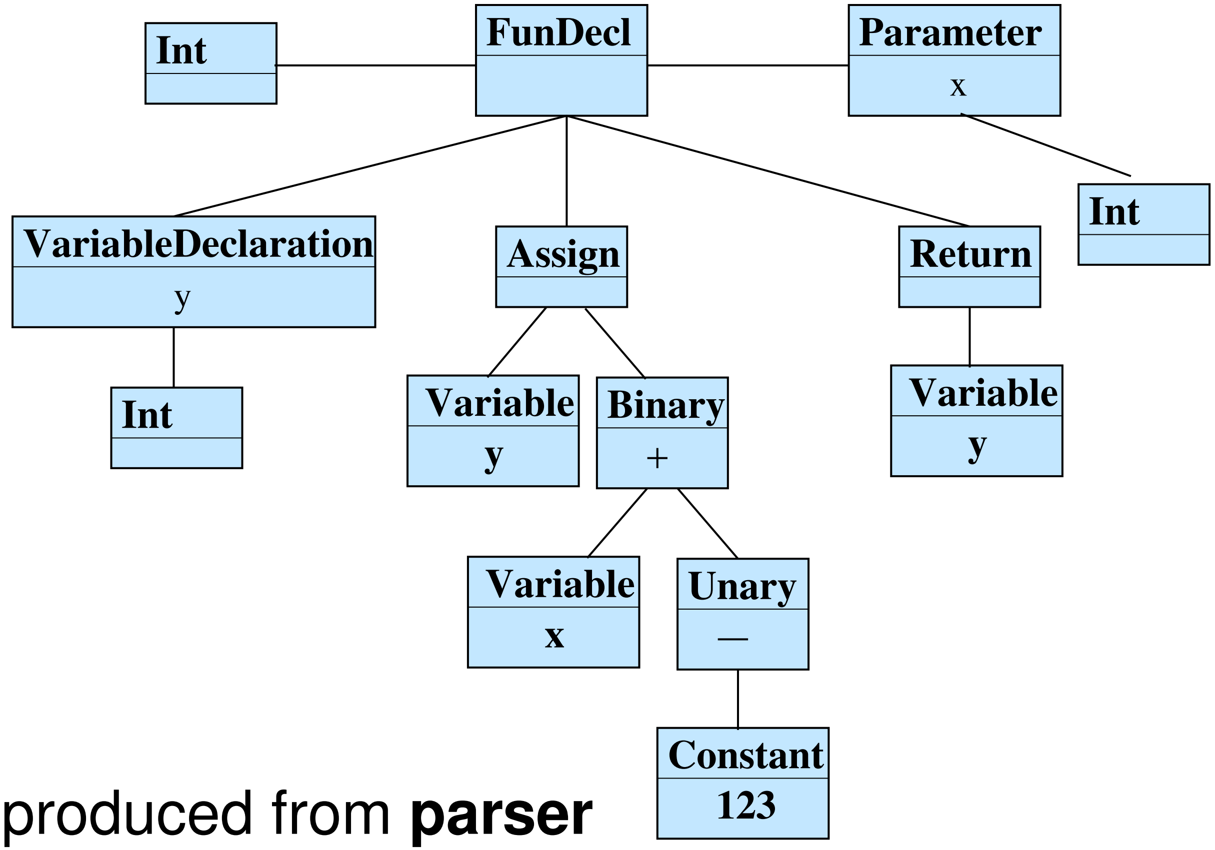
source program $\longrightarrow$ $\boxed{\text{compiler/ translator}}$ $\longrightarrow$ target program

- For any program in the source language, construct an “equivalent” program in the target language.
- Parse the source program and construct an AST.
- Traverse the AST generating target language code for each node.

$\Longrightarrow$ Need to design target code for each kind of AST node.

Abstract Syntax Trees (AST)

```
int f(int x) {
    int y;
    y = x + -123;
    return y;
}
```



- Abstract syntax tree produced from **parser**
- Abstract syntax tree used for e.g. **type checking**
- Abstract syntax tree turned into **intermediate language** or **target code**

Program Translation

- Target language may be:
  - another programming language (C, JS),
  - virtual machine code (JVM, CLR, LLVM),
  - assembler language,
  - machine code.
- Each presents different challenges for translation.

  E.g. translating to another programming language/assembler removes the need to determine addresses for variables and jumps.
- We'll consider JVM for now, machine code later.

Java Bytecode Example

```
class Test {
 public int f(int x) {
  int y = x * 2;
  return y + x;
}}
```

```
public int f(int);
  Code:
   Stack=2, Locals=3
   0:    iload_1
   1:    iconst_2
   2:    imul
   3:    istore_2
   4:    iload_2
   5:    iload_1
   6:    iadd
   7:    ireturn
```

How do we get from AST to bytecode?

http://homepages.inf.ed.ac.uk/kwxm/JVM/codeByFn.html
lists bytecodes by function.

Program Translation

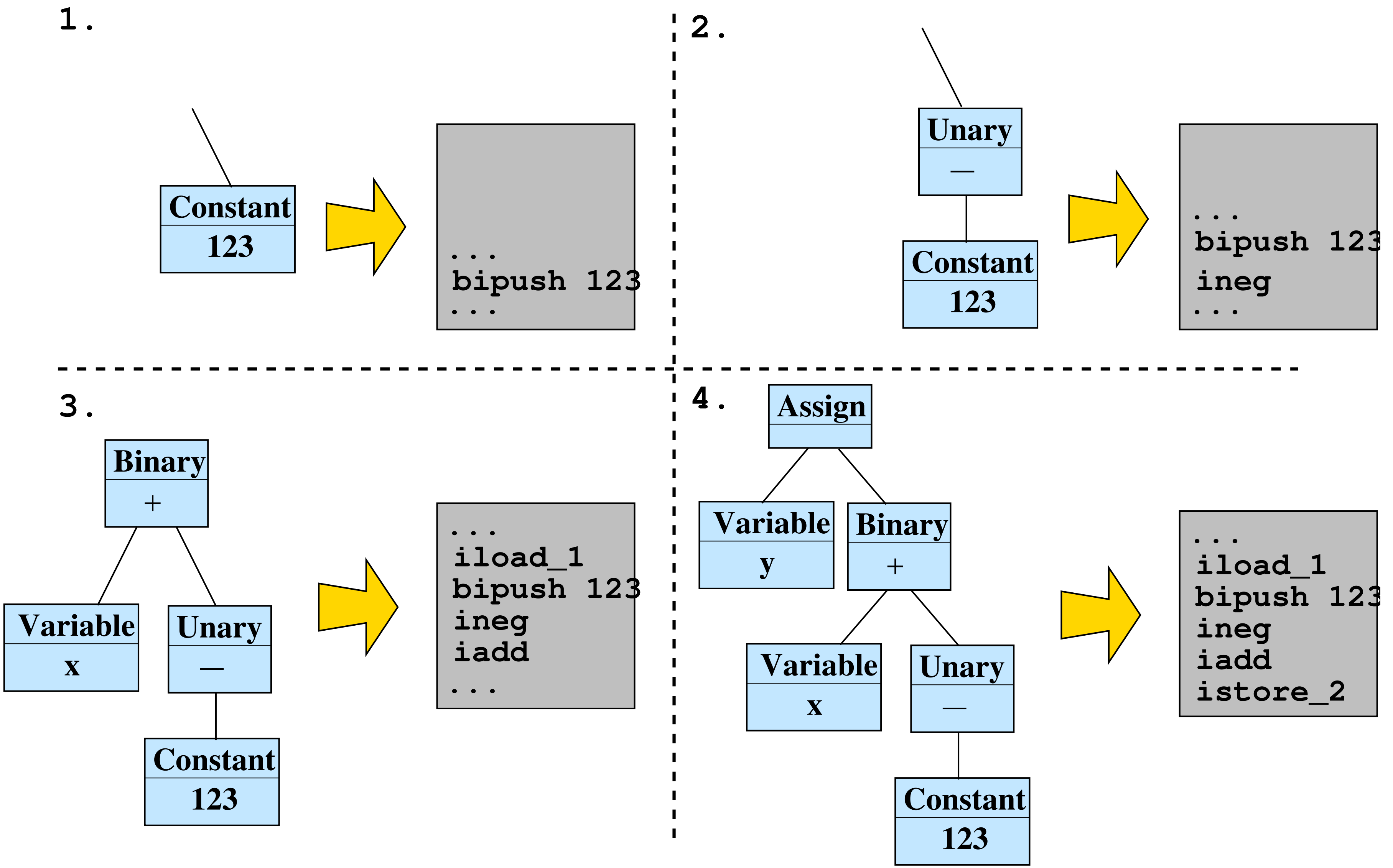
Aspects to consider:

- Allocate storage for variables, and assign addresses to variable names.
- Accessing/updating components of arrays/records/strings/objects.
- Computation: appying arithmentic/logical/relational operators.
- Flow of control: branching for conditionals, loops, etc.
- Linkage for subroutine/procedure/method/function calls.
- Dynamic storage/garbage collection, I/O, interface with operating system.
- Concurrency?
- ...

Storage Allocation

- Allocate space for local variables in a stack frame for that method.
  Keep track of next available location and record in symbol table.
  E.g. in above example: $\mathtt{x} \to 1$, $\mathtt{y} \to 2$.
- Need to determine how much space each variable takes — mapping source language types to target data types.
  How much space does each Java type use?
- If you can't work out the size at compile/load/call time, allocate space on the heap.
  E.g. Java arrays go on the heap, C arrays go on the stack.
- Need to determine scope/lifetime of variables, so storage can be reused.
- Can work out how much space is needed for a method at point of call.

Basic Calculations



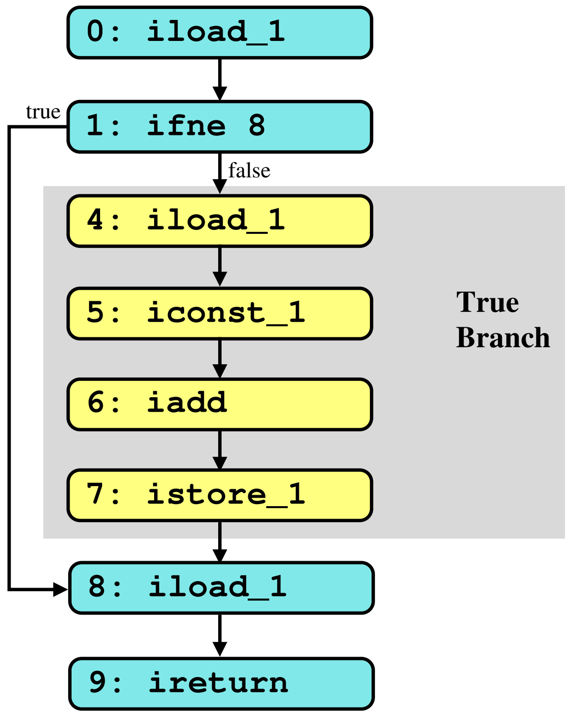
Generating Simple Bytecodes

- Often several choices of bytecode:

Bytecode	Format	Description
`iload`	[1 byte op][1 byte X]	*Push local variable X*
`iload_0`	[1 byte op]	*Push local variable 0*
`iload_1`	[1 byte op]	*Push local variable 1*
...		
`istore`	[1 byte op] [1 byte X]	*Pop stack to local variable X*
`istore_0`	[1 byte op]	*Pop stack to local variable 0*
`istore_1`	[1 byte op]	*Pop stack to local variable 1*
...		
`bipush`	[1 byte op] [1 byte]	*Push int constant (-128...+127)*
`sipush`	[1 byte op] [2 bytes]	*Push int constant (-32768...+32767)*
`ldc`	[1 byte op] [1 byte idx]	*Push int constant from constant pool*
`iconst_0`	[1 byte op]	*Push int zero*
`iconst_1`	[1 byte op]	*Push int one*
...		

Translating If-Statements

```
int f(int y) {
 if(y == 0) {
  y = y + 1;
 }
 return y;
}
```

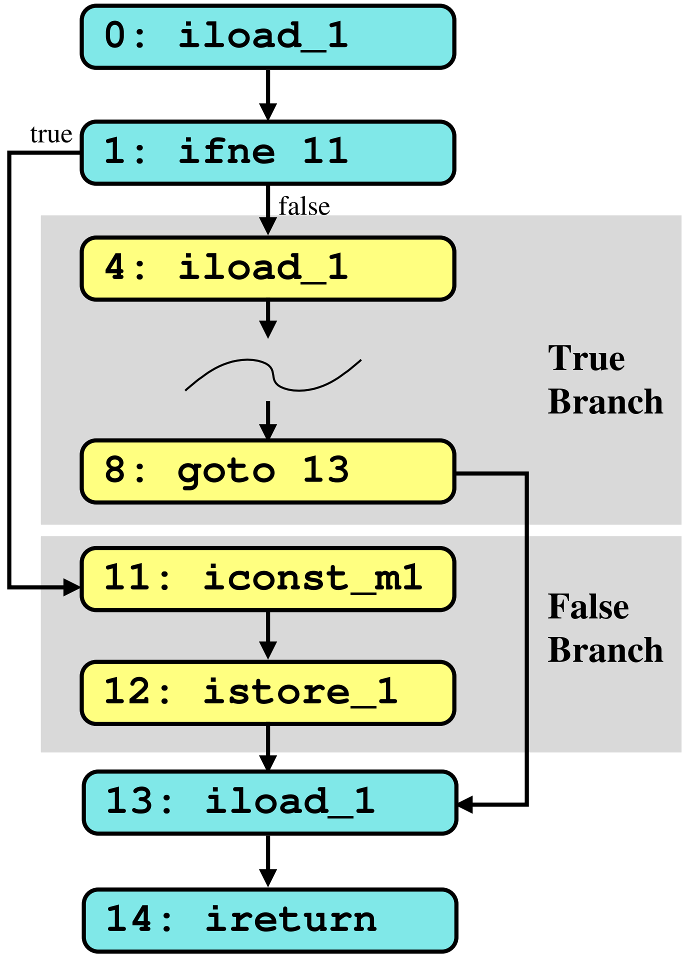


- In this case, no **else** branch — easy!
- But ... can't determine address for branch at line 1 until we've generated code for the true branch.

  Need to be able to insert instruction (or address) at an earlier location.

Translating If-Else-Conditionals

```
int f(int y) {
 if(y == 0) {
  y = y + 1;
 } else {
  y = -1;
 }
 return y;
}
```

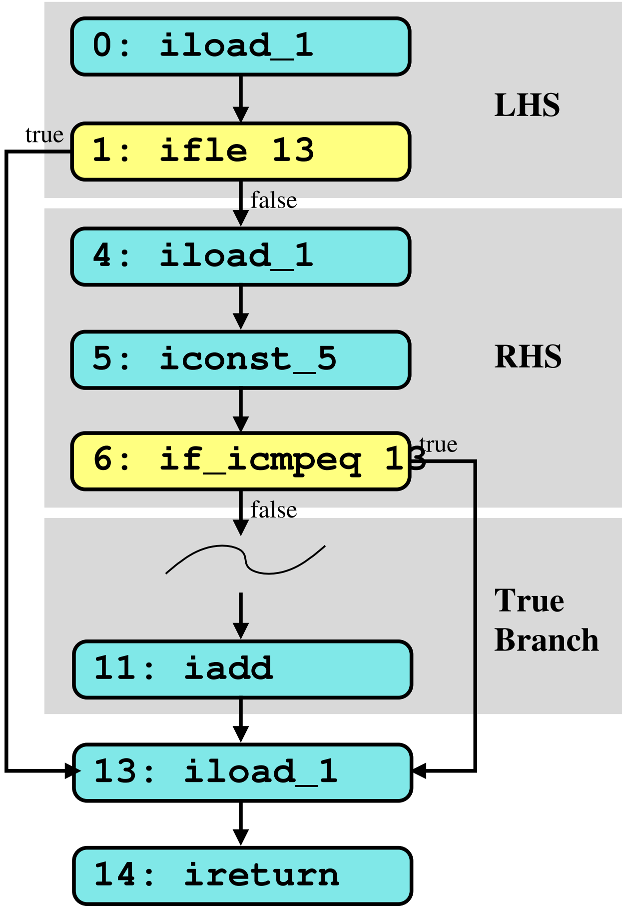


- The true branch **jumps over** the false branch!

Short Circuiting

- Logical connectives are translated using **short-circuiting**:

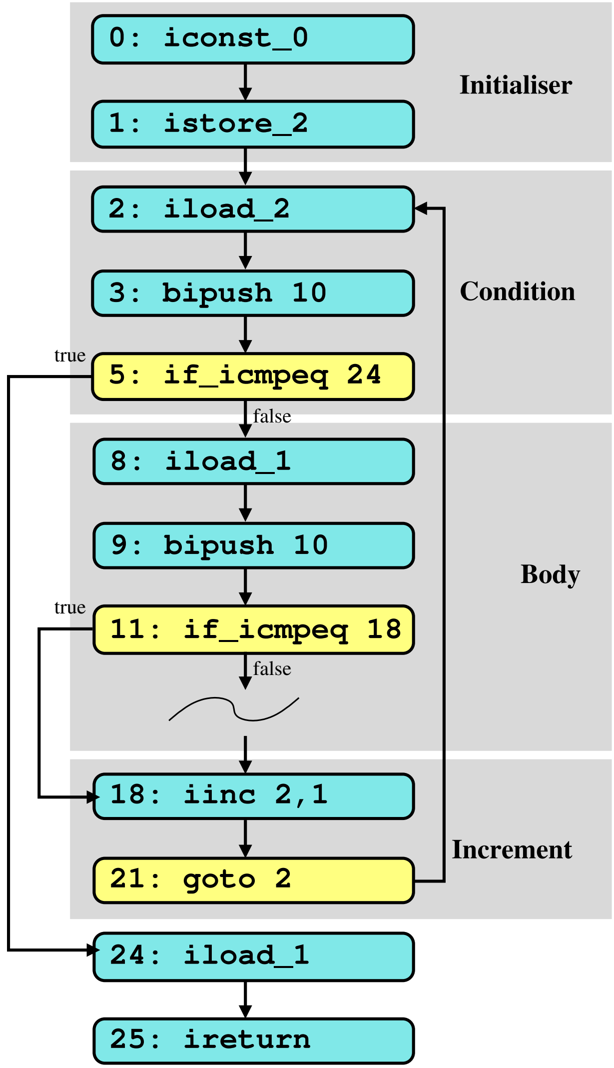
```
int f(int y) {
 if(y > 0 && y!=5) {
  y = y + 1;
 }
 return y;
}
```



- Here, right-hand expression **only executed if** left-hand gives true.

Translating Loops

```
int f(int y) {
 for(int i=0;i!=10;++i) {
  if(y==10) continue;
  y = y * 2;
 }
 return y;
}
```



Generating Branch Bytecodes

`goto`	[1 byte op][2 bytes offset] *Unconditional Branch (range -32768...+32767)*
`goto_w`	[1 byte op][4 bytes offset] *Unconditional Wide Branch (range $-2^{31}-1\ldots 2^{31}$)*
`ifeq`	[1 byte op][2 bytes offset] *Branch if top two stack locations equal (range -32768...+32767)*

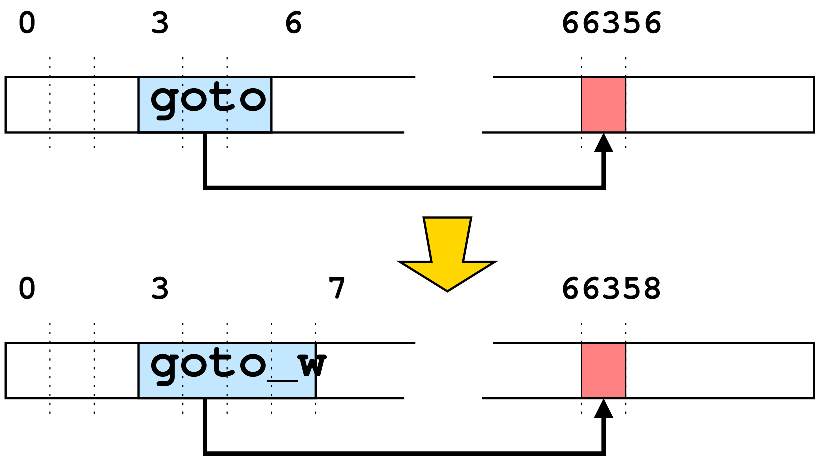
...

- Branch bytecodes use *relative addressing*, not *absolute addressing*
- Target address calculated by adding offset to current address:

```
void f(int):
 ...
 24: goto +35
 ...
 59: ...
```

Here, target address of **`goto`** bytecode is $(24+35)=59$

Calculating Branch Offsets



- Algorithm for calculating branch offsets:
  1. Generate all bytecodes, assuming branches take 3 bytes
  2. If branch exists which cannot reach target:

     Replace it with a *wide branch*:

     Update offsets of all branches (since they may have changed)
  3. Repeat step 2 until all branches can reach destination

- **Does this algorithm always terminate?**
  (need to consider padding of `tableswitch` + `lookupswitch`)

Generating Switch Bytecodes

- Two bytecodes for switch statements:

  `tableswitch` [op][padding][default][low][high][offsets]

  Padding: 0-3 zeroed bytes, so next byte word-aligned.
  Default: target address for default label
  Low: lowest value in case range
  High: Highest value in case range
  Offsets: Array of (high-low+1) Case Offsets

  `lookupswitch` [op][padding][default][npairs][pairs]

  padding: *0-3 zeroed bytes, so next byte word-aligned.*
  default: *target address for default label*
  npairs: *number of case value pairs*
  pairs: *array of pairs mapping case values to offsets*

Generating Switch Bytecodes (cont'd)

```
void f(int x) {
 int y;
 switch(x) {
  case 0:
   y = 1;
   break;
  case 1:
   y = 2;
  case 2:
   y = 3;
  default:
   y = -1;
 }
}
```

```
public void f(int);
   0:   iload_1
   1:   tableswitch
          default: 37
          low: 0
          high: 2
          offsets: +27, +32, +34
   28:  iconst_1
   29:  istore_2
   30:  goto    39
   33:  iconst_2
   34:  istore_2
   35:  iconst_3
   36:  istore_2
   37:  iconst_m1
   38:  istore_2
   39:  return
```

- Tableswitch is useful for contiguous case values
- **How many bytes of padding required here?**

Generating Switch Bytecodes (cont'd)

```
void f(int x) {
 int y;
 switch(x) {
  case 0:
   y = 1;
   break;
  case 12:
   y = 2;
  case 2046:
   y = 3;
  default:
   y = -1;
 }
}
```

```
public void f(int);
   0:   iload_1
   1:   lookupswitch
          default: 45
          npairs: 3
          pairs: 0→+35, 12→+40, 2046→+42
   36:  iconst_1
   37:  istore_2
   38:  goto    47
   41:  iconst_2
   42:  istore_2
   43:  iconst_3
   44:  istore_2
   45:  iconst_m1
   46:  istore_2
   47:  return
```

- Lookupswitch is useful for non-contiguous case values
- Notice that lookupswitch bytecode is much larger than before.

Generating Invoke Bytecodes

`invokevirtual`	[1 byte op][2 bytes index] *Invoke method on a receiver of class type. The method and receiver types are located in the constant pool at the given index.*
`invokeinterface`	[1 byte op][2 bytes index] *Invoke method on a receiver of interface type. The method and receiver types are located in the constant pool at the given index.*
`invokestatic`	[1 byte op][2 bytes index] *Invoke static method. The method and receiver types are located in the constant pool at the given index.*
`invokespecial`	[1 byte op][2 bytes index] *Invoke special method (e.g. constructor). The method and receiver types are located in the constant pool at the given index.*

Generating Invoke Bytecodes (Cont'd)

```
class Test {
  Test(int x) { }
  int f(String s, int i) {
    return 1;
  }

  static void m(String[] s){
    Test t = new Test(123);
    t.f(s[0],2);
  }
}
```

```
static void m(String[] s):
Code:
 0:    new Test
 3:    dup
 4:    bipush  123
 6:    invokespecial Test.<init>:(I)V
 9:    astore_1
 10:   aload_1
 11:   aload_0
 12:   iconst_0
 13:   aaload
 14:   iconst_2
 15:   invokevirtual Test.f:(L...;I)I
 18:   pop
 19:   return
```

- Receiver pushed on stack first (line 10)
- Parameters pushed on stack next in order (lines 13-14)
- Return value is popped afterwards since its not used (line 18)

Generating Bytecode for While Language

- Assignment 3 will be to generate Java bytecode for the While language.
- Use David's `Jasm` Assembler / Disassembler for Java Bytecode (`http://whiley.github.io/Jasm/`).

  Provides operations for generating JVM instructions, writing class files, etc.

  Saves you dealing with a lot of details — like using an assembler language.
- You'll also be given a skeleton translator, which takes care of a lot of the details of the class file and shows you how to do some of the translation.