Lecture Overview

Code generation in milestone 2
  o Code generation for array indexing
  o Some rational implementation
    ▪ Over
    ▪ Express Over
  o Creating records for arrays
  o Short-circuiting Or
  o If statement

Theory: attributed grammars
Code generation for array indexing

I’ll assume that you are using OperatorNode with a LextantToken containing Punctuator.ARRAY_INDEXING. We want to create a SimpleCodeGenerator for this operation.

Like other binary operators, we will have the first argument (array) and second argument (index) put on the stack, in that order. We want each of these operands to be values. But since array elements are targetable, we want to create an address as the overall result on the stack. So here’s what our code should look like:

| address code for index operation | value code for operand1 (array) | value code for operand2 (index) | code to implement indexing |
The code to implement indexing will need to use the index and the array more than once. So I recommend storing them in static memory somewhere. Create two four-byte locations in RunTime.java; I call mine ARRAY_INDEXING_ARRAY and ARRAY_INDEXING_INDEX.

```java
public static final String ARRAY_INDEXING_ARRAY = "$a-indexing-array";
public static final String ARRAY_INDEXING_INDEX = "$a-indexing-index";
```

Create a function in which to declare temporary variables, and declare the two temporaries in it.

```java
private ASMCodeFragment temporaryVariables() {
  ASMCodeFragment frag =
      new ASMCodeFragment(GENERATES_VOID);
  Macros.declareI(frag, ARRAY_INDEXING_ARRAY);
  Macros.declareI(frag, ARRAY_INDEXING_INDEX);
  return frag;
}
```

Macros.declareI is a shorthand for declaring space for an integer (or any 4-byte quantity). It writes a Dlabel and a DataZ to the fragment.

Now we must modify environmentASM(), which writes the code that is responsible for setting up the runtime
environment. We must add a call to our function that declares the temporaries.

```java
private ASMCodeFragment environmentASM() {
    ASMCodeFragment result =
        new ASMCodeFragment(GENERATES_VOID);
    result.append(jumpToMain());
    result.append(stringsForPrintf());
    result.append(runtimeErrors());
    result.append(temporaryVariables());
    result.add(DLabel, USABLE_MEMORY_START);
    return result;
}
```

So now we have the two temporary variables available to us at runtime. We can now think of the code as:

<table>
<thead>
<tr>
<th>address code for index operation</th>
<th>value code for operand1 (array)</th>
<th>value code for operand2 (index)</th>
</tr>
</thead>
<tbody>
<tr>
<td>store operand2 to ARRAY_INDEXING_INDEX</td>
<td>store operand1 to ARRAY_INDEXING_ARRAY</td>
<td>code to implement indexing</td>
</tr>
</tbody>
</table>
First we must check that the array is not null, then we must check that the index is in bounds for the array. We expand only the lower box of the above diagram.

<table>
<thead>
<tr>
<th>code to implement indexing</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>check if array is null</td>
<td></td>
</tr>
<tr>
<td>check index &gt;= lower bound of the array (0)</td>
<td></td>
</tr>
<tr>
<td>check index &lt; size of array</td>
<td></td>
</tr>
<tr>
<td>compute address of indexed element</td>
<td></td>
</tr>
</tbody>
</table>

To check if the array is null, we load the array argument and then jump if zero to a runtime error. To check if the index is greater than 0, we load the index and then jump if negative to a different runtime error.
To check if the index is less than the array size, we load the index and then load the array length (which we do not know at compile time). We then subtract and jump if zero-or-positive to the array-index out of bounds runtime error.

To compute the address of the indexed element, we load the array temporary (containing a pointer to the array), we add the header size (16 for an array), and add the index times the subelementSize (which we do know at compile time).
Bringing back the four boxes we omitted into the diagram, here's what the code looks like:

<table>
<thead>
<tr>
<th>Code to implement indexing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load array temp</td>
</tr>
<tr>
<td>Add</td>
</tr>
<tr>
<td>Pushi subelement size</td>
</tr>
<tr>
<td>Multiply</td>
</tr>
<tr>
<td>Load index temp</td>
</tr>
<tr>
<td>Add</td>
</tr>
<tr>
<td>Pushi 16</td>
</tr>
<tr>
<td>Load array temp</td>
</tr>
<tr>
<td>Jump if &gt;= 0 index-out-rte</td>
</tr>
<tr>
<td>Subtract</td>
</tr>
<tr>
<td>Load array temp</td>
</tr>
<tr>
<td>Load index temp</td>
</tr>
<tr>
<td>JumpNeg index-out-rte</td>
</tr>
<tr>
<td>Load index temp</td>
</tr>
<tr>
<td>Load index-out-rte</td>
</tr>
<tr>
<td>JumpZero null-array-rte</td>
</tr>
</tbody>
</table>

Here is the table:

<table>
<thead>
<tr>
<th>Code to implement indexing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load array temp</td>
</tr>
<tr>
<td>Add</td>
</tr>
<tr>
<td>Pushi subelement size</td>
</tr>
<tr>
<td>Multiply</td>
</tr>
<tr>
<td>Load index temp</td>
</tr>
<tr>
<td>Add</td>
</tr>
<tr>
<td>Pushi 16</td>
</tr>
<tr>
<td>Load array temp</td>
</tr>
<tr>
<td>Jump if &gt;= 0 index-out-rte</td>
</tr>
<tr>
<td>Subtract</td>
</tr>
<tr>
<td>Load array temp</td>
</tr>
<tr>
<td>Load index temp</td>
</tr>
<tr>
<td>JumpNeg index-out-rte</td>
</tr>
<tr>
<td>Load index temp</td>
</tr>
<tr>
<td>Load index-out-rte</td>
</tr>
<tr>
<td>JumpZero null-array-rte</td>
</tr>
<tr>
<td>value code for operand1 (array)</td>
</tr>
</tbody>
</table>

Address code for index operation
Now we translate that into java that writes real ASM. I’ll call the SimpleCodeGenerator that I make ArrayIndexingCodeGenerator. I keep mine in a subpackage of asmCodeGenerator called operators; you can keep yours wherever you like.

```java
package asmCodeGenerator.operators;

import static asmCodeGenerator.codeStorage.ASMOpcode.*;
import asmCodeGenerator.codeStorage.ASMCodeFragment;

public class ArrayIndexingCodeGenerator extends SimpleCodeGenerator {
    public ArrayIndexingCodeGenerator() {
    }

    @Override
    public ASMCodeFragment generate(ParseNode node) {
        ASMCodeFragment frag = new ASMCodeFragment(
            CodeType.GENERATES_ADDRESS);
        ...
        return frag;
    }
}
```
We now must fill in the . . .
SimpleCodeGenerator assumes that the arguments have already been put on the stack, so we don’t need to worry about our first two boxes. For the third box, “store operand2 to ARRAY_INDEXING_INDEX” We need to
  PushD ARRAY_INDEXING_INDEX
  Exch
  StoreI
However, that is already conveniently implemented in a macro in Macros.java. It’s called StoreITo, which takes the location as an argument. So our first line of java in the . . . becomes

Macros.storeITo(frag, ARRAY_INDEXING_INDEX);

And the next one, storing the array, becomes

Macros.storeITo(frag, ARRAY_INDEXING_ARRAY);

The third box says:

  load array temp
  JumpZero null-array-rte
And there’s a macro for loading an integer from a location, called LoadIFrom. So this becomes code

```
Macros.loadIFrom(frag, ARRAY_INDEXING_ARRAY);
frag.add(JumpZero, NULL_ARRAY_RUNTIME_ERROR);
```

The fourth box says:

```
load index temp
JumpNeg index-out-rte
```

and this translates as:

```
Macros.loadIFrom(frag, ARRAY_INDEXING_INDEX);
frag.add(JumpNeg,
        INDEX_OUT_OF_BOUNDS_RUNTIME_ERROR);
```

On to the fifth box:

```
load index temp
load array length
subtract
jump if >= 0 index-out-rte
```
The first line translates simply enough:

```
Macros.loadIFrom(frag, ARRAY_INDEXING_INDEX);
```

But the second line takes a little more. The array temp holds the address of the array record, and the length of the array is 12 bytes from the start of the array record, according to Milestone 2. So we must:

```
Macros.loadIFrom(frag, ARRAY_INDEXING_ARRAY);
frag.add(PushI, ARRAY_RECORD_LENGTH_OFFSET);
frag.add(Add);
frag.add(LoadI);
```

And there’s a macro for that, too. It’s called `readIPtrOffset()`. You should be able to figure it out if you want to use it.

The third line becomes:

```
frag.add(Subtract);
```

But the fourth one cannot be simply translated. We don’t have an instruction that jumps on non-negative values. We do have one that jumps on negative values, though, so we can implement it as:
if i<0 jump to label
jump to index-out-rte
label: Nop

To get a unique label for this part of the code, ask a Labeller for one.

Labeller labeller = new Labeller(“array-indexing”);
String label = labeller.newLabel(“in-bounds”);
frag.add(JumpNeg, label);
frag.add(Jump,
    INDEX_OUT_OF_BOUNDS_RUNTIME_ERROR);
frag.add(Label, label);
frag.add(Nop);

Just the last box left now:
load array temp
PushI 16
Add
load index temp
PushI subelement size
Multiply
Add
This translates easily, except for the subelement size. To get that, we need to use the node parameter to generate(). The indexing node’s first child (child(0)) is the array. The type of that array is the type of the node (child(0).getType()). The subelement type should be a getter on the Array type. And the size is a method on all types. So we get:

```java
Macros.loadIFrom(frag, ARRAY_INDEXING_ARRAY);
frag.add(PushI, ARRAY_HEADER_SIZE);
frag.add(Add);
Macros.loadIFrom(frag, ARRAY_INDEXING_INDEX);

Array arrayType = (Array)(node.child(0).getType());
Type subtype = arrayType.getSubtype();
frag.add(PushI, subtype.getSize());
frag.add(Multiply);
frag.add(Add);
```

Now the indexing code generator is all done, except for a couple of constants (ARRAY_RECORD_LENGTH_OFFSET, ARRAY_HEADER_SIZE) and a couple of runtime errors. I’ll leave you to appropriately define the constants in the appropriate place, and to create the new runtime errors in
RunTime.java. Imitate the divide-by-zero runtime errors. Be sure to call the new error methods from runtimeErrors().

It’s probably useful to comment the indexing code generator, using the commenting conventions that I discussed in the lecture on the ASM. For example

```java
// [... arr index]
Macros.storeITo(frag, ARRAY_INDEXING_INDEX);  // [... arr]
Macros.storeITo(frag, ARRAY_INDEXING_ARRAY);   // [... ]
Macros.loadIFrom(frag, ARRAY_INDEXING_ARRAY);  // [... arr]
frag.add(JumpZero, NULL_ARRAY_RUNTIME_ERROR);  // [... ]
Macros.loadIFrom(frag, ARRAY_INDEXING_INDEX);  // [... index]
frag.add(JumpNeg,
        INDEX_OUT_OF_BOUNDS_RUNTIME_ERROR);      // [... ]
```
Some Rational Implementation

Over and Express Over
Lexical analysis and parsing for these two should be straightforward. For semantic analysis, we just need to set the FunctionSignatures correctly. Over (\(/\)) just has the signature \((i, i) \rightarrow r\), and Express Over (\(//\)) has the two signatures \((r, i) \rightarrow i\) and \((f, i) \rightarrow i\).

```java
new FunctionSignatures(Punctuator.\text{OVER},
    new FunctionSignature(
        new FormRationalCodeGenerator(),
        INTEGER, INTEGER, RATIONAL)
);
new FunctionSignatures(Punctuator.\text{EXPRESS\_OVER},
    new FunctionSignature(
        new RationalExpressOverCodeGenerator(),
        RATIONAL, INTEGER, INTEGER),
    new FunctionSignature(
        new FloatingExpressOverCodeGenerator(),
        FLOATING, INTEGER, INTEGER)
);
```

Over
Over takes two operands; the left one (first on the stack) is the numerator, and the right one (second on the stack, so on the top) is the denominator. Placing the argument code in the normal places above the operator code
accomplishes almost all you need. If we did not have the requirement to store rationals in lowest terms, we could use the opcode Nop as whichVariant rather than the FormRationalCodeGenerator().

So all we have to do is to call the lowest-terms subroutine.

```java
public class FormRationalCodeGenerator extends SimpleCodeGenerator {
    public FormRationalCodeGenerator() {
    }

    @Override
    public ASMCodeFragment generate(ParseNode node) {
        ASMCodeFragment frag = new ASMCodeFragment(CodeType.GENERATES_VALUE);
        frag.add(Call, RunTime.LOWEST_TERMS);
        return frag;
    }
}
```

I put mine in RunTime.java, as you can see. The code I write from RunTime first stashes the return address in a static location, and after that expects a numerator and denominator on the stack (denominator on top). It first checks for zero denominator and issues a runtime error if that’s true. Then it stores its arguments in more static
locations, and proceeds with the GCD (Greatest Common Divisor) algorithm. Once it has the GCD, it stores it in yet another static location. Then it loads the numerator, and divides that by the GCD. Then it loads the denominator, and divides that by the GCD. Now the lowest-terms rational is properly on the stack. The code then loads the return address from where it stashed it, and executes a Return.

Hopefully that is enough for you to construct the detail around the GCD for the lowest-term subroutine.

**Express Over**
Express Over has two signatures and two code generators. We’ll look at the rational-argument one first.

First we need to check if the denominator (already on the stack with the numerator) is zero. Here are the relevant parts of the code generator:

```java
fragment.add(Duplicate);
fragment.add(JumpFalse,
    RunTime.RATIONAL_DIVIDE_BY_ZERO_RUNTIME_ERROR);
```

Next we store the arguments:
storeITo(fragment, RunTime.EXPRESS_OVER_DENOMINATOR);

That’s the second argument, the target denominator, stored. To store the first argument, we must perform two stores, because it’s a rational and takes up two stack locations.

storeITo(fragment, RunTime.RATIONAL_DENOMINATOR_TEMP);
storeITo(fragment, RunTime.RATIONAL_NUMERATOR_TEMP);

Following the checkpoint specification, we need to multiply the first argument \( f \) (a rational) with the second argument \( d \), and then convert to integer. If \( f = \frac{num}{den} \), we simply need to multiply \( num \times d \), then divide by \( den \). Integer division will take care of the convert to integer operation.

loadIFrom(fragment, RunTime.RATIONAL_NUMERATOR_TEMP);
loadIFrom(fragment, RunTime.EXPRESS_OVER_DENOMINATOR);
fragment.add(Multiply);

loadIFrom(fragment, RunTime.RATIONAL_DENOMINATOR_TEMP);
fragment.add(Divide);
And that’s all that’s needed. There is probably a way to do it with less stores and loads, but I like to keep things simple.

By the way, the Rationalize operator does exactly the same thing, followed by

```java
loadIFrom(fragment, RunTime.EXPRESS_OVER_DENOMINATOR);
frag.add(Call, RunTime.LOWEST_TERMS);
```

The floating-point argument Express Over is even simpler than the rational one was. First, check for a zero target denominator. Then convert the second argument to floating. Floating-multiply the two arguments, and convert back to integer.
Creating records for arrays

We’ll look at creating records for empty arrays. The empty array creation operator takes a single argument, the number of elements of the array to create. We can think of the code as follows:

<table>
<thead>
<tr>
<th>Code to implement empty array</th>
</tr>
</thead>
<tbody>
<tr>
<td>give error if nElements &lt; 0</td>
</tr>
<tr>
<td>compute record size</td>
</tr>
<tr>
<td>create record of that size</td>
</tr>
<tr>
<td>with <strong>record</strong> header filled in</td>
</tr>
<tr>
<td>zero out elements of array</td>
</tr>
<tr>
<td>fill in <strong>array</strong> header</td>
</tr>
</tbody>
</table>

Creating a record of the given size is going to be code that is reusable for any type that has records, so I’m going to put it in a separate subroutine. I’ll also create a static location that I call RECORD_CREATION_TEMP where I will store the new record. At this point I assume you know how to do that (in RunTime.java).

The things that are variable about creating a Record are the size, the typecode, and the status flags. The size must be computed at run time, so it will come in on the stack. We assume that it is the size of the record including the
header. The typecode and status flags are known at compile time and can be passed in.

```
// leaves new record in RECORD_CREATION_TEMPORARY
// [... size] -> [...]
public static void createRecord(ASMCodeFragment code,
                                  int typecode, int statusFlags) {

    The first thing to do is to get the record from the memory manager.

        code.add(Call,
                  MemoryManager.MEM_MANAGER_ALLOCATE);

    Next we store the returned pointer to the record in our temporary.

        Macros.storeITo(code, RECORD_CREATION_TEMPORARY);

    Finally, we store the typecode and statusFlags to the appropriate place. Here I use a macro that loads a pointer from a named location, adds an offset, and then stores a number there.
writeIPBaseOffset(code, 
   RECORD_CREATION_TEMPORARY, 
   Record.RECORD_TYPEID_OFFSET, typecode);
writeIPBaseOffset(code, 
   RECORD_CREATION_TEMPORARY, 
   Record.RECORD_STATUS_OFFSET, statusFlags);

I have a class called Record that holds a bunch of constants to do with records, so I can refer to them by symbolic names rather than numbers.

That does it for createRecord. Now our code looks like:

give error if nElements < 0
calculate record size
call createRecord
zero out elements of array
fill in array header

Here’s the header of the method I use for this.

```java
public static void createEmptyArrayRecord( 
   ASMCodeFragment code, 
   int statusFlags, int subtypeSize) {
   final int typecode = Record.ARRAY_TYPE_ID;
```
The first box of the code is straightforward. Since nElements arrives as the one argument of empty array creation, we implement it as just:

```java
code.add(Duplicate); // [... nElems nElems]
code.add(JumpNeg, // [... nElems]
    RunTime.NEGATIVE_LENGTH_ARRAY_RUNTIME_ERROR);
```

The second box is not so bad, but it computes a value (the length of the array without the header) that I want to save for later:

```java
code.add(Duplicate); // [... nElems nElems]
code.add(PushI, subtypeSize);
    // [... nElems nElems subSize]
code.add(Multiply); // [... nElems elemsSize]
code.add(Duplicate);
    // [... nElems elemsSize elemsSize]
storeITo(code, ARRAY_DATASIZE_TEMPORARY);
    // [... nElems elemsSize]
code.add(PushI, Record.ARRAY_HEADER_SIZE);
    // [... nElems elemsSize AHS]
code.add(Add); // [... nElems totalRecordSize]
```

The third box, as we said, is a simple call now:

```java
createRecord(code, typecode, statusFlags);
    // [... nElems]
```
The fourth box is zeroing out the elements. I’ll use a subroutine for this, too. The subroutine will take two arguments on the stack: a base pointer and a number of bytes. \([\ldots \text{baseAddr} \ \text{numBytes}] \rightarrow [\ldots]\) I call it CLEAR_N_BYTES. To compute the base address, we need to add the array header size to array pointer.

```plaintext
loadIFrom(code, RECORD_CREATION_TEMPORARY);
   // [... nElems_ptr]
code.add(PushI, Record.ARRAY_HEADER_SIZE);
   // [... nElems_ptr AHS]
code.add(Add);  // [... nElems elemsPtr]
loadIFrom(code, ARRAY_DATASIZE_TEMPORARY);
   // [... nElems elemsPtr elemSize]
code.add(Call, CLEAR_N_BYTES);
```

The final box is to write the array header. This means the parts of the array header that were not written by createRecord. These are the subtypeSize and the array length. I use two macros to do the indirect writing of these two values. The first is known at compile time, so its macro gets an extra parameter. The second is known only at runtime, and is the “nElems” that we have been careful to maintain on the bottom of the stack.
That ends createEmptyArrayRecord. To recap, you call it as

```c
createEmptyArrayRecord(ASMCodeFragment code, int statusFlags, int subtypeSize)
```

and it writes onto the end of the ASMCodeFragment code the code that creates an empty array record with the given status flags and subtypeSize. You should call this from the SimpleCodeGenerator for the `new` operator.
Short-circuiting Or

Short-circuiting boolean operators present a problem that SimpleCodeGenerator doesn’t solve. The code is no longer:

<table>
<thead>
<tr>
<th>code for operand 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>code for operand 2</td>
</tr>
<tr>
<td>code for operator</td>
</tr>
</tbody>
</table>

But, for short-circuiting Or, it’s more like:

<table>
<thead>
<tr>
<th>code for operand 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>if operand1 is true, branch to end with result true.</td>
</tr>
<tr>
<td>code for operand 2</td>
</tr>
<tr>
<td>result is result of operand 2</td>
</tr>
</tbody>
</table>

So we need to put something between the code for the operands. A new mechanism – a new type of whichVariant – is called for. We can call it a FullCodeGenerator.
First let’s prepare for it over in asmCodeGenerator inside visitNormalBinaryOperatorNode. We last left this method looking like:

```java
private void visitNormalBinaryOperatorNode(BinaryOperatorNode node) {
    newValueCode(node);
    ASMCodeFragment arg1 = removeValueCode(node.child(0));
    ASMCodeFragment arg2 = removeValueCode(node.child(1));

    code.append(arg1);
    code.append(arg2);

    Object variant = node.getSignature().getVariant();
    if (variant instanceof ASMOpcode) {
        ...
    } else if (variant instanceof SimpleCodeGenerator) {
        ...
    }
}
```

We’ll have to move the `code.append` calls inside the clauses of the if statement:
visitNormalBinaryOperatorNode(BinaryOperatorNode node) {
    newValueCode(node);
    ASMCodeFragment arg1 = removeValueCode(node.child(0));
    ASMCodeFragment arg2 = removeValueCode(node.child(1));

    Object variant = node.getSignature().getVariant();
    if (variant instanceof ASMOpcode) {
        code.append(arg1);
        code.append(arg2);
        ...
    } else if (variant instanceof SimpleCodeGenerator) {
        code.append(arg1);
        code.append(arg2);
        ...
    }
}

And then we need to handle the case when the variant is an instance of FullCodeGenerator:

else if (variant instanceof FullCodeGenerator) { 
    ...
}
The code in that case should look exactly like the old code for the SimpleCodeGenerator, except SimpleCodeGenerator is replaced by FullCodeGenerator:

```java
else if(variant instanceof FullCodeGenerator) {
    FullCodeGenerator generator = (FullCodeGenerator)variant;
    ASMCodeFragment fragment = generator.generate(node, arg1, arg2);
    code.append(fragment);
    if(fragment.isAddress()) {
        code.markAsAddress();
    }
}
```

The interface FullCodeGenerator has the one method

```java
generate(ParseNode node, ASMCodeFragment... operandCode);
```

Then the short-circuiting or code generator looks like:
public class ShortCircuitOrCodeGenerator extends FullCodeGenerator {

@Override
protected void generate (ParseNode node, ASMCodeFragment... args) {

    Labeller labeller = new Labeller("SC-Or");
    final String trueLabel =
        labeller.newLabel("short-circuit-true");
    final String endLabel =
        labeller.newLabel("end");

    // compute arg 1
    code.append(args[0]);  // [... bool]

    // short circuiting test
    code.add(Duplicate);    // [... bool bool]
    code.add(JumpTrue, trueLabel); // [... bool]
    code.add(Pop);          // [... 0] -> [...]

    // compute arg 2
    code.append(args[1]);  // [... bool]
    code.add(Jump, endLabel);

    // the end
    code.add(Label, trueLabel); // [... 1]
    code.add(Label, endLabel);
}
}
Control Statements

Pika-2 introduces the control statements **if** and **while**.

An IfStatementNode will have two or three children. We consider the three-child case (when the **if** has an **else** clause). The code for such an ifStatement should look like:

<table>
<thead>
<tr>
<th>Code for the condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>if condition is <em>false</em>, branch to falseClause</td>
</tr>
<tr>
<td>code for the then-clause</td>
</tr>
<tr>
<td>jump to end</td>
</tr>
<tr>
<td>label falseClause</td>
</tr>
<tr>
<td>code for the else-clause</td>
</tr>
<tr>
<td>label end</td>
</tr>
</tbody>
</table>

The two-child case is the same except the else-clause is empty.

The code generation for this belongs in the CodeVisitor inside ASMCodeGenerator.java. More specifically, it belongs in visitLeave(IfStatementNode) in that class.
First off in that method, we need to generate the labels that we need for the branch and the jump:

```java
public void visitLeave(IfStatementNode node) {
    Labeller labeller = new Labeller("if");
    String falseLabel = labeller.newLabel("false");
    String endLabel = labeller.newLabel("end");
}
```

Since `if` is a statement, it should leave nothing on the stack. That means that it is `void code`.

```java
newVoidCode(node);
```

The condition of the `ifStatement` is its first child, and that should generate a value. The then clause is the second child, and that shouldn’t generate a value.

```java
ASMCodeFragment condition = removeValueCode(node.child(0));
ASMCodeFragment thenClause = removeVoidCode(node.child(1));
```

Now we can construct the code up to the else clause:
```java
code.append(condition);
code.add(JumpFalse, falseLabel);
code.append(thenClause);
code.add(Jump, endLabel);

code.add(Label, falseLabel);

Next we check if the node has an else clause, and add that clause to the code if it does. If it doesn’t have an else clause, add nothing to the code.

if(hasElseClause(node)) {
    ASMCodeFragment elseClause = removeVoidCode(node.child(2));
    code.append(elseClause);
}

We end with the label for the end:

code.add(Label, endLabel);
}

This was a very straightforward implementation of the code outline above. In case you’re wondering about hasElseClause(node),

private boolean hasElseClause(IfStatementNode node) {
    return node.nChildren()==3;
}
```
The translation of the **while** statement happens in a similar vein. I’ll leave the particulars to you.
Attribution Grammars

An attributed grammar (or attribute grammar) is a grammar along with a set of rules for each production. Each rule defines an attribute of a grammar symbols in the production, often in terms of other attributes of grammar symbols in the production. For example,

<table>
<thead>
<tr>
<th>production</th>
<th>rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_1 \rightarrow (E_2)$</td>
<td>$E_1$.type := $E_2$.type</td>
</tr>
<tr>
<td></td>
<td>$E_1$.isConstant := $E_2$.isConstant</td>
</tr>
</tbody>
</table>

Where subscripts have been used, as is conventional, to distinguish between two instances of the nonterminal $E$ in the production.

If a rule gives a grammar symbol an attribute, then each instance of that grammar symbol in the parse tree gets its own instance of that attribute. In the above grammar, all $E$’s would get an attribute $type$ and an attribute $isConstant$. 
The rules are declarative. They imply no specific evaluation order—they require only that the attributes in the right-hand side of the rule have already been evaluated.

Let’s examine a small attributed grammar.

<table>
<thead>
<tr>
<th>production</th>
<th>rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_1 \rightarrow L + E_2$</td>
<td>$E_1.type := \text{if (L.type == int &amp;&amp; E_2.type == int)}$ \nthen $\text{int}$ else $\text{float}$ \n$E_1.isConstant := L.isConstant &amp;&amp; E_2.isConstant$</td>
</tr>
<tr>
<td>$E \rightarrow L$</td>
<td>$E.type := L.type$ \n$E.isConstant := L.isConstant$</td>
</tr>
<tr>
<td>$L \rightarrow (E)$</td>
<td>$L.type := E.type$ \n$L.isConstant := E.isConstant$</td>
</tr>
<tr>
<td>$L \rightarrow v$</td>
<td>$L.type := v.type$ \n$v.type := v.getBinding().getType()$ \n$L.isConstant := \text{false}$</td>
</tr>
<tr>
<td>$L \rightarrow ic$</td>
<td>$L.type := ic.type$ \n$ic.type := \text{int}$ \n$L.isConstant := \text{true}$</td>
</tr>
<tr>
<td>$L \rightarrow fc$</td>
<td>$L.type := fc.type$ \n$fc.type := \text{float}$ \n$L.isConstant := \text{true}$</td>
</tr>
</tbody>
</table>
Consider parsing \( \text{ic} + (\mathbf{v} + \text{fc}) \) where the variable \( \mathbf{v} \) is of type \text{int}. Here’s the parse tree:
Here’s the parse tree with all of the attributes at each node.
Now we’ll draw a digraph on the attributes where an edge from a to b means that the value of a is required to compute the value of b.
Finally, here is the diagram with all of the values of the attributes computed.
If the digraph is not a directed acyclic graph (DAG), then the attribution scheme has a problem.

In this attributed grammar, all of the flow of values goes up the tree. (This type of grammar is called S-attributed.) Not all attributed grammars are S-attributed. An example of a non-S-attributed grammar is given in Section 4.3 of the text; please read this section and Section 4.4.

In general, if an attribute is computed from attributes of symbols below it in the tree, it is called a synthesized or synthetic attribute. If it is computed using attributes of its siblings and its parents, it is called an inherited attribute.
An attributed grammar is called **L-attributed** if every attribute at a node is computable if one knows the value of all the attributes of the node’s children and all of the attributes of the node’s left siblings. The class of L-attributed grammars contains the class of S-attributed grammars.

With an L-attributed grammar, one can evaluate the attributes in a straightforward left-to-right depth first search.

If a grammar is not L-attributed, then the compiler must determine the evaluation ordering in another way; at worst it needs to do a **topological sort** on the attribute digraph.
The book talks about **ad-hoc syntax-directed translation**. Syntax-directed translation is translation based on the grammar. Attributed grammars are thus syntax-directed translation but they are systematic rather than ad-hoc.

The Visitor mechanism in our compiler is based on a visit routine for each type of node in the AST, which is essentially each grammar symbol. It is therefore syntax-directed translation. It is ad-hoc, though it is very similar to (and can be based on) attributed grammars. It is not restricted to L-attributed grammars, though: it can pass information down the tree if necessary. It also uses some nonlocal information, in the form of symbol tables located at nodes elsewhere in the tree.