{Bite}Code Generation

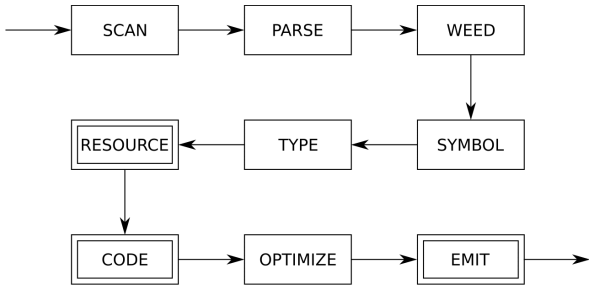
COMP 520: Compiler Design (4 credits)

Alexander Krolik

alexander.krolik@mail.mcgill.ca

MWF 8:30-9:30, TR 1080

http://www.cs.mcgill.ca/~cs520/2019/

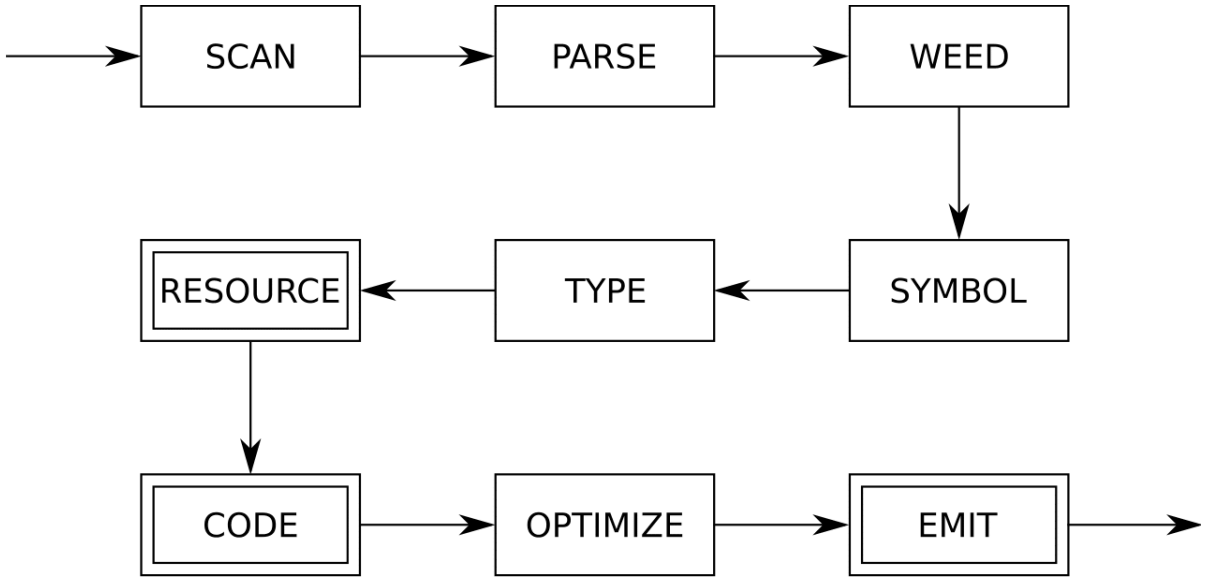


Compiley “Mompiler” McCompilerface

Code Generation

The *code generation* phase has several sub-phases

- Computing *resources* such as stack layouts, offsets, labels, registers, and dimensions;
- Generating an internal representation of machine codes for statements and expressions;
- Optimizing the code (ignored for now); and
- Emitting the code to files in assembler or binary format.



Resources - JOOS

In a general sense, resources are metadata necessary to generate code for the target architecture. They represent information that the machine will use for setup and/or execution of a code segment.

- Local and stack limits for methods;
- Offsets for locals and formals; and
- Labels for control structures.

These values cannot be computed based on a single statement – we must perform a global traversal of the parse trees and collect state information.

Resources - Offsets and Locals Limit

```
public class Resources {
    public Resources() { super(); }
    public void Method(int p, int q, Resources r) {
        int x;
        int y;
        {
            int z;
            z = 87;
        }
        {
            boolean a;
            {
                int z;
                a = true;
            }
            {
                int k;
                k = q;
            }
        }
    }
}
```

What are the local variable offsets and locals limit?

Resources - Offsets and Locals Limit

```
public class Resources {
    public Resources() { super(); }
    public void Method(int p [1], int q [2], Resources r [3]) {
        int x [4];
        int y [5];
        {
            int z [6];
            z = 87;
        }
        {
            boolean a [6];
            {
                int z [7];
                a = true;
            }
            {
                int k [7];
                k = q;
            }
        }
    }
}
```

The locals limit is the largest offset generated on any path + one extra slot for `this` (non-static methods).

Resources - Offsets and Locals Limit

The AST is recursively traversed, associating each variable declaration with the next offset

```
int offset, localslimit;

// Generate the next offset, keeping track of the limit
int nextoffset() {
   offset++;
   if (offset > localslimit) localslimit = offset;
   return offset;
}

void resFORMAL(FORMAL *f) {
   if (f != NULL) {
      resFORMAL(f->next);
      f->offset = nextoffset();
   }
}

void resLOCAL(LOCAL *l) {
   if (l != NULL) {
      resLOCAL(l->next);
      l->offset = nextoffset();
   }
}
```

Resources - Offsets and Locals Limit

Handling blocks

Blocks open a new scope, where each declaration is local to the new scope. We therefore restore the previous locals offset after computing its resources.

```
case blockK:
     baseoffset = offset;
     resSTATEMENT(s->val.blockS.body);
     offset = baseoffset;
     break;
```

Resources - Labels

When executing a branch instruction, the JVM jumps to a new section of code tagged with a label. Each control structure therefore contains labels to implement its respective control flow.

`if`:	1 label
`ifelse`:	2 labels
`while`:	2 labels
`||` and `&&`:	1 label
`==`, `<`, `>`, `<=`, `>=`, and `!=`:	2 labels
`!`:	2 labels
`toString` coercion:	2 labels

Labels are generated consecutively, for each method and constructor separately.

The Jasmin assembler converts labels to addresses. An address in Java bytecode is a 16-bit offset with respect to the branching instruction. The target address must be part of the code array of the same method.

Resources - Labels

A recursive traversal generates the necessary labels for each control structure

```
int label;
int nextlabel() {
   return label++;
}

[...]

case whileK:
   s->val.whileS.startlabel = nextlabel();
   s->val.whileS.stoplabel = nextlabel();
   resEXP(s->val.whileS.condition);
   resSTATEMENT(s->val.whileS.body);
   break;

[...]

case orK:
   e->val.orE.truelabel = nextlabel();
   resEXP(e->val.orE.left);
   resEXP(e->val.orE.right);
   break;
```

Internal Representation of Bytecode

Before optimizing and emitting, the compiler must generate an internal representation of the machine code.

Jasmin bytecode

```
typedef struct CODE {
   enum {
      newCK,
      instanceofCK,
      imulCK,
      ifeqK,
      if_acmpeqCK,
      invokevirtualCK,
      [...]
   } kind;
   union {
      char *newC;
      char *instanceofC;
      int ifeqC;
      char *invokevirtualC;
      [...]
   } val;
   struct CODE *next;
} CODE;
```

Announcements (Monday, February 18th)

Assignments

- Assignment 2 has been graded

Milestone 1

- Get started early!
- Any questions?
- **Due:** Friday, March 1st 11:59 PM

Midterm

- **Date:** Tuesday, February 26th 6:00 - 7:30 PM
- **Review:** Friday 22nd in class
- Sample midterm (2018) has been emailed!

Code Templates

To generate internal machine code representation based on the AST, compilers use code templates that

- Show how to generate correct code for each language construct;
- Ignore the surrounding context; and
- Dictate a *simple*, recursive strategy.

This strategy generates *correct* but *inefficient* code that can be later optimized. Why?

Why code templates?

- Generating efficient code directly from the AST requires extensive context knowledge;
- Is extremely tricky to get right in general; and
- Ignores the possibility of more complex transformations (e.g. common subexpression elimination).

Note: While templates might be globally unoptimal, they should try be locally optimal.

Code Template Invariants

In Java bytecode, the generated code must have the following invariants for the stack

- Evaluation of a statement leaves the stack height unchanged; and
- Evaluation of an expression increases the stack height by one.

This follows the logic that expressions (and not statements) have an associated value.

Special case of `ExpressionStatement`

- If the expression evaluates to a value, the result is popped off the stack; but
- For `void` return expressions, nothing is popped.

Template - If Statement

`if (`E`)` S

Template

E

`ifeq stop`

S

`stop:`

JOOS source

```
case ifK:
   codeEXP(s->val.ifS.condition);
   code_ifeq(s->val.ifS.stoplabel);
   codeSTATEMENT(s->val.ifS.body);
   code_label("stop", s->val.ifS.stoplabel);
   break;
```

Template - IfElse Statement

`if (`E`)` S_1 `else` S_2

Template

E

`ifeq else`

S_1

`goto stop`

`else:`

S_2

`stop:`

JOOS source

```
case ifelseK:
   codeEXP(s->val.ifelseS.condition);
   code_ifeq(s->val.ifelseS.elselabel);
   codeSTATEMENT(s->val.ifelseS.thenpart);
   code_goto(s->val.ifelseS.stoplabel);
   code_label("else", s->val.ifelseS.elselabel);
   codeSTATEMENT(s->val.ifelseS.elsepart);
   code_label("stop", s->val.ifelseS.stoplabel);
   break;
```

Let's Practice!

Compute the resources and generate the Jasmin code for the following method

```
public int m(int x) {
   if (x < 0)
      return (x * x);
   else
      return (x * x * x);
}
```

Jasmin Code

```
public int m(int x) {
   if (x < 0)
      return (x * x);
   else
      return (x * x * x);
}
```

```
.method public m(I)I
.limit locals 2
.limit stack 2
  iload_1
  iconst_0
  if_icmplt true_2
  iconst_0
  goto stop_3
true_2:
  iconst_1
stop_3:
  ifeq else_0
  iload_1
  iload_1
  imul
  ireturn
  goto stop_1
else_0:
  iload_1
  iload_1
  imul
  iload_1
  imul
  ireturn
stop_1:
  nop
.end method
```

Let's Practice!

Compute the resources and generate the Jasmin code for the following method. What would be different for this compared to the previous method?

```
public int m(int x) {
   if (x < 0)
      return (x * x);
   else
      return (x * (x * x));
}
```

Jasmin Code

```
public int m(int x) {
   if (x < 0)
      return (x * x);
   else
      return (x * (x * x));
}
```

```
.method public m(I)I
.limit locals 2
.limit stack 3
   iload_1
   iconst_0
   if_icmplt true_2
   iconst_0
   goto stop_3
true_2:
   iconst_1
stop_3:
   ifeq else_0
   iload_1
   iload_1
   imul
   ireturn
   goto stop_1
else_0:
   iload_1 ; load x three times
   iload_1
   iload_1
   imul  ; two imuls
   imul
   ireturn
stop_1:
   nop
.end method
```

Template - While Statement

`while (`E`)` S

Template

`start:`

E

`ifeq stop`

S

`goto start`

`stop:`

JOOS source

```
case whileK:
   code_label("start", s->val.whileS.startlabel);
   codeEXP(s->val.whileS.condition);
   code_ifeq(s->val.whileS.stoplabel);
   codeSTATEMENT(s->val.whileS.body);
   code_goto(s->val.whileS.startlabel);
   code_label("stop", s->val.whileS.stoplabel);
   break;
```

Template - Expression Statement

E;

Template (E type void)

E

Template (otherwise)

E

pop

JOOS source

```
case expK:
   codeEXP(s->val.expS);
   if (s->val.expS->type->kind != voidK) {
         code_pop();
   }
   break;
```

Template - Local Variable Expression

x

Template (x type `int` or `boolean`)

`iload` *offset(*x*)*

Template (otherwise)

`aload` *offset(*x*)*

JOOS source

```
case localSym:
   if (e->val.idE.idsym->val.localS.type->kind == refK) {
      code_aload(e->val.idE.idsym->val.localS->offset);
   } else {
      code_iload(e->val.idE.idsym->val.localS->offset);
   }
   break;
```

Template - Assignment

$x = E$

An assignment in JOOS is an expression on its own. It must therefore leave its value on the stack

Template (x has type `int` or `boolean`)

E
dup
istore *offset(x)*

Template (otherwise)

E
dup
astore *offset(x)*

JOOS source

```
case formalSym:
   codeEXP(e->val.assignE.right);
   code_dup();
   if (e->val.assignE.leftsym->val.formalS->type->kind == refK) {
      code_astore(e->val.assignE.leftsym->val.formalS->offset);
   } else {
      code_istore(e->val.assignE.leftsym->val.formalS->offset);
   }
   break;
```

Announcements (Wednesday, February 20th)

Milestone 1

- Get started early!
- Any questions?
- **Due:** Friday, March 1st 11:59 PM

Midterm

- **Date:** Tuesday, February 26th 6:00 - 7:30 PM
- **Review:** Friday 22nd in class
- Sample midterm (2018) has been emailed!

Template - Equality Expression

E_1 == E_2

Template (E_i has type `int` or `boolean`) *use* `if_acmpeq` *otherwise*

E_1
E_2
```
if_icmpeq true
ldc_int 0
goto stop
true:
ldc_int 1
stop:
```

JOOS source

```
case eqK:
   codeEXP(e->val.eqE.left);
   codeEXP(e->val.eqE.right);
   if (e->val.eqE.left->type->kind==refK) {
      code_if_acmpeq(e->val.eqE.truelabel);
   } else {
      code_if_icmpeq(e->val.eqE.truelabel);
   }
   code_ldc_int(0);
   code_goto(e->val.eqE.stoplabel);
   code_label("true",e->val.eqE.truelabel);
   code_ldc_int(1);
   code_label("stop",e->val.eqE.stoplabel);
   break;
```

Short-Circuiting Logical Operators

Consider the following method which has both `or`/`and` logical operators

```
public int m(int x, int y) {
   if (y != 0 || x != 0)
      return x * y;
   if (y != 0 && x / y > 2)
      return x / y;
   return 0;
}
```

Which parts of the conditions are executed, and which may be skipped?

Template - Or Expression

E_1 || E_2

Template

E_1

```
dup
ifne true
pop
```

E_2

```
true:
```

JOOS source

```
case orK:
   codeEXP(e->val.orE.left);
   code_dup();
   code_ifne(e->val.orE.truelabel);
   code_pop();
   codeEXP(e->val.orE.right);
   code_label("true",e->val.orE.truelabel);
   break;
```

Short-Circuiting `Or`

```
if (y != 0 || x != 0)
   return x * y;
```

```
   iload_2
   iconst_0
   if_icmpne true_8
   iconst_0
   goto stop_9
true_8:
   iconst_1
stop_9:
   dup
   ifne true_7
   pop
   iload_1
   iconst_0
   if_icmpne true_10
   iconst_0
   goto stop_11
true_10:
   iconst_1
stop_11:
true_7:
   ifeq stop_6
   iload_1
   iload_2
   imul
   ireturn
stop_6:
```

Short-Circuiting `And`

```
if (y != 0 && x / y > 2)
   return x / y;
```

```
   iload_2
   iconst_0
   if_icmpne true_2
   iconst_0
   goto stop_3
true_2:
   iconst_1
stop_3:
   dup
   ifeq false_1
   pop
   iload_1
   iload_2
   idiv
   iconst_2
   if_icmpgt true_4
   iconst_0
   goto stop_5
true_4:
   iconst_1
stop_5:
false_1:
   ifeq stop_0
   iload_1
   iload_2
   idiv
   ireturn
stop_0:
```

Template - Not Expression

$!E$

Template

E

```
ifeq true
ldc_int 0
goto stop
true:
ldc_int 1
stop:
```

JOOS source

```
case notK:
   codeEXP(e->val.notE.not);
   code_ifeq(e->val.notE.truelabel);
   code_ldc_int(0);
   code_goto(e->val.notE.stoplabel);
   code_label("true",e->val.notE.truelabel);
   code_ldc_int(1);
   code_label("stop",e->val.notE.stoplabel);
   break;
```

Not Expression Branching

```
public int m(int x) {
   if (!!!!(x < 0))
      return (x * x);
   else
      return (x * x * x);
}
```

What is the minimal number of labels we need?

```
.method public m(I)I
.limit locals 2
.limit stack 2
   iload_1
   iconst_0
   if_icmplt true_10
   iconst_0
   goto stop_11
true_10:
   iconst_1
stop_11:
   ifeq true_8
   iconst_0
   goto stop_9
true_8:
   iconst_1
stop_9:
   ifeq true_6
   iconst_0
   goto stop_7
true_6:
  iconst_1
stop_7:
  ifeq true_4
  iconst_0
  goto stop_5
true_4:
  iconst_1
stop_5:
   ifeq true_2
   iconst_0
   goto stop_3
true_2:
   iconst_1
stop_3:
   ifeq else_0
   iload_1
   iload_1
   imul
   ireturn
   goto stop_1
else_0:
   iload_1
   iload_1
   imul
   iload_1
   imul
   ireturn
stop_1:
   nop
.end method
```

Alternative Translation of Boolean Expressions

Given an expression of form

$$!\,!\,!\,!\,!\,!\,!\,\boldsymbol{E}$$

the templates we described earlier would generate lots of jumps.

Idea: Encode boolean logic by more clever introduction and swaps of labels

Define a function *trans*$(\boldsymbol{b}, \boldsymbol{l}, \boldsymbol{t}, \boldsymbol{f})$ to represent the translation of expression $\boldsymbol{b}$ to code

- $\boldsymbol{b}$: Boolean expression
- $\boldsymbol{l}$: label for evaluating current expression
- $\boldsymbol{t}$: jump-label in case $\boldsymbol{b}$ evaluates to $\mathbf{true}$
- $\boldsymbol{f}$: jump-label in case $\boldsymbol{b}$ evaluates to $\mathbf{false}$

Alternative Translation of Boolean Expressions

The translation of *trans*($E_1 == E_2, l, t, f$) is given by

```
l: E_1
   E_2
   if_icmpeq true
   ldc_int 0
   goto f
   true:
   ldc_int 1
   goto t
```

Using the idea of short circuiting and swapping labels, we can define the following equivalences

- *trans*(`!` E, l, t, f) = *trans*(E, l, f, t)
- *trans*(E_1 `&&` E_2, l, t, f) = *trans*(E_1, l, l', f), *trans*(E_2, l', t, f)
- *trans*(E_1 `||` E_2, l, t, f) = *trans*(E_1, l, t, l'), *trans*(E_2, l', t, f)

Jumping code can be longer in comparison but for each branch it will usually execute less instructions.

Template - Plus Expression

E_1 + E_2

Template (E_i has type `int`)

E_1
E_2
`iadd`

Template (otherwise)

E_1
E_2
`invokevirtual java/lang/String/concat(Ljava/lang/String;)Ljava/lang/String;`

JOOS source

```
case plusK:
   codeEXP(e->val.plusE.left);
   codeEXP(e->val.plusE.right);
   if (e->type->kind == intK) {
      code_iadd();
   } else {
      code_invokevirtual("java/lang/.../String;");
   }
   break;
```

(A separate test of an `e->tostring` field is used to handle string coercion)

Template - `this`

```
this
```

Template

```
aload 0
```

JOOS source

```
case thisK:
   code_aload(0);
   break;
```

Template - `null`

```
null
```

Template (`toString` coerced)

```
ldc_string "null"
```

Template (otherwise)

```
aconst_null
```

JOOS source

```
case nullK:
   if (e->tostring) {
      code_ldc_string("null");
   } else {
      code_aconst_null();
   }
   break;
```

Template - `toString` coercion

E

Template (E has type `int`)

```
new java/lang/Integer
dup
E
invokespecial java/lang/Integer/⟨init⟩(I)V
invokevirtual java/lang/Integer/toString()Ljava/lang/String;
```

Template (E has type `boolean`)

```
new java/lang/Boolean
dup
E
invokespecial java/lang/Boolean/⟨init⟩(Z)V
invokevirtual java/lang/Boolean/toString()Ljava/lang/String;
```

Template - `toString` coercion

E

Template (***E*** has type `char`)

```
new java/lang/Character
dup
E
invokespecial java/lang/Character/⟨init⟩(C)V
invokevirtual java/lang/Character/toString()Ljava/lang/String;
```

Template (otherwise)

```
E
dup
ifnull nulllabel
invokevirtual  signature(class(E), toString)
goto stoplabel
nulllabel:
pop
ldc_string "null"
stoplabel:
```

Template - Method Call

$E.m(E\_1,...,E\_n)$

Template

E
E_1
.
.
.
E_n
`invokevirtual` *signature(class(*E*),*m*)*

Definitions

- *class(*E*)* is the declared class of E
- *signature(C,*m*)* is the signature of the first implementation of m that is found from *C*

Template - Super Method Calls

`super.`m`(`E_1`,...,`E_n`)`

Template

```
aload 0
```
E_1

.

.

.

E_n

```
invokespecial  signature(parent(thisclass)),
```
m`)`

Definitions

- *`thisclass`* is the current class
- *`parent(C)`* is the parent of *`C`* in the hierarchy
- *`signature(C,`*m*`)`* is the signature of the first implementation of m that is found from *`parent(C)`*

Template - Method Calls

JOOS source

```
case invokeK:
   codeRECEIVER(e->val.invokeE.receiver);
   codeARGUMENT(e->val.invokeE.args);
   switch (e->val.invokeE.receiver->kind) {
      case objectK: {
         SYMBOL *s = lookupHierarchyClass(
            e->val.invokeE.method->name,
            e->val.invokeE.receiver->objectR->type->class
         );
         code_invokevirtual(codeMethod(s, e->val.invokeE.method));
         break;
      }
      case superK: {
         CLASS *c = lookupHierarchyClass(
            e->val.invokeE.method->name,
            currentclass->parent
         );
         code_invokenonvirtual(codeMethod(c, e->val.invokeE.method));
         break;
      }
   }
   break;
```

Signature of a Method

The signature of a method `m` in a class `C` with argument types $\tau_1, \ldots, \tau_k$ and return type τ is represented in Jasmin as

$$\texttt{C/m(}\mathit{rep}(\tau_1)\ldots\mathit{rep}(\tau_k)\texttt{)}\,\mathrm{rep}(\tau)$$

The representation of each type in Jasmin code is given by

- *rep(*`int`*)* = `I`
- *rep(*`boolean`*)* = `Z`
- *rep(*`char`*)* = `C`
- *rep(*`void`*)* = `V`
- *rep(*`C`*)* = `LC;`

Example

The `char charAt(int index)` method as part of the `String` class has signature

```
java/lang/String/charAt(I)C
```

Computing the Stack Limit

```
public void Method() {
   int x, y;
   x = 12;
   y = 87;
   x = 2 * (x + y * (x - y));
}
```

```
.method public Method()V
.limit locals 3
.limit stack 5
   ldc 12        ← 1
   dup           ← 2
   istore_1      ← 1
   pop           ← 0
   ldc 87        ← 1
   dup           ← 2
   istore_2      ← 1
   pop           ← 0
   iconst_2      ← 1
   iload_1       ← 2
   iload_2       ← 3
   iload_1       ← 4
   iload_2       ← [5]
   isub          ← 4
   imul          ← 3
   iadd          ← 2
   imul          ← 1
   dup           ← 2
   istore_1      ← 1
   pop           ← 0
   return
.end method
```

Computing the Stack Limit

The stack limit is the maximum height of the stack during the evaluation of an expression in the method.

This requires detailed knowledge of

- The code that is generated; and
- The virtual machine.

Stupid A- JOOS source

```
int limitCODE(CODE *c) {
   return 25;
}
```

The A+ source code computes the liit using a recursive traversal, simulating the effect of operations on the stack.

Emitting Code

Code is emitted in Jasmin format following the structure of a class file

```
.class public C
.super parent(C)

.field protected x_1 type(x_1)
.
.
.
.field protected x_k type(x_k)

.method public m_1 signature(C,m_1)
.limit locals l_1
.limit stack s_1
  S_1
.end method
.
.
.
.method public m_n signature(C,m_n)
.limit locals l_n
.limit stack s_n
  S_n
.end method
```

Tiny JOOS Example Class

```
import joos.lib.*;

public class Tree {
   protected Object value;
   protected Tree left;
   protected Tree right;

   public Tree(Object v, Tree l, Tree r) {
      super();
      value = v;
      left = l;
      right = r;
   }

   public void setValue(Object newValue) {
      value = newValue;
   }
}
```

Tiny JOOS Example Class - Jasmin File

```
.class public Tree
.super java/lang/Object
.field protected value Ljava/lang/Object;
.field protected left LTree;
.field protected right LTree;

.method public <init>(Ljava/lang/Object;LTree;LTree;)V
.limit locals 4
.limit stack 3
  aload_0
  invokenonvirtual java/lang/Object/<init>()V
  aload_0
  aload_1
  putfield Tree/value Ljava/lang/Object;
  aload_0
  aload_2
  putfield Tree/left LTree;
  aload_0
  aload_3
  putfield Tree/right LTree;
  return
.end method

[...]
```

Tiny JOOS Example Class - Jasmin File

```
[...]

.method public setValue(Ljava/lang/Object;)V
.limit locals 2
.limit stack 3
  aload_0
  aload_1
  putfield Tree/value Ljava/lang/Object;
  return
.end method
```

Hex Dump of Class File

```
cafe babe 0003 002d 001a 0100 064c 5472
6565 3b07 0010 0900 0200 0501 0015 284c
6a61 7661 2f6c 616e 672f 4f62 6a65 6374
3b29 560c 0018 0001 0100 0654 7265 652e
6a01 000a 536f 7572 6365 4669 6c65 0100
0443 6f64 6507 000d 0c00 0e00 1209 0002
0017 0100 2128 4c6a 6176 612f 6c61 6e67
2f4f 626a 6563 743b 4c54 7265 653b 4c54
7265 653b 2956 0100 106a 6176 612f 6c61
6e67 2f4f 626a 6563 7401 0005 7661 6c75
650c 0011 0019 0100 0454 7265 6501 0006
3c69 6e69 743e 0100 124c 6a61 7661 2f6c
616e 672f 4f62 6a65 6374 3b0a 0009 000f
0100 0873 6574 5661 6c75 6509 0002 000a
0100 046c 6566 740c 0016 0001 0100 0572
6967 6874 0100 0328 2956 0021 0002 0009
0000 0003 0006 000e 0012 0000 0006 0016
0001 0000 0006 0018 0001 0000 0002 0001
0011 000c 0001 0008 0000 0020 0003 0004
0000 0014 2ab7 0013 2a2b b500 152a 2cb5
000b 2a2d b500 03b1 0000 0000 0001 0014
0004 0001 0008 0000 0012 0003 0002 0000
0006 2a2b b500 15b1 0000 0000 0001 0007
0000 0002 0006
```

djas -w Tree.class

```
; magic number 0xCAFEBABE
; bytecode version 45.3

; constant pool count 26
; cp[1] (offset 0xf) -> CONSTANT_NameAndType 9, 25
; cp[2] (offset 0x22) -> CONSTANT_Utf8 "java/lang/Object"
; cp[3] (offset 0x27) -> CONSTANT_Fieldref 12, 21
; cp[4] (offset 0x30) -> CONSTANT_Utf8 "<init>"
; cp[5] (offset 0x3b) -> CONSTANT_Utf8 "setValue"
; cp[6] (offset 0x3e) -> CONSTANT_Class 2
; cp[7] (offset 0x43) -> CONSTANT_NameAndType 4, 10
; cp[8] (offset 0x48) -> CONSTANT_Fieldref 12, 1
; cp[9] (offset 0x4f) -> CONSTANT_Utf8 "left"
; cp[10] (offset 0x55) -> CONSTANT_Utf8 "()V"
; cp[11] (offset 0x5c) -> CONSTANT_Utf8 "Code"
; cp[12] (offset 0x5f) -> CONSTANT_Class 22
; cp[13] (offset 0x64) -> CONSTANT_Fieldref 12, 18
; cp[14] (offset 0x71) -> CONSTANT_Utf8 "SourceFile"
; cp[15] (offset 0x86) -> CONSTANT_Utf8 "Ljava/lang/Object;"
; cp[16] (offset 0xaa) -> CONSTANT_Utf8 "(Ljava/lang/Object;LTree;LTree;)V"
; cp[17] (offset 0xaf) -> CONSTANT_Methodref 6, 7
; cp[18] (offset 0xb4) -> CONSTANT_NameAndType 24, 25
; cp[19] (offset 0xcc) -> CONSTANT_Utf8 "(Ljava/lang/Object;)V"
; cp[20] (offset 0xd4) -> CONSTANT_Utf8 "value"
; cp[21] (offset 0xd9) -> CONSTANT_NameAndType 20, 15
```

```
; cp[22] (offset 0xe0) -> CONSTANT_Utf8 "Tree"
; cp[23] (offset 0xe9) -> CONSTANT_Utf8 "Tree.j"
; cp[24] (offset 0xf1) -> CONSTANT_Utf8 "right"
; cp[25] (offset 0xfa) -> CONSTANT_Utf8 "LTree;"

; access flags = 0x21 [ ACC_SUPER ACC_PUBLIC ]
; this_class index = 12
; super_class index = 6

; interfaces_count = 0


; fields_count = 3

; fields[0] (offset 0x104) :
;     access_flags 0x4 [ ACC_PROTECTED ]
;     name_index 20 (value)
;     descriptor_index 15 (Ljava/lang/Object;)
;     attributes_count 0

; fields[1] (offset 0x10c) :
;     access_flags 0x4 [ ACC_PROTECTED ]
;     name_index 9 (left)
;     descriptor_index 25 (LTree;)
;     attributes_count 0

; fields[2] (offset 0x114) :
;     access_flags 0x4 [ ACC_PROTECTED ]
```

```
;     name_index 24 (right)
;     descriptor_index 25 (LTree;)
;     attributes_count 0

; methods_count 2

; methods[0] (offset 0x11e) :
;     access_flags 0x1 [ ACC_PUBLIC ]
;     name_index 4 (<init>)
;     descriptor_index 16 ((Ljava/lang/Object;LTree;LTree;)V)
;     attributes_count 1
;     method_attributes[0] :
;         name_index 11 (Code)
;         attribute_length 41
;         max_stack 3
;         max_locals 4
;         code_length 29
;         code :
;                0: aload_0
;                1: invokespecial 17 (java/lang/Object/<init> ()V)
;                4: aload_1
;                5: dup
;                6: aload_0
;                7: swap
;                8: putfield 3 (Tree/value Ljava/lang/Object;)
;               11: pop
;               12: aload_2
```

```
;                13: dup
;                14: aload_0
;                15: swap
;                16: putfield 8 (Tree/left LTree;)
;                19: pop
;                20: aload_3
;                21: dup
;                22: aload_0
;                23: swap
;                24: putfield 13 (Tree/right LTree;)
;                27: pop
;                28: return
;          exception_table_length 0
;          attributes_count 0


; methods[1] (offset 0x155) :
;      access_flags 0x1 [ ACC_PUBLIC ]
;      name_index 5 (setValue)
;      descriptor_index 19 ((Ljava/lang/Object;)V)
;      attributes_count 1
;      method_attributes[0] :
;          name_index 11 (Code)
;          attribute_length 21
;          max_stack 3
;          max_locals 2
;          code_length 9
```

```
;          code :
;                 0: aload_1
;                 1: dup
;                 2: aload_0
;                 3: swap
;                 4: putfield 3 (Tree/value Ljava/lang/Object;)
;                 7: pop
;                 8: return
;          exception_table_length 0
;          attributes_count 0



; attributes_count 1

; class_attributes[0] (offset 0x17a) :
;      name_index 14 (SourceFile)
;      attribute_length 2
;      sourcefile_index 23

; End of file reached successfully. Enjoy :)
```

Testing Strategy

The testing strategy for the code generator involves two phases

1. A careful argumentation that each code template is correct; and
2. A demonstration that each code template is generated correctly.