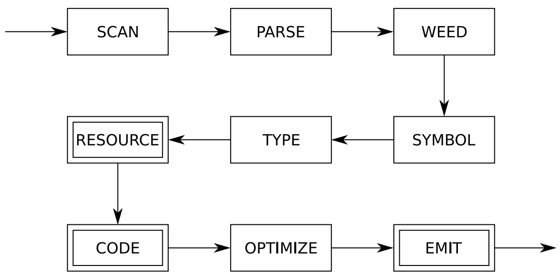
Code generation



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 in JOOS:

- offsets for locals and formals;
- labels for control structures; and
- local and stack limits for methods.

These are values that cannot be computed based on a single statement.

We must perform a global traversal of the parse trees.

Computing offsets and the locals limit:

```
public class Example {

  public Example() { super(); }

  public void Method(int p[1], int q[2], Example r[3]) {
     int x[4];
     int y[5];
     { int z[6];
       z = 87;
     }
     { boolean a[6];
       Example x[7];
       { boolean b[8];
         int z[9];
         b = true;
       }
       { int y[8];
         y = x;
       }
     }
  }
}
```

The locals limit is the largest offset generated in the method + one extra slot for `this`.

Corresponding JOOS source:

```
int offset, localslimit;

int nextoffset()
{ offset++;
  if (offset > localslimit) localslimit = offset;
  return offset;
}

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

void resID(ID *i)
{ if (i!=NULL) {
     resID(i->next);
     i->offset = nextoffset();
  }
}

...

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


Computing labels for control structures:

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

Corresponding JOOS source:

```
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;

.
.
.
```



```
typedef struct CODE {
   enum {nopCK,i2cCK,
         newCK,instanceofCK,checkcastCK,
         imulCK,inegCK,iremCK,isubCK,idivCK,iaddCK,iincCK,
         labelCK,gotoCK,ifeqCK,ifneCK,
         if_acmpeqCK,if_acmpneCK, ifnullCK,ifnonnullCK,
         if_icmpeqCK,if_icmpgtCK,if_icmpltCK,
         if_icmpleCK,if_icmpgeCK,if_icmpneCK,
         ireturnCK,areturnCK,returnCK,
         aloadCK,astoreCK,iloadCK,istoreCK,dupCK,popCK,
         swapCK,ldc_intCK,ldc_stringCK,aconst_nullCK,
         getfieldCK,putfieldCK,
         invokevirtualCK,invokenonvirtualCK} kind;
   union {
     char *newC;
     char *instanceofC;
     char *checkcastC;
     struct {int offset; int amount;} iincC;
     int labelC;
     int gotoC;
     int ifeqC;
     ...
     int istoreC;
     int ldc_intC;
     char *ldc_stringC;
     char *getfieldC;
     char *putfieldC;
     char *invokevirtualC;
     char *invokenonvirtualC;
   } val;
   struct CODE *next;
} CODE;
```

Code templates:

- show how to generate code for each language construct;
- ignore the surrounding context; and
- dictate a simple, recursive strategy.

Code template invariants:

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

Special case of `ExpressionStatement`:

- Expression is evaluated, result is then popped off the stack, except
- for `void` return expressions, nothing is popped.

The statement:

```
if (E) S
```

has code template:

```
E
ifeq stop
S
stop:
```

Corresponding 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;
```


The statement:

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

has code template:

```
E
ifeq else
S_1
goto stop
else:
S_2
stop:
```

Corresponding 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;
```


The statement:

```
while (E) S
```

has code template:

```
start:
E
ifeq stop
S
goto start
stop:
```

Corresponding 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;
```

The statement:

E

has code template:

E

if ***E*** has type `void` and otherwise:

```
E
pop
```

Corresponding JOOS source:

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


The local variable expression:

x

has code template:

```
iload  offset(x)
```

if ***x*** has type `int` or `boolean` and otherwise:

```
aload  offset(x)
```

Corresponding 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;
```


The assignment to a formal:

x=E

is an expression on its own and has code template:

```
E
dup
istore  offset(x)
```

if ***x*** has type `int` or `boolean` and otherwise:

```
E
dup
astore  offset(x)
```

Corresponding 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;
```


The expression:

E_1* || *E_2

has code template:

```
E_1
dup
ifne true
pop
E_2
true:
```

Corresponding 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;
```

The expression:

```
E_1 == E_2
```

has code template:

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

if E_i has type `int` or `boolean`.

Corresponding 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;
```


The expression:

```
E_1 + E_2
```

has code template:

```
E_1
E_2
iadd
```

if E_i has type `int` and otherwise:

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

Corresponding 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.)

The expression:

```
!E
```

has code template:

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

Corresponding 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;
```


Alternative translation of Boolean expressions: Short-circuit or Jumping code

Motivating example: Expression

$$!!!!!!!E$$

would generate lots of jumps when using the template described earlier. (Other Boolean operations, too.)

Idea: Can encode Boolean logic by more clever introduction and swaps of labels.

Use function $\mathit{trans}(b, l, t, f)$ with:

b Boolean expression

l label for evaluating current expression

t jump-label in case b evaluates to **true**

f jump-label in case b evaluates to **false** *trans*($\boldsymbol{E_1}$ == $\boldsymbol{E_2}$, $\boldsymbol{l}$, $\boldsymbol{t}$, $\boldsymbol{f}$)=

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

trans(!$\boldsymbol{E}$, $\boldsymbol{l}$, $\boldsymbol{t}$, $\boldsymbol{f}$)= *trans*($\boldsymbol{E}$, $\boldsymbol{l}$, $\boldsymbol{f}$, $\boldsymbol{t}$)

trans($\boldsymbol{E_1}$ && $\boldsymbol{E_2}$, $\boldsymbol{l}$, $\boldsymbol{t}$, $\boldsymbol{f}$)=
trans($\boldsymbol{E_1}$, $\boldsymbol{l}$, $\boldsymbol{l'}$, $\boldsymbol{f}$), *trans*($\boldsymbol{E_2}$, $\boldsymbol{l'}$, $\boldsymbol{t}$, $\boldsymbol{f}$)

trans($\boldsymbol{E_1}$ || $\boldsymbol{E_2}$, $\boldsymbol{l}$, $\boldsymbol{t}$, $\boldsymbol{f}$)=
trans($\boldsymbol{E_1}$, $\boldsymbol{l}$, $\boldsymbol{t}$, $\boldsymbol{l'}$), *trans*($\boldsymbol{E_2}$, $\boldsymbol{l'}$, $\boldsymbol{t}$, $\boldsymbol{f}$)

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

The expression:

```
this
```

has code template:

```
aload 0
```

Corresponding JOOS source:

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


The expression:

```
null
```

has code template:

```
ldc_string "null"
```

if it is `toString` coerced and otherwise:

```
aconst_null
```

Corresponding JOOS source:

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


The expression:

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

has code template:

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

class(E) is the declared class of $\boldsymbol{E}$.

signature(C,m) is the signature of the first implementation of $\boldsymbol{m}$ that is found from *C*.

The expression:

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

has code template:

```
aload 0
E_1
.
.
.
E_n
invokespecial  signature(parent(thisclass)),m)
```

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)*.

Corresponding JOOS source:

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


A `toString` coercion of the expression:

    E

has code template:

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

if ***E*** has type `int`, and:

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

if ***E*** has type `boolean`, and:

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

if ***E*** has type `char`.

A `toString` coercion of the expression:

    E

has code template:

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

if ***E*** does not have type `int`, `boolean`, or `char`.

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
```


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;
}
```


Code is emitted in Jasmin format:

```
.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
```


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:

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

where:

- $\mathit{rep}(\texttt{int}) = \texttt{I}$
- $\mathit{rep}(\texttt{boolean}) = \texttt{Z}$
- $\mathit{rep}(\texttt{char}) = \texttt{C}$
- $\mathit{rep}(\texttt{void}) = \texttt{V}$
- $\mathit{rep}(\texttt{C}) = \texttt{LC;}$

A tiny JOOS 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; }
}
```


The compiled 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

.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 the 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
```


The testing strategy for the code generator involves two phases.

First a careful argumentation that each code template is correct.

Second a demonstration that each code template is generated correctly.