Part VII : Code Generation

- Code Generation
- Stack vs Register Machines
- JVM Instructions
- Code for arithmetic Expressions
- Code for variable access
- Indexed variables
- Code for assignments
- Items
- How to use items in the compiler

Code Generation

- So far, a compiler only determined whether a given source program was legal according to the rules of a programming language.
- This is called *analysis.*
- We now discuss the second task of a compiler: translation to a directly executable *target language*.
- This is called *synthesis*
- Two kinds of target languages: Machine languages or intermediate languages.
- Machine languages can be executed by hardware (*concrete machines*), intermediate languages are either interpreted (by *virtual or abstract machines*) or compiled further.

Stack vs Register Machines

- Today's machines are either stack-oriented or register-oriented.
- Stack machine's operations:
  - Load value on stack.

    Example: **`iload 5`** loads local variable at address 5 on top of stack (TOS).
  - Operate on values on top of stack, replacing them with the result of the operation.

    Example: **`iadd`** Adds two integers on TOS.
  - Store top of stack into memory.

    Example: **`istore 3`** stores TOS into local variable at address 3.

Stack vs Register Machines (2)

- Register machines are often organized as *load-store* machines.
- Example operations:
  - Load value into register.

    Example `LDW R1, 5[R2]` loads variable at offset 5 from register R2 into register R1.
  - Operate on values in registers and place result into another register.

    Example: `ADD R1, R4, R5`. Adds two integers in R4 and R5, placing result into R1.
  - Store register into memory.

    Example: `STW R1, 3[R2]` stores R1 into variable at offset 3 from register R2.

JVM Instructions

- We will generatie intermediate code for the Java Virtual Machine (JVM)
- Programs for the JVM are represented as *class files*; they have extension `class`.
- The instructions in class files are called *byte codes*. They control a stack machine.
- Here 's a <u>summary of useful by code instructions</u>

Code for arithmetic Expressions

Example: Translate expression x + y * z

Code	Contents of stack
iload x	x
iload y	x y
iload z	x y z
imul	x (y*z)
iadd	(x+y*z)

Code for arithmetic Expressions (2)

Code templates:	code (E)
`E(t) = Operation Add` E_1 `(`t_1`)` E_2 `(`t_2`)`	`code (`E_1`); code(`E_2`); iadd`
`	Operation Sub` $E_1$ `(`$t_1$`)` $E_2$ `(`$t_2$`)`	`code (`E_1`); code(`E_2`); isub`
`	Operation Mul` $E_1$ `(`$t_1$`)` $E_2$ `(`$t_2$`)`	`code (`E_1`); code(`E_2`); imul`
`	Operation Neg` $E_1$ `(`$t_1$`) null`	`code (`E_1`); ineg`
`	...`	

Simple Code Generation Visitor

```
Class CodeGen implements Tree.Visitor {
  /** the visitor method
   */
  public void genCode (Tree tree) {
    tree.apply(this);
  }
  /** an example case
   */
  Item caseOperation(Operation tree) {
    genCode(tree.left);
    if (tree.right != null) genCode(tree.right);
    switch (tree.operator) {
    case ADD: code.emit1(iadd); break;
    case SUB: code.emit1(isub); break;
    ...
    }
  }
}
```

Local variable access

- Variables are treated according to where they come from:
- *Local* variables are accessed by offsets: **`iload, aload`**.
- There are two kinds of load instructions, one for integers (iload), the other for *references*, i.e. objects, arrays (aload).
- Local variables have addresses numbered 0, 1, 2, 3, …
- 0 is Address of first parameter, followed by other parameters and local variables.
- Example :

```
void f(int x, int y) {          adr(x) = 0
   int z;                       adr(y) = 1
   z = 10;                      adr(z) = 2
   while (z > 0) {              adr(u) = 3
       int u;
       u = u + z;
       z = z - 1
   }
}
```

Global Variable Access

- *Global* variables are accessed by a constant pool index, which points to their name: **`getstatic`**.
- The mapping of names to addresses is done by the JVM at load-time.
- There is only one load instruction for global variables; the type of the variable is kept along with its name.
- Reason for difference: Separate compilation.

Code Example

- Consider the JO program

```
module M {
   int x;
   int f(int y) {
      return (x + x) * (y + y) / 2;
   }
}
```

- Which code is generated for the returned expression ?

Indexed Variables

- Indexed variables are loaded by
  - (1.) loading the array address,
  - (2.) loading the index,
  - (3.) executing an indexed load instruction (**`iaload, aaload`** depending on whether the element type of the array is integer or a reference.
- Example : a[i][j] * b[i] translates to :

Instructions	Stack contents
`aload a`	
`iload i`	
`aaload`	
`iload j`	
`iaload`	
`aload b`	
`iload i`	
`iaload`	
`imul`	

Code Generation for Simple Expressions

```
E(t) = ...
    | Ident (sym, t)              if ("sym is local")
                                     if (t == int) iload(slot(sym))
                                     else aload(slot(sym)
                                  else
                                     getstatic(mkref(sym))

    | Indexed E1(t1) E2(t2)       code(E1); code(E2);
                                  if (t == int) iaload else aaload

    | NumLit (intval)             bipush(intval) or
                                  sipush(intval) or
                                  ldc1(mkref(intval))

    | StrLit (strval)             ldc1(mkref(strval))

    | NewArray T E1               code (E1) ; newarray(arrayCode(T))
```

Code for assignments

Question: what's the code for an assignment `x = y + z`?

Translated code:

```
iload #y
iload #z
iadd
istore #x
```

What about `a[i] = y + z`?

```
aload #a
iload #i
iload #y
iload #z
iadd
iastore
```

The store comes last, depends on the left-hand side of assignment which was generated earlier.

Need to delay generation of code.

Items (1)

- An *item* is a structure which represents some partially generated code.
- Items offer methods that complete code generation in a number of ways.
- Example Implementation:

```
abstract class Item {
  Type type;
  void load();  // complete code by loading
                // value on stack
  void store(); // complete code by storing into
                // variable represented by this item.
                // this will raise an exception if
                // the item does not represent a variable.
  }
```

Items (2)

- Concrete subclasses of this class will represent the various prossibilities of code generation.
- Here's a simple one:

```
/** A class for things that are already on stack. */
class StackItem extends Item {
   void load() {
     // do nothing, item is already on stack
   }
   void store() {
     throw new InternalError("store to stack item");
   }
}
```

Items (3)

- Here's another one:

```
/** A class for items that represent local variables. */
class LocalItem extends Item {
    int slot;
    Type type;
    void load() {
      code.emit1(
        type.tag == Type.INT ? Code.iload : Code.aload);
      code.emit1(slot);
    }
    void store() {
      code.emit1(
        type.tag == Type.INT ? Code.istore:Code.astore);
      code.emit1(slot);
    }
}
```

Items (4)

- And here's a third one:

```
class IndexedItem extends Item {
    Type type;
    void load() {
      code.emit1(
        type.tag == Type.INT ? Code.iaload : Code.aaload);
    }
    void store() {
      code.emit1(
        type.tag == Type.INT ? Code.iastore : Code.aastore);
    }
}
```

How to use items in the compiler

- All visitor methods handling expressions now return an item as result.
- The item returned from a visitor of an expression tells how to complete code generation for that expressions.
- That is, the code generation class should start as follows :

```
class CodeGen implements Tree.Visitor {
  Item result; // to be assigned
       // in each expression case
  /** the visitor method
   */
  public Item genCode(Tree tree) {
       tree.apply(this);
       return result;
  }
  ...
}
```

Example Cases :

```
void caseOperation(Operation tree) {
   genCode(tree.left).load();
   if (tree.right != null) ri = genCode(tree.right)load();
   switch (tree.operator) {
   case ADD: code.emit1(iadd); break;
   case SUB: code.emit2(isub); break;
   ...
   }
   result = new StackItem(); // or reuse single one
}
void caseIdent(Ident tree) {
   Symbol sym = tree.sym;
   if (sym.isGlobal()) result = new GlobalItem(sym)
   else result = new LocalItem(sym.type, sym.adr)
}
void caseIndexed(Indexed tree) {
   genCode(tree.left).load();
   genCode(tree.right).load();
   result = new IndexedItem(tree.type);
}
```

The JO Code Generation Framework : Class Code

```
Public class Code {

/** the slot number for the next available local variable slot */
     public int nextSlot;

/** emit a byte of code: */
     public void emit1(int x) { ... }

/** emit two bytes of code: */
     public void emit2(int x) { ... }

/** the current size of code
  */
     public int size() { ... }

/** generate code to load an integer literal on stack */
     public void loadLiteral(int x) { ... }

/** generate code to load a string literal on stack */
     public void loadLiteral (String x) { ... }

/** generate code to jump with given opcode;
     return address of jump target field */
     public int jump(int opcode) { ... }
```

The JO Code Generation Framework : Class *Code* (2)

```
/** patch jump target address field */
    public void patch(int fieldIndex, int target) { ... }

/** generate code to allocate array with given signature */
    public void allocArray(String sig) { ... }

/** Bytecode constants */
    public static final int
        nop                = 0,
        aconst_null        = 1,
        ...
        jsr_w              = 201,
        breakpoint         = 202;
}
```

Translation of Control Statements

- If-then:

Statement

```
if (x < y) S1;
```

Code

```
     iload #x
     iload #y
     if_icmpge L1
     code (S1)
L1:
```

- if-then-else:

Statement

```
if (x == y) S1; else S2;
```

Code

```
     iload #x
     iload #y
     if_icmpne L1
     code(S1)
     goto L2
L1:
     code(S2)
L2:
```

Translation of Control Statement (2)

- While:

Statement	Code
`while (x != y) S1;`	

```
L1:
        iload #x
        iload #y
        if_icmpeq L2
        code(S1)
        goto L1
L2:
```

or, slightly more efficient:

Statement	Code
`while (x != y) S1;`	

```
        goto L2:
L1:
        code(S1)
L2:     iload #x
        iload #y
        if_icmpne L1
```

Jumps in the JVM

- The JVM has a variety of conditional and unconditional jump instructions.
- Jump addresses are relative in the JVM.
- A jump contains is the difference between the address of the jump instruction and the address of the target (as a signed 2-byte integer).
- The code generation framework abstracts from the precise layout.
- Three relevant operations :

```
int jump(int opcode)
int size()
void patch(int fieldIndex, int target)
```

Translation of Jumps

- Depends whether we jump forward or backwards.
- For a forward jump :

```
        int jumpAdr = code.jump(opcode)
        ...
    L:
        int label = code.size();
        code.patch(jumpAdr, label)
```

- For a backward jump :

```
    L:
        int label = code.size();
        ...
        int jumpAdr = code.jump(opcode);
        code.patch(jumpAdr, label);
```

Translation of Boolean Expressions

- Sometimes, a boolean expression is not used as the condition of a control statement, but is used as an assigned value instead.
- Example :

Statement	Code
`int b;`	

`b = (y > z);`	`iload #y`
	`iload #z`
	`if_icmple L1`
	`iconst_1`
	`goto L2`
	`L1:`
	`iconst_0`
	`L2:`
	`istore #b`

- *As* the example shows, Boolean expressions are translated to if-then-else sequences with two branches. One branch returns *true* (coded as 1), the other *false* (coded as 0).

Translation of Single Variable Constrol Conditions

- Question : how do we translate `if (cond) S` ?
- Answer : Use `ifeq` instruction.

Statement	Code

`if (b) S`	`iload #b`
	`ifeq L1`
	
	`code (S)`
	`L1:`

Translation of complex conditions

- Question : how do we translate `(x < y) & (u == v)` ?
- Answer : Convert each operand of the & to a value, then use `iand`.

Expression	Code
`(x < y) & (u == v)`	

```
        iload #x
        iload #y
        if_icmpge L1
        iconst 1
        goto L2
L1:     iconst 0
L2:     iload #u
        iload #v
        if_icmpne L3
        iconst 1
        goto L4
L3:     iconst 0
L4:     iand
```

Translation of complex conditions (2)

- How do we control switching back between the value view (one boolean value on stack) and the comparison view (two values on stack, to be compared)?
- *« Simple-Minded » solution* : We don 't. Conditions always evaluate to either 0 or 1. We always jump with **`ifeq`**.
- *Complex, but more efficient solution* : We extend the item framework to items that encapsulate conditions (see next pages).

Conditional Items

- A **`CondItem`** represents a condition with either one or two values on stack.
- It also contains the jump instruction to be used if the condition is *false*.
- Like all other items, **`CondItems`** have **`load`** and **`store`** methods.
- Furthermore, all other items need to be able to turn themselves into a **`CondItem`**.
- This is achieved by defining a method

```
CondItem makeCondition()
```

  in class **`Item`**.

The Abstract Class Item

- This is example code - feel free to use a different design.

```
Abstract class Item {
    static Code code; // the current code structure;
                         set from outside
    Type type;
    Item(Type type) { this.type = type; }
    abstract void load();
    abstract void store();
    CondItem makeCondition() {
        if (type.tag != Type.INT)
            throw new InternalError("make condition from non int");
        load();
        return new CondItem(Code.ifeq);
    }
}
```

An Item Class for Things on Stack

```
Class StackItem extends Item {

     StackItem(Type type) {
          super(type);

     }

     void load() {
          if (type.tag == Type.VOID)
               throw new InternalError("void load");
     }

     void store() { throw new InternalError("store to stack item");
     }
}
```

An Item Class for Local Variables

```
Class LocalItem extends Item {

     int slot;

     LocalItem(Symbol sym) {
          super(sym.type);
          this.slot = sym.slot();
     }

     void load() {
          code.emit1(type.tag == Type.INT ? Code.iload : code.aload);
          code.emit1(slot);
     }

     void store() {
          code.emit1(type.tag == Type.INT ? Code.istore : Code.astore)
          code.emit1(slot);
     }
}
```

An Item Class for Global Variables

```
Class GlobalItem extends Item {

     int poolIndex;

     GlobalItem(Symbol e) {
          super(e.type);
          this.poolIndex = e.poolIndex();
     }

     void load() {
          code.emit1(Code.getstatic);
          code.emit2(poolIndex);
     }

     void store() {
          code.emit1(Code.putstatic);
          code.emit2(poolIndex);
     }
}
```

An Item Class for Array Elements

```
Class IndexedItem extends Item {

    IndexedItem(type type) {
        super(type);
    }

    void load() {
        code.emit1(type.tag == Type.INT ? code.iaload : code.aaload)
    }

    void store() {
        code.emit1(type.tag == Type.INT ? code.iastore : code.aastore)
    }
}
```

An Item Class for Conditions

```
Class CondItem extends Item {

    int jumpCode; // the opcode of a "jump if this condition is false"

    CondItem(int jumpCode) {
        super(Type.intType);
        this.jumpCode = jumpCode;
    }

    void load() {
        int jumpAdr1 = jumpIfFalse();
        code.emit1(Code.iconst_1);
        int jumpAdr2 = code.jump(Code.goto_);
        code.patch(jumpAdr1, code.size());
        code.emit1(Code.iconst_0);
        code.patch(jumpAdr2, code.size());
    }

    void store() { throw new InternalError("store to cmp item"); }

    CondItem makeCondition() {
        return this;
}
```

An Item Class for Conditions (2)

```
    int jumpIfFalse() {
        return code.jump(jumpCode);
    }

    CondItem negate() {
        return new CondItem(((jumpCode + 1) ^ 1) - 1);
    }
}
```

Method Invocation

Method parameters are passed on the stack.

Parameters are pushed in the order they appear on the parameter list.

Example : Given

```
int min(int x, int y)
```

we have the following translation :

Statement	Code

`a = min(a, b)`	`iload #a`
	`iload #b`
	`invokestatic #min`
	`istore #a`

The `invokestatic` operation takes a 2-byte constant pool index of the method to be called as operand.

This is analogous to `getstatic` and `putstatic`.