

Code Generation

Dr. Phung Nguyen

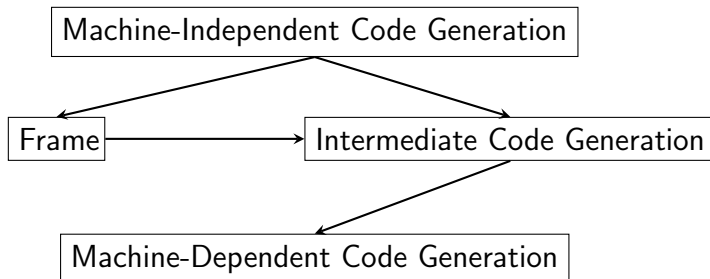
Faculty of Computer Science and Engineering
University of Technology
HCMC Vietnam National University

November 25, 2020

Outline

1 Translation to a stack-based machine

Code Generation Design

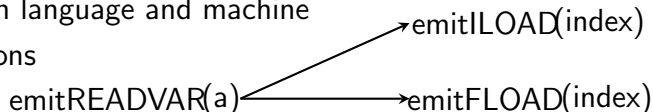


Machine-Dependent Code Generation

- Generating specified machine code
E.g.: `emitLDC(20)` \rightarrow `"ldc 20"`
- Implemented in `JasminCode`

Intermediate Code Generation

- Depend on both language and machine
- Select instructions



- Select data objects
- Simulate the execution of the machine
 - ◁ `emitICONST` → `push()`
 - ◁ `emitISTORE` → `pop()`
- Implemented in class `Emitter`

Directives Generation APIs

- emitVAR(self, index, varName, inType, fromLabel, toLabel)
.var 0 is this Lio; from Label0 to Label1
- emitATTRIBUTE(self, lexeme, inType, isFinal, value = None)
.field public static writer Ljava/io/Writer;
- emitMETHOD(self, lexeme, inType, isStatic)
.method public foo(I)I
- emitENDMETHOD(self, frame)
.limit stack 1
.limit locals 1
.end method
- emitPROLOG(self, name, parent)
.source io.java
.class public io
.super java/lang/Object
- emitEPILOG(self)

Type

- `class IntType(Type)`
- `class FloatType(Type)`
- `class StringType(Type)`
- `class VoidType(Type)`
- `class BoolType(Type)`
- `class ClassType(Type): # cname:str`
- `class ArrayType(Type): # eleType:Type,dimen:List[int]`
- `class MType(Type): # partype:List[Type],rettype:Type`

Operation Generation APIs

- `emitADDOP(self, lexeme, inType, frame) \Rightarrow iadd, fadd, isub, fsub`
- `emitMULOP(self, lexeme, inType, frame) \Rightarrow imul, fmul, idiv, fdiv`
- `emitDIV(self, frame) \Rightarrow idiv`
- `emitMOD(self, frame) \Rightarrow irem`
- `emitANDOP(self, frame) \Rightarrow iand`
- `emitOROP(self, frame) \Rightarrow ior`
- `emitREOP(self, op, inType, frame) \Rightarrow code for >, <, >=, <=, !=, ==`
- `emitRELOP(self, op, inType, trueLabel, falseLabel, frame) \Rightarrow code for condition in if statement`

Read/Write Variables APIs

- `emitREADVAR(self, name, inType, index, frame)` \Rightarrow `[ifa]load`
- `emitALOAD(self, inType, frame)` \Rightarrow `[ifa]aload`
- `emitWRITEVAR(self, name, inType, index, frame)` \Rightarrow `[ifa]store`
- `emitASTORE(self, inType, frame)` \Rightarrow `[ifa]astore`
- `emitGETSTATIC(self, lexeme, inType, frame)` \Rightarrow `getstatic`
- `emitGETFIELD(self, lexeme, inType, frame)` \Rightarrow `getfield`
- `emitPUTSTATIC(self, lexeme, inType, frame)` \Rightarrow `putstatic`
- `emitPUTFIELD(self, lexeme, inType, frame)` \Rightarrow `putfield`

Other APIs

- emitPUSHCONST(self, input, frame) \Rightarrow iconst, bipush, sipush, ldc
- emitPUSHFCONST(self, input, frame) \Rightarrow fconst, ldc
- emitINVOKESTATIC(self, lexeme, inType, frame) \Rightarrow invokestatic
- emitINVOKESPECIAL(self, frame, lexeme=None, inType=None) \Rightarrow invokespecial
- emitINVOKEVIRTUAL(self, lexeme, inType, frame) \Rightarrow invokevirtual
- emitIFTRUE(self, label, frame) \Rightarrow ifgt
- emitIFFALSE(self, label, frame) \Rightarrow ifle
- emitDUP(self, frame) \Rightarrow dup
- emitPOP(self, frame) \Rightarrow pop
- emitI2F(self, frame) \Rightarrow i2f
- emitRETURN(self, inType, frame) \Rightarrow return, ireturn
- emitLABEL(self, label, frame) \Rightarrow Label
- emitGOTO(self, label, frame) \Rightarrow goto

Tools are used to manage information used to generate code for a method

- Labels: are valid in the body of a method
 - ◁ `getNewLabel()`: return a new label
 - ◁ `getStartLabel()`: return the beginning label of a scope
 - ◁ `getEndLabel()`: return the end label of a scope
 - ◁ `getContinueLabel()`: return the label where a continue should come
 - ◁ `getBreakLabel()`: return the label where a break should come
 - ◁ `enterScope()`
 - ◁ `exitScope()`
 - ◁ `enterLoop()`
 - ◁ `exitLoop()`

Frame (cont'd)

- Local variable array
 - ◁ `getNewIndex()`: return a new index for a variable
 - ◁ `getMaxIndex()`: return the size of the local variable array
- Operand stack
 - ◁ `push()`: simulating a push execution
 - ◁ `pop()`: simulating a pop execution
 - ◁ `getMaxOpStackSize()`: return the max size of the operand stack
- Implemented in class `Frame`

Machine-Independent Code Generation

- Based on the source language
- Use facilities of Frame and Intermediate Code Generation (Emitter)

BKIT-Java mapping

- A source program \Rightarrow Java class
- A global variable \Rightarrow a static field
- A function \Rightarrow a static method
- A parameter \Rightarrow a parameter
- A local variable \Rightarrow a local variable
- An expression \Rightarrow an expression
- A statement \Rightarrow a statement
- An invocation \Rightarrow an invocation

Summary

- Use BCEL to know which code should be generated
- Generate code for expressions first
- Generate code for statements later
- Good luck