Chapter 6: Code Generation

Aarne Ranta

Slides for the book ”Implementing Programming Languages. An Introduction to Compilers and Interpreters”, College Publications, 2012.

Bridging the **semantic gap** between high-level and machine languages.

Yet another use of syntax-directed translation.

In detail: from imperative language to JVM.

In summary: compiling to native Intel x86 code.

All the concepts and tools needed for solving Assignment 4, which is a compiler from a fragment of C++ to JVM.

The semantic gap

high-level code	**machine code**
statement	instruction
expression	instruction
variable	memory address
value	bit vector
type	memory layout
control structure	jump
function	subroutine
tree structure	linear structure

General picture: machine code is simpler

The correspondence of concepts is *many-one*: both statements and expressions are compiled to instructions.

The compilation of constructs is *one-many*: one statement or expression translates to many instructions.

```
x + 3         iload 0
              bipush 3
              iadd
```

Code generation *ignores* some information in the source language.

But it also needs some information that is not explicit in the source code, e.g. type information for `+`.

Inference rules for code generation

Judgement form

$$\gamma \vdash e \downarrow c$$

Read: *expression e generates code c in environment* γ.

Example:

$$\frac{\gamma \vdash a \downarrow c \qquad \gamma \vdash b \downarrow d}{\gamma \vdash [a * b : \mathtt{int}] \downarrow c\, d\, \mathtt{imul}} \qquad \frac{\gamma \vdash a \downarrow c \qquad \gamma \vdash b \downarrow d}{\gamma \vdash [a * b : \mathtt{double}] \downarrow c\, d\, \mathtt{dmul}}$$

Notice: type annotations are assumed to be in place.

Pseudocode for code generation

More traditional than inference rules.

As the generated code can be long, the rules could become too wide to fit on the page...

Here is the pseudocode for the `*` rule above:

```
*compile*$(\gamma, [a * b : t])$ :
  $c$ := *compile*$(\gamma, a)$
  $d$ := *compile*$(\gamma, b)$
  **if** $t$ = *int*
    **return** $c\ d$ imul
  **else**
    **return** $c\ d$ dmul
```

In the pseudocode above, we assumed

```
$\mathit{Code}\ \mathit{compile}(\mathit{Env}\ \gamma, \mathit{Exp}\ e)$
```

But it is more common (and convenient) to use

```
$\mathit{Void}\ \mathit{compile}(\mathit{Exp}\ e)$

$\mathit{compile}([a * b : t])$ :
 $\mathit{compile}(a)$
 $\mathit{compile}(b)$
 if $t = \mathit{int}$
  $\mathit{emit}$(imul)
 else
  $\mathit{emit}$(dmul)
```

This format involves two simplifications:

- the environment is kept implicit - as a global variable,
- code is generated as a side effect - by the function *Void emit*(*Code* c), which writes the code into a file.

The compilation environment

The environment stores information on functions and variables.

In addition, we need to generate fresh labels for jump instructions.

Thus the environment contains

- for each function, its type in the JVM notation;
- for each variable, its address as an integer;
- a counter for variable addresses;
- a counter for jump labels

All labels in the code for a function must be distinct, because they must uniquely identify a code position.

Signatures of compilation helper functions

Void	*compile*	(*Exp* e)
Void	*compile*	(*Stm* s)
Void	*compile*	(*Def* d)
Void	*emit*	(*Code* c)
Address	*lookup*	(*Ident* x)
FunType	*lookup*	(*Ident* f)
Void	*extend*	(*Ident* x, *Type* t)
Void	*extend*	(*Def* d)
Void	*newBlock*	()
Void	*exitBlock*	()
Void	*emptyEnv*	()
Label	*newLabel*	()

Addresses and sizes

The addresses start from 0.

Each new variable gets a new address.

The next address is incremented by the **size** of the variable.

For integers, booleans, and strings, the size is 1. For doubles, it is 2.

The first variables are the function parameters, after which the locals follow.

Blocks can overshadow old variables as usual.

Example of addressess

M is the counter giving the next available address

```
int foo (double x, int y)
{               // x -> 0, y -> 2                                M=3
  string i ;    // x -> 0, y -> 2, i -> 3,                       M=4
  bool b ;      // x -> 0, y -> 2, i -> 3, b -> 4                M=5
  {             // x -> 0, y -> 2, i -> 3, b -> 4 .              M=5
    double i ;  // x -> 0, y -> 2, i -> 3, b -> 4 . i -> 5  M=7
  }             // x -> 0, y -> 2, i -> 3, b -> 4                M=5
  int z ;       // x -> 0, y -> 2, i -> 3, b -> 4, z -> 5   M=6
}
```

The maximum value of M is the maximum amount of variable storage needed by the program, here 7. This information is needed in JVM code generation.

Compiling integer and double literals

We can use the instructions

- `ldc` *i*, for pushing an integer or a string *i*
- `ldc2_w` *d*, for pushing a double *d*

These instructions use a separate storage called the **runtime constant pool**. This makes them less efficient than the following special instructions:

- `bipush` *b*, for integers whose size is one byte
- `iconst_m1` for -1, `iconst_0` for 0, . . . , `iconst_5` for 5
- `dconst_0` for 0.0, `dconst_1` for 1.0

The `dconst` and `iconst` sets are better than `bipush` because they need no second byte for the argument.

It is easy to optimize the code generation to one of these. But let us assume, for simplicity, the use of the worst-case instructions:

```
compile($i$) : // integer literals
  emit(ldc $i$)

compile($d$) : // double literals
  emit(ldc2_w $d$)

compile($s$) : // string literals
  emit(ldc $s$)
```

Arithmetic operations

Subtraction and division: see multiplication above.

Addition: add string concatenation, compiled as a function call (`invokestatic`)

```
compile([$a + b : t$]) :
  compile($a$)
  compile($b$)
  if $t$ = int
    emit(iadd)
  elseif $t$ = double
    emit(dadd)
  else
    emit(invokestatic runtime/plusString(Ljava/lang/String;
              Ljava/lang/String;)Ljava/lang/String;)
```

Variable expressions

Dependent on type:

$$\mathit{compile}([x : \mathit{int}]) : \quad \mathit{emit}(\texttt{iload}\ \mathit{lookup}(x))$$

$$\mathit{compile}([x : \mathit{double}]) : \quad \mathit{emit}(\texttt{dload}\ \mathit{lookup}(x))$$

$$\mathit{compile}([x : \mathit{string}]) : \quad \mathit{emit}(\texttt{aload}\ \mathit{lookup}(x))$$

Like for constants, there are special instructions available for small addresses.

Assignments

Some care needed, since they both have side effects and return values.

Simple-minded compilation:

```
i = 3 ⟹ iconst_3 istore_1
```

But the semantics (Chapter 5) says that, after `istore`, the value 3 is no more on the stack.

If the value is needed, we need both to store it and then load it back on the stack:

```
iconst_3 istore_1 iload_1
```

Another way is to **duplicate** the top of the stack first, with the instruction `dup`:

$$\langle \texttt{dup}, P, V, S.v\rangle \longrightarrow \langle P+1, V, S.v.v\rangle$$

This works for integers and strings; the variant for doubles is `dup2`.

The compilation scheme for assignments

```
*compile*([$x = e : t$]) :
 *compile*($e$)
 **if** $t =$ *int*
  *emit*(dup)
  *emit*(istore *lookup*($x$))
 **else if** $t =$ *double*
  *emit*(dup2)
  *emit*(dstore *lookup*($x$))
 **else**
  *emit*(dup)
  *emit*(astore *lookup*($x$))
```

The pop instruction

What about if the value is *not* needed? For instance, if the assignment is just used as a statement.

Then we can use the `pop` instruction,

$$\langle \texttt{pop}, P, V, S.v \rangle \longrightarrow \langle P + 1, V, S \rangle$$

and its big sister `pop2`.

The compilation scheme for expression statements

```
$\mathit{compile}([e : t];)$ :
  $\mathit{compile}(e)$
  **if** $t \in \{\mathit{int}, \mathit{bool}, \mathit{string}\}$
    $\mathit{emit}(\texttt{pop})$
  **else if** $t = \mathit{double}$
    $\mathit{emit}(\texttt{pop2})$
  **else** $\mathit{return}$
```

The last case takes case of expressions of type `void` (typically function calls).

Declarations

Emit no code, but reserve a place in the variable storage:

```
*compile*($t\ x;$) :
  *extend*($x, t$)
```

The `extend` helper function looks up the smallest available address for a variable, say *i*, and updates the compilation environment with the entry $(x \rightarrow i)$. It also increments the ''smallest available address'' by the size of the type.

Blocks

Create a new part of storage, and free it at exit from the block:

```
compile($\{s_1 \ldots s_n\}$) :
  newBlock()
  for $i = 1, \ldots, n$ : compile($s_i$)
  exitBlock()
```

Using jumps: while statements

```
$compile(\texttt{while}(exp)stm)$ :
  $TEST := newLabel()$
  $END := newLabel()$
  $emit(TEST{:})$
  $compile(exp)$
  $emit(\texttt{ifeq}\ END)$
  $compile(stm)$
  $emit(\texttt{goto}\ TEST)$
  $emit(END{:})$
```

```
                           TEST:
while (exp)                  exp
  stm                        ifeq END
                             stm
                             goto TEST
                           END:
```

As specified in semantics, `ifeq` checks if the top of the stack is 0. If yes, the execution jumps to the label; if not, it continues to the next instruction.

The checked value is the value of `exp` in the `while` condition.

- if 0, false: the body is not executed
- otherwise, true, and the body `stm` is executed, and we jump back to the test

Compiling if statements

With the minimum of labels:

```
  if (exp)                exp
    stm1                  ifeq FALSE
  else                    stm1
    stm2                  goto TRUE
                        FALSE:
                          stm2
                        TRUE:
```

Idea: have a label for the false case, similar to the label `END` in `while` statements.

But we also need a label for true, to prevent the execution of the `else` branch. The compilation scheme is straightforward to extract from this

Comparisons

JVM has no comparison operations returning boolean values.

Therefore, if we want the value of `exp1` $<$ `exp2`, we execute code corresponding to

```
  if (exp1 < exp2) 1 ; else 0 ;
```

We use he conditional jump `if_icmplt` *LABEL*, which compares the two elements on the top of the stack and jumps if the second-last is less than the last:

$$\langle \texttt{if\_icmplt}\ L, P, V, S.v.w\rangle \longrightarrow \langle P(L), V, S\rangle\ (v < w)$$

$$\langle \texttt{if\_icmplt}\ L, P, V, S.v.w\rangle \longrightarrow \langle P + 1, V, S\rangle\ (v \geq w)$$

To do this with just one label: first push 1 on the stack. Overwritten by 0 if the comparison does not succeed:

```
    bipush 1
    exp1
    exp2
    if_icmplt TRUE
    pop
    bipush 0
  TRUE:
```

There are instructions similar to `if_icmplt` for all comparisons of integers: `eq`, `ne`, `lt`, `gt`, `ge`, and `le`.

Comparing doubles

The mechanism is different. There is just one instruction, `dcmpg`:

$$\langle \texttt{dcmpg}, P, V, S.d.e \rangle \longrightarrow \langle P + 1, V, S.v \rangle$$

where $v = 1$ if $d > e, v = 0$ if $d = e,$ and $v = -1$ if $d < e$.

Spaghetti code

```
while (x < 9) stm
```

```
TEST:
  bipush 1
  iload 0
  bipush 9
  if_icmplt TRUE
  pop
  bipush 0
TRUE:
  ifeq goto END
  stm
  goto TEST
END:
```

Putting together the compilation of comparisons and `while` loops gives awful code with lots of jumps.

Less spaghetti

```
                             TEST:                TEST:
  while (x < 9) stm            bipush 1
                               iload 0              iload 0
                               bipush 9             bipush 9
                               if_icmplt TRUE       if_icmpge END
                               pop
                               bipush 0
                             TRUE:
                               ifeq goto END
                               stm                  stm
                               goto TEST            goto TEST
                             END:                 END:
```

The right column makes the comparison directly in the `while` jump, by using its *negation* `if_icmpge`; recall that `!(a < b) == (a >= b)`.

Problem: how to get this code by using the compilation schemes?

Compositionality

A syntax-directed translation function T is **compositional**, if the value returned for a tree is a function of the values for its immediate subtrees:

$$T(Ct_1 \ldots t_n) = f(T(t_1), \ldots, T(t_n))$$

In the implementation, this means that,

- in Haskell, pattern matching does not need patterns deeper than one;
- in Java, one visitor definition per class and function is enough.

Non-compositional compilation

Avoiding the `while` + comparison spaghetti code would need **non-compositional** compilation schemes.

Simple in Haskell: use deeper patterns,

```
  compile (SWhile (ELt exp1 exp2) stm) = ...
```

In Java, another visitor must be written to define what can happen depending on the condition part of `while`.

Another approach is to use compositional code generation followed by a separate phase of **back-end optimization** of the generated code. This technique is more modular and therefore usually preferable to non-compositional code generation.

Function calls

Function calls in JVM are a generalization of arithmetic operations:

1. Push the function arguments on the stack.
2. Evaluate the function (with the arguments on the top of the stack as parameters).
3. Return the value on the stack, popping the arguments.

In a function call $f(a, b, c)$, the stack evolves as follows:

S	before the call
S.a.b.c	entering *f*
S.	executing *f*, with *a*,*b*,*c* in variable storage
S.v	returning from *f*

Entering a function `f` means a jump to the code for `f`, with the arguments as the first available variables.

The evaluation doesn’t have access to old variables or to the stack of the calling code, but these become available again when the function returns.

Compilation scheme for function calls

```
$\mathit{compile}(f(a_1, \ldots, a_n))$ :
  for $i = 1, \ldots, n$ : $\mathit{compile}(a_i)$
  $\mathit{typ} := \mathit{lookup}(f)$
  $\mathit{emit}(\texttt{invokestatic}\ C/f\ \mathit{typ})$
```

The JVM instruction for function calls is `invokestatic`.

It works for Java's `static` methods only.

The instruction needs to know the type of the function, and its class.

We assume for simplicity that there is just one class `C`.

Notation for function calls

Example:

```
  invokestatic C/mean(II)I
```

This calls a function `int mean (int x, int y)` in class `C`.

So the type is written with a special syntax. Simple types have one-letter symbols:

```
  I = int, D = double, V = void, Z = boolean
```

There is no difference between integers and booleans in execution, but the JVM interpreter may use the distinction for **.bytecode verification**, that is, type checking at run time.

Complex types (corresponding to classes) have very special encodings:

```
  Ljava/lang/String; = string
```

Function definitions

Example function and its compilation as a **method**:

```
  int mean (int x, int y)         .method public static mean(II)I
  {                               .limit locals 2
                                  .limit stack 2
                                     iload_0
                                     iload_1
                                     iadd
                                     iconst_2
                                     idiv
    return ((x+y) / 2) ;             ireturn
  }                                .end method
```

Before the body, two limits are specified: the storage needed for local variables (V in the semantic rules) and the storage needed for the evaluation stack (S in the semantics).

The only variables are the two arguments. Since they are integers, the limit is 2.

The stack can be calculated by simulating the JVM: it reaches 2 when pushing the two variables, but never beyond that.

The code generator can calculate these limits by maintaining them in the environment.

Compilation scheme for function definitions

We write *funtypeJVM*$(t_1, \ldots, t_m, t))$ to create the JVM representation for the type of the function.

```
$compile(t\ f(t_1\ x_1, \ldots, t_m\ x_m)\{s_1, \ldots, s_n\})$ :
  $emit($.method public static $f$ $funtypeJVM(t_1, \ldots, t_m, t))$
  $emit($.limit locals $locals(f))$
  $emit($.limit stack $stack(f))$
  for $i = 1, \ldots, m$ : $extend(x_i, t_i)$
  for $i = 1, \ldots, n$ : $compile(s_i)$
  $emit($.end method$)$
```

Return statements

```
*compile*(return [$e : t$]; ) :
  *compile*($e$)
  **if** $t$ = *string*
    *emit*(areturn)
  **else if** $t$ = *double*
    *emit*(dreturn)
  **else**
    *emit*(ireturn)
```

```
*compile*(return; ) :
  *emit*(return)
```

Putting together a class file

Class files can be built with the following template:

```
.class public Foo
.super java/lang/Object

.method public <init>()V
  aload_0
  invokenonvirtual java/lang/Object/&lt;init>()V
  return
.end method

; user's methods one by one
```

The methods are compiled as described in the previous section. Each method has its own stack, locals, and labels.

A jump from one method can never reach a label in another method.

The main method

If we follow the **C** convention as in Assignment 4, the class must have a `main` method. In JVM, its type signature is different from C:

```
  .method public static main([Ljava/lang/String;)V
```

following the **.Java** convention that `main` takes an array of strings as its argument and returns a `void`.

The code generator must therefore treat `main` as a special case: create this type signature and reserve address 0 for the array variable. The first available address for local variables is 1.

The class name, `Foo` in the above template, can be generated from the file name (without suffix).

IO functions and the runtime class

Predefined functions for reading and printing integers, doubles, and strings (as in the interpreter).

Put them into a separate class, `runtime`, and call as usual:

```
  invokestatic runtime/printInt(I)V
  invokestatic runtime/readInt()I
```

Also a function for string concatenation (+) can be included in this class.

To produce the this class, write a Java program `runtime.java` and compile it to `runtime.class`.

You will of course be able to run ”standard” Java code together with code generated by your own compiler!

Assembling the code

We have been showing assembly code in the format called **Jasmin**.

Thus we first generate a Jasmin file `Foo.j`.

Then we create the class file `Foo.class` by calling Jasmin:

```
  jasmin Foo.j
```

To run the program, we write as usual,

```
  java Foo
```

This executes the `main` function.

To get Jasmin: http://jasmin.sourceforge.net/

Disassembly

You can disassemble a Java class file with

```
  javap -c Foo
```

The notation is slightly different from Jasmin, but this is still a good way to compare your own compiler with the standard `javac` compiler.

The main differences are that jumps use line numbers instead of labels, and that the `ldc` and `invokestatic` instructions refer to the **runtime constant pool** instead of showing explicit arguments.

Implementing code generation

Our compilation schemes leave the environment implicit, and code generation is performed by side effects.

This is simple in Java, as the environment can be managed by side effects, and recursive calls to the compiler in the visitor need not pass around the environment explicitly.

In Haskell, the standard functional style would require each call to return a new environment. This can be avoided by using a **state monad**.

Code generation in Haskell: the state monad

In the standard library `Control.Monad.State`, the type `State s v` is a monad that maintains a state of type `s` and returns a value of type `v`.

Internally, it is a function of type

```
  s -> (s,v)
```

which takes a state as its input and returns a value and a new state.

The state can be inspected and modified by the library functions

```
  get    :: State s s
  modify :: (s -> s) -> State s ()
```

Compilation in the state monad

We use (), which is Haskell's version of*Void*:

```
compileStm :: Stm -> State Env ()
compileExp :: Stm -> State Env ()
```

Example: multiplication expressions

```
EMul a b -> do
  compileExp a
  compileExp b
  emit $ case typExp e of
    Type_int -> imul_Instr
    Type_double -> dmul_Instr
```

The helper function `typExp` is easy to define if the type checker has type-annotated all trees.

The environment

Symbol tables for functions and variables

Counters for variable addresses and labels

Counter for the maximum stack depth

The code that has been generated

Here is a partial definition:

```
data Env = Env {
  vars      :: [Map Ident Int],
  maxvar    :: Int,
  code      :: [Instruction]
  }
```

Emitting code

Change the `code` part of the environment by using the state monad library function `modify`

```
emit :: Instruction -> State Env ()
emit i c = modify (\s -> s{code = i : code s})
```

Notice that the instructions are collected in reverse order, which is more efficient.

Looking up variable addresses

Inspect the `vars` part of the environment by using the `get` library function:

```
lookupVar :: Ident -> State Env Int
lookupVar x = do
  s <- get
  -- then look up the first occurrence of x in (vars s)
```

Notice that a stack (i.e. list) of variable environments is needed, to take care of block structure.

All the operations needed can be implemented with `get` and `modify`, so that the imperative flavour and simplicity of the compilation schemes is preserved.

Code generation in Java

Use a `CodeGenerator` class similar to the `TypeChecker` class in Chapter 4 and the `Interpreter` class in Chapter 5.

A visitor takes care of syntax-directed translation.

But we now declare an environment as a class variable in `CodeGenerator`, so that we can access it by side effects.

Thus we don't pass around an `Env` argument, but just a dummy `Object`.

(We could have done so in the type checker and the interpreter as well - but we wanted to keep the Java code as close to the abstract specification as possible.)

Code generation: either print the instructions, or collect them in the environment.

A code generator class, with a simplified environment

```
public class CodeGenerator {
  public void compile(Program p)
  private static class Env {
    private LinkedList<HashMap<String,Integer>> vars;
    private Integer maxvar;
    public Env() {
      vars = new LinkedList<HashMap<String,Integer>>();
      maxvar = 0 ;
    }
    public void addVar(String x, TypeCode t) {
    // use TypeCode to determine the increment of maxvar
    }
  }
  private Env env = new Env() ;
```

Compilation of statements

```
private void compileStm(Stm st, Object arg) {
  st.accept(new StmCompiler(), arg);
}

private class StmCompiler implements Stm.Visitor<Object,Object> {
  public Object visit(Mini.Absyn.SDecl p, Object arg) {
    env.addVar(p.ident_, typeCode(p.type));
    return null;
  }
}
```

Compilation of expressions

```
  private Object compileExp(Exp e, Object arg) {
    return e.accept(new ExpCompiler(), arg);
  }

  private class ExpCompiler implements Exp.Visitor<Object,Object> {
    public Object visit(Mini.Absyn.EMul p, Object arg) {
      p.exp_1.accept(this, arg) ;
      p.exp_2.accept(this, arg) ;
      if (typeCodeExp(p.exp_1) == TypeCode.INT) {
        System.err.println("imul");
        } else {
        System.err.println("dmul");
        }
      return null ;
    }
  }
}
```

The type code of an expression is assumed to be put in place by an annotating type checker.

Compiling to native code*

Native code: machine language of a real processor

Example: **Intel x86** series (the 8086 processor from 1978; later 8088 (the **IBM PC**), 80286, 80386, Pentium), used in Linux, Mac OS, and Windows alike.

We will use the assembly language **NASM** (**Netwide Assembler**) for Intel x86.

Registers

The main difference between x86 and JVM is that x86 has **registers**: places for data in the processor itself (as opposed to the memory).

Registers be accessed immediately by for instance arithmetic operations:

```
  add eax, ebx
```

is an instruction to add the value of the register `ebx` to the value in `eax`. It corresponds to an assignment

```
  eax := eax + ebx
```

The stack

x86 also has a notion of a **stack**: a part of the memory that can be accessed by its address

Memory address = register + **offset**:

```
  add eax, [ebp-8]
```

means an addition to `eax` of the value stored in the address `[ebp-8]`, where `ebp` is the register pointing to the beginning of the current **stack frame**, that is, the memory segment available for the current function call.

Example: Fibonacci numbers less than 500

```
int fib ()           fib:
{                      enter 0,0     ; prepare stack frame
  int hi,lo ;          ; reserve eax for hi, ebx for lo
  hi = 1 ;             mov eax, 1    ; eax := 1
  lo = hi ;            mov ebx, eax  ; ebx := eax
  printInt(hi) ;       call printInt ; print eax
  while (hi < 500) { test:
                       cmp eax, 500  ; set values to compare
                       jge end       ; jump if eax >= 500
    printInt(hi) ;     call printInt
    hi = hi + lo ;     add eax, ebx  ; eax := eax + ebx
    lo = hi - lo ;     mov ecx, eax  ; ecx := eax temporarily
                       sub ecx, ebx  ; ecx := ecx - ebx
                       mov ebx, ecx  ; ebx := ecx
                       jmp test      ; jump back to test
    }                end:
  return 0 ;           mov eax, 0    ; return value
                       leave         ; leave stack frame
}                      ret           ; return to caller
```

Comparisons to JVM code

The `while` loop is similar: labels and jumps

Arithmetic operations need fewer instructions than in JVM, if they have the form

> *dest* := *dest op src*.

Notice, however, that we use three instructions and a temporary register `ecx` for the subtraction `lo = hi - lo`, because it is not in this form.

Register allocation

x86 has only a few registers available for integers.

Any number of them may be needed as temporaries when evaluating expressions.

Therefore a compiler cannot usually reserve registers for variables, but they are stored in memory, and the code becomes less efficient than the idealized hand-written code shown above.

Register allocation is an optimization that tries to maximize the use of registers (as opposed to the stack).

Compiling functions

A major difference from JVM

- JVM has an inherent structure of classes and methods corresponding to the structure of Java programs
- x86 has no such structure - just code loosely grouped by labels

This means more freedom - and more challenge - for the assembly programmer and the compiler writer.

As a compiler must work for all possible input in a systematic way, it must follow some discipline of **calling conventions**: the way in which arguments are passed and values are returned.

JVM has fixed calling conventions: the arguments are pushed on the stack when entering a function, and popped when the function returns and leaves just the value on the stack.

In x86, this is just one possible convention: one can also use registers.

For instance, the Fibonacci code above uses `eax` for the return value of `fib`, but also for passing the argument to `printInt`.

We also stored the variable `hi` in `eax`, because `hi` was the argument always passed to `printInt`. In this way we saved many copying (`mov`) instructions. But in general, a compiler cannot assume such lucky coincidences.

Normal C calling conventions

Like in JVM, a stack is used for local variables and the temporary values of evaluation.

Unlike in JVM, some values may be held in registers instead of the stack.

Each function call has its own part of the stack, known as its **stack frame**.

The address of the beginning of the stack frame is stored in the **frame pointer**, for which the register `ebp` is used.

Another register, `esp`, is used as the **stack pointer**, which points to the current top of the stack.

As the called function cannot know anything about where it was called from,

- registers must be saved on the stack before calling
- the return address (code pointer) must be saved

Calling function ’new’ from function ’old’

1. Push values of the registers on the stack (to save them).
2. Push the arguments in reverse order.
3. Push the **return address** (i.e. code pointer back to *old*).
4. Push the frame pointer value of *old* (to save it).
5. Use the old stack pointer as frame pointer of *new*.
6. Reserve stack space for the local variables of *new*.
7. Execute the code of *new*.
8. Return the value of *new* in register `eax`.
9. Restore the *old* frame pointer value.
10. Jump to the saved return address.
11. Restore the values of saved registers.

Example: the stack when `old` has called `new`, where

```
old(x,y) { ... new(e) ... }
new(x)   { ... }
```

⋮	local variables of the caller of `old`
⋮	saved registers of the caller of old
2	second argument to `old`
1	first argument to `old`
ret	saved return address for the call of `old`
fp	saved frame pointer of the caller of `old`
____	← frame pointer of `old`
⋮	local variables of `old`
⋮	saved registers of `old`
3	argument to `new`
ret	saved return address for the call of `new`
fp	saved frame pointer of `old`
____	← frame pointer of `new`
⋮	local variables of `new`
	← stack pointer

Convention: the stack grows downwards; offsets are positive before the frame pointer, negative after it.

For instance, inside `old` before calling `new`,

- the saved frame pointer is in `[ebp+4]`,
- the return address is in `[ebp+8]`,
- the first argument is `[ebp+12]`,
- the second argument is `[ebp+16]`;
- the first local variable is `[ebp]`,
- the second local variable is `[ebp-4]`.
- the third local variable is `[ebp-8]`.

assuming the arguments and variables are integers and consume 4 bytes each.

CISC and RISC

Complex instructions, ”doing many things”, for C-style calling conventions

- `enter` for entering a stack frame,
- `leave` for leaving a stack frame,
- `pusha` for pushing certain registers to the stack to save them, and
- `popa` for popping the values back to the same registers.

They make the assembly code shorter, but may consume more processor time than simpler instructions.

x86 has **CISC** architecture = **Complex Instruction Set Computing**

Alternative: **RISC = Reduced Instruction Set Computing**, e.g. in the **ARM processor** used in mobile phones.

LLVM

Derived from ”Low Level Virtual Machine”

Intermediate code, with back ends to many machines

RISC-style instructions with an infinite supply of **virtual registers**

Compilers can just generate LLVM and leave the generation of native code to LLVM tools.

Thus they can share front ends and intermediate code optimizations

Code optimization*

An important part of modern compilers

A magic trick that turns badly written source code into efficient machine code?

Sometimes, yes - but usually not: the main purpose is to tidy up the mess that earlier compilation phases have created!

In this way, each phase can be kept timple. Examples:

- to keep instruction selection compositional
- to allow an infinite supply of registers

Constant folding

This *is* a way to improve source code

To encourage more structure, without fear of inefficiency.

Example:

```
  int age = 26 * 365 + 7 * 366 + 31 + 28 + 4 ;
```

can be optimized to

```
  int age = 12115 ;
```

Principle of constant folding: operations on constants are carried out at compile time.

Partial evaluation

A generalization of constant folding: evaluate ”as much as you can” at compile time.

What you can’t evaluate are run-time variables.

This can be tricky. Could we for instance apply the general rule

$$e - e \Longrightarrow 0$$

at compile time, for any expression *e*?

No, because e can have side effects - for instance,

```
  i++ - i++
```

or

```
  f() - f()
```

which prints `hello` at each call.

Pure languages

Expression evaluation doesn't have side effects.

Example: Haskell

Can be optimized more aggressively than non-pure languages - very much exploited by GHC (Glasgow Haskell Compiler).

Even in pure languages, care is needed - for instance, the expression

```
1/x - 1/x
```

cannot be optimized to 0, because `x` might be 0 and raise a run-time exception.

Tail recursion elimination

Yet another way to encourage high-level programming: recursive functions.

A function is tail recursive if the last thing it does is call itself:

$$f(\ldots)\{\ldots f(\ldots);\}$$

Unoptimized compilation of functions creates new stack frames for each call.

In a tail recursive function, this is unnecessary: instead of building a new stack frame, the function can just re-run and update the variables of the old stack frame in place.

Peephole optimization

Applied to generated machine code.

Goes through the code and looks for segments that could be improved.

The segments typically have a limited size - e.g. 3 instructions.

Example: constant folding for JVM:

```
bipush 5
bipush 6       bipush 5
bipush 7   ⟹   bipush 42   ⟹   bipush 47
imul           iadd
iadd
```

Another example: elimination of stack duplication for an assignment whose value is not needed

```
              bipush 6
i = 6 ;  ⟶   dup          ⟹  bipush 6
              istore 4        istore 4
              pop
```

Register allocation

Tries to fit all variables and temporary values to a given set of registers.

For instance, to the small set of registers in x86.

Usually performed on an intermediate language such as LLVM, after first letting the code generator use an unlimited supply of registers, known as **virtual registers**.

Main technique: **liveness analysis**: variables that are "live" at the same time cannot be kept in the same register.

Live = may be needed later in the program.

Liveness example

```
int x = 1 ;         // x     live
int y = 2 ;         // x y   live
printInt(y) ;       // x     live
int z = 3 ;         // x   z live
printInt(x + z) ; // x   z live
y = z - x ;         //   y   live
z = f(y) ;          //   y   live
printInt(y) ;
return ;
```

How many registers are needed for the three variables `x`, `y`, and `z`?

Answer: two, because `y` and `z` are never live at the same time and can hence be kept in the same register.

Register allocation in two steps

1. **Liveness analysis**: find out which variables are live at the same time, to build an **interference graph**, where the nodes are variables and edges express the relation ”live at the same time”.
2. **Graph colouring**: colour the nodes of the graph so that the same colour can be used for two nodes if they are not connected.

The example program interference graph:

```
 x

/ \

y z
```

This shows that `y` and `z` can be given the same colour.

Dataflow analysis

The family of techniques related to liveness analysis.

Another example: **dead-code elimination**: a piece of code is dead if it can never be reached when running a program.

Assignment to a variable that is not live is dead code, as `z = f(y)` in the code above

But note: `f(y)` may have to be executed for side effects!

Optimization is undecidable

In a conditional statement

`if` (*condition*) *then_branch* `else` *else_branch*

the *else_branch* is dead code if *condition* is uniformly true.

But *condition* may express an undecidable proposition, even if it doesn't depend on run-time variables.