8. Code Generation

Prof. O. Nierstrasz

Thanks to Jens Palsberg and Tony Hosking for their kind permission to reuse and adapt the CS132 and CS502 lecture notes.
http://www.cs.ucla.edu/~palsberg/
http://www.cs.purdue.edu/homes/hosking/

Roadmap

- Runtime storage organization
- Procedure call conventions
- Instruction selection
- Register allocation
- Example: generating Java bytecode

See, *Modern compiler implementation in Java* (Second edition), chapters 6 & 9.

Roadmap

- **Runtime storage organization**
- Procedure call conventions
- Instruction selection
- Register allocation
- Example: generating Java bytecode

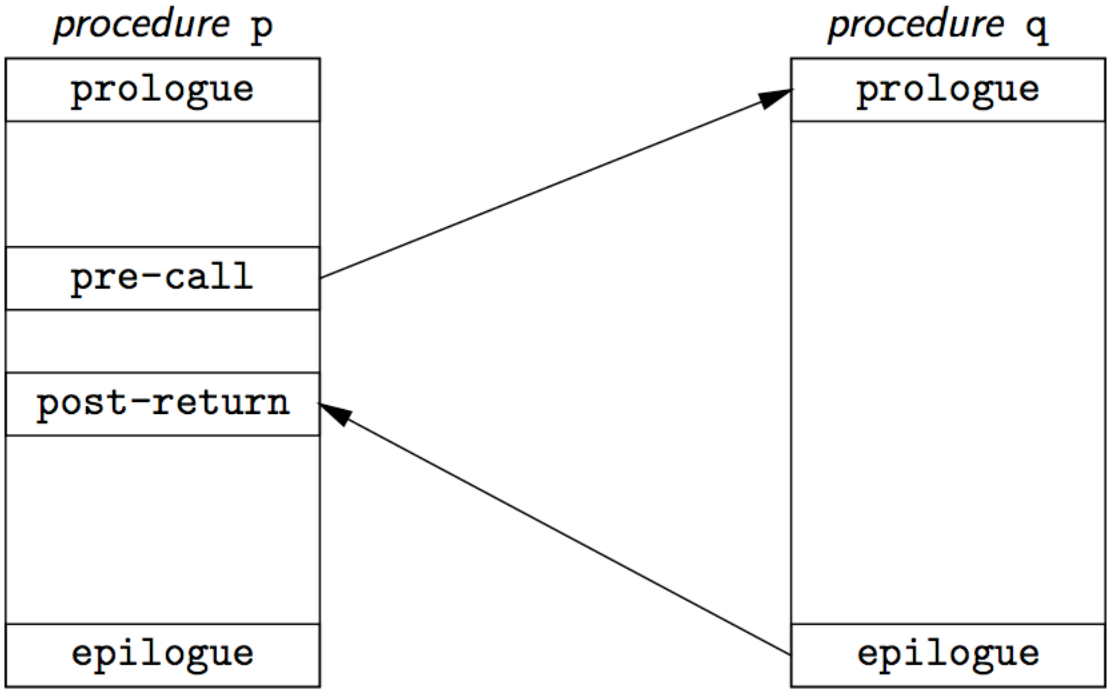
Runtime organization

- The *procedure abstraction* supports separate compilation
  - build large programs
  - keep compile times reasonable
  - independent procedures

- The linkage convention:
  - *a “social contract”* — procedures inherit a valid run-time environment *and* restore one for their parents
  - *machine dependent* — code generated at compile time
  - *distributes responsibility* — executes at run time

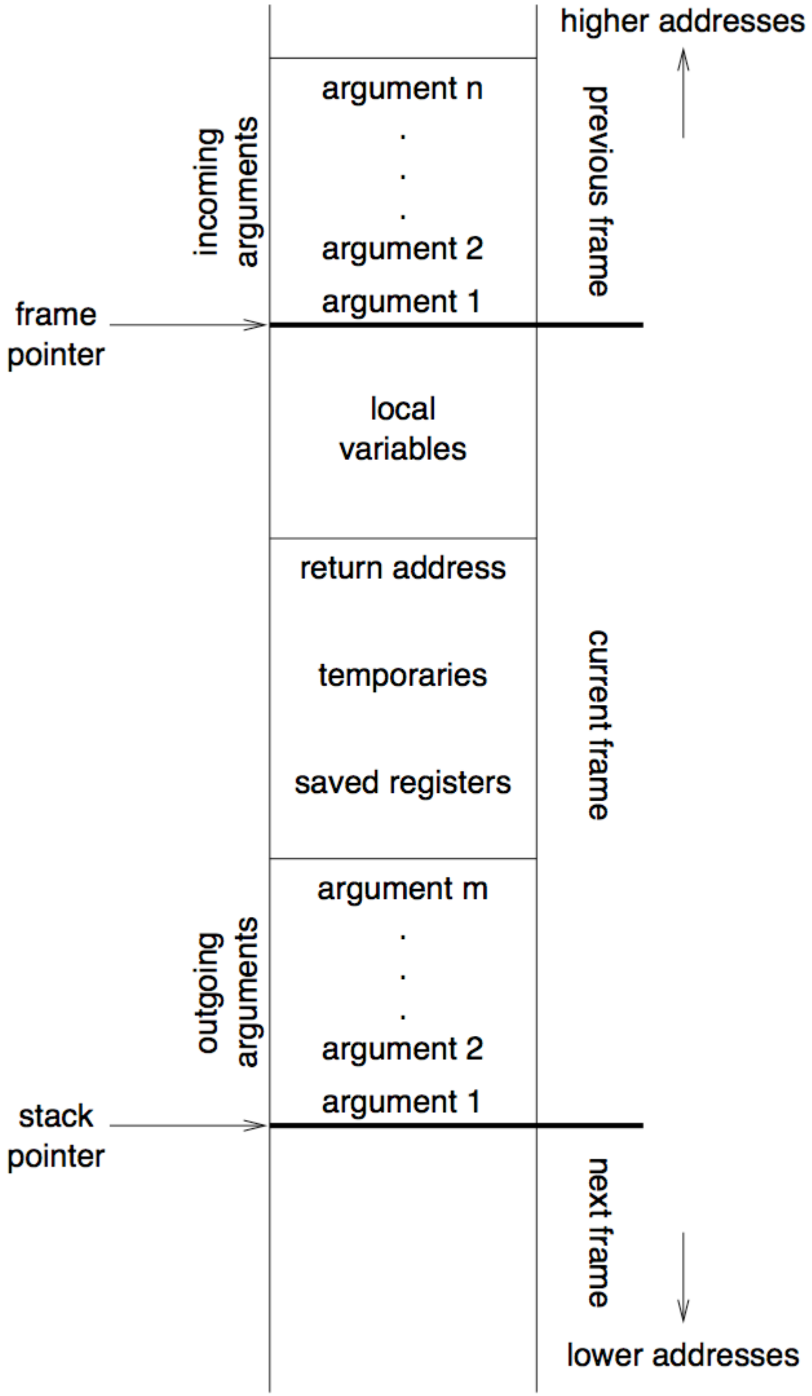
The procedure abstraction

- on entry, establish *p*'s environment
- when calling *q*, preserve *p*'s environment
- on exit, tear down *p*'s environment



Procedure linkages

Each procedure activation has an *activation record* or *stack frame*



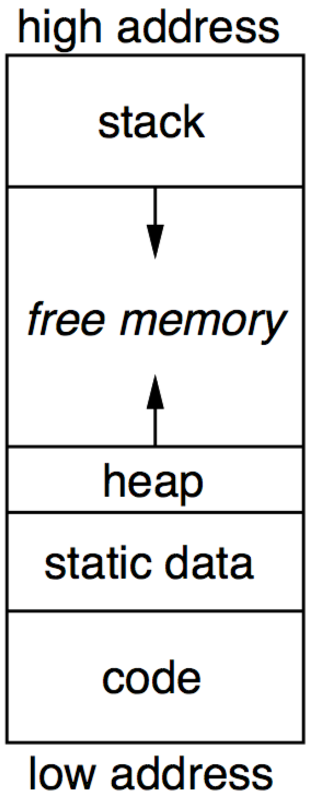
Procedure linkage contract

	Caller	***Callee***
Call	*pre-call* 1.allocate basic frame 2.evaluate & store parameters 3.store return address 4.jump to child	*prologue* 1.save registers, state 2.store FP (dynamic link) 3.set new FP 4.store static link to outer scope 5.extend basic frame for local data 6.initialize locals 7.fall through to code
Return	*post-call* 1.copy return value 2.de-allocate basic frame 3.restore parameters (if copy out)	*epilogue* 1.store return value 2.restore state 3.cut back to basic frame 4.restore parent’s FP 5.jump to return address

Typical run-time storage organization

Heap grows "up", stack grows "down".

- Allows both stack and heap maximal freedom.
- Code and static data may be separate or intermingled.



Variable scoping

Who sees local variables? Where can they be allocated?

Downward exposure

- called procedures see my variables?
- dynamic scoping vs. lexical scoping

Upward exposure

- can I return a reference to my variables?
- functions that return functions
- continuation-passing style

Only with downward exposure can the compiler allocate local variables in frames on the run-time stack.

Storage classes

- **Static variables**
  - addresses compiled into code
  - usually allocated at compile-time (fixed-size objects)
  - naming scheme to control access
- **Global variables**
  - similar to static variables
  - layout may be important
  - universal access
- **Procedure local variables**
  - allocated on stack …
  - if fixed size, limited lifetime, and values not preserved
- **Dynamically allocated variables**
  - call-by-reference implies non-local lifetime
  - usually explicit allocation
  - de-allocation explicit or implicit

Access to non-local data

- Map name to (*level*, *offset*) pair
  - reflects lexical scoping
  - look up name to find most recent declaration
  - If *level* = *current level* then variable is local,
  - else must generate code to look up stack
  - Must maintain *access links* to previous stack frame
  - Alternative: use *display* (table of access links)

http://en.wikipedia.org/wiki/Call_stack

Roadmap

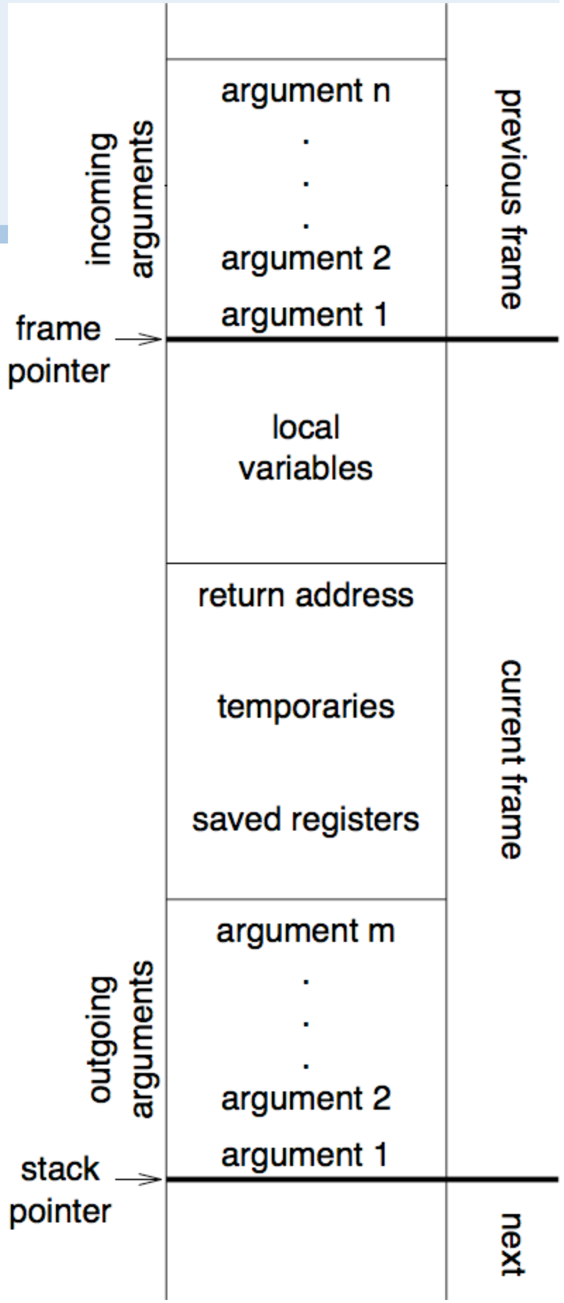
- Runtime storage organization
- **Procedure call conventions**
- Instruction selection
- Register allocation
- Example: generating Java bytecode

Calls: Saving and restoring registers

	callee saves	***caller saves***
caller's registers	Call includes bitmap of caller's registers to be saved/restored. *Best: saves fewer registers, compact call sequences*	Caller saves and restores own registers. Unstructured returns (e.g., exceptions) cause some problems to locate and execute restore code.
callee's registers	Backpatch code to save registers used in callee on entry, restore on exit. Non-local gotos/exceptions must unwind dynamic chain to restore callee-saved registers.	Bitmap in callee's stack frame is used by caller to save/ restore. Unwind dynamic chain as at left.
all registers	Easy. Non-local gotos/exceptions must restore all registers from "outermost callee"	Easy. (Use utility routine to keep calls compact.) Non-local gotos/exceptions need only restore original registers.

Call/return (callee saves)

1. caller pushes space for return value
2. caller pushes SP (stack pointer)
3. caller pushes space for: return address, static chain, saved registers
4. caller evaluates and pushes actuals onto stack
5. caller sets return address, callee's static chain, performs call
6. callee saves registers in register-save area
7. callee copies by-value arrays/records using addresses passed as actuals
8. callee allocates dynamic arrays as needed
9. on return, callee restores saved registers
10. callee jumps to return address



MIPS registers

Name	Number	Use	Callee must preserve?
$zero	$0	constant 0	N/A
$at	$1	assembler temporary	no
$v0–$v1	$2–$3	Values for function returns and expression evaluation	no
$a0–$a3	$4–$7	function arguments	no
$t0–$t7	$8–$15	temporaries	no
$s0–$s7	$16–$23	saved temporaries	yes
$t8–$t9	$24–$25	temporaries	no
$k0–$k1	$26–$27	reserved for OS kernel	no
$gp	$28	global pointer	yes
$sp	$29	stack pointer	yes
$fp	$30	frame pointer	yes
$ra	$31	return address	N/A

http://en.wikipedia.org/wiki/MIPS_architecture

MIPS procedure call convention

- ***Philosophy:***
  - Use full, general calling sequence only when necessary
  - Omit portions of it where possible
    (e.g., avoid using FP register whenever possible)

- ***Classify routines:***
  - *non-leaf routines* call other routines
  - *leaf routines* don’t
    - identify those that require stack storage for locals
    - and those that don’t

MIPS procedure call convention

- ***Pre-call:***
  1. Pass arguments: use registers a0 . . . a3; remaining arguments are pushed on the stack along with save space for a0 . . . a3
  2. Save caller-saved registers if necessary
  3. Execute a `jal` instruction:
     - jumps to target address (callee's first instruction), saves return address in register ra

MIPS procedure call convention

- ***Prologue:***
  1. Leaf procedures that use the stack and non-leaf procedures:
     a) Allocate all stack space needed by routine:
        - local variables
        - saved registers
        - arguments to routines called by this routine

        ```
        subu $sp, framesize
        ```
     b) Save registers (ra etc.), e.g.:

        ```
        sw $31, framesize+frameoffset($sp)
        sw $17, framesize+frameoffset-4($sp)
        sw $16, framesize+frameoffset-8($sp)
        ```

        where `framesize` and `frameoffset` (usually negative) are compile- time constants
  2. Emit code for routine

MIPS procedure call convention

> ***Epilogue:***

1. Copy return values into result registers (if not already there)
2. Restore saved registers

    ```
    lw $31, framesize+frameoffset-N($sp)
    ```
3. Get return address

    ```
    lw $31, framesize+frameoffset($sp)
    ```
4. Clean up stack

    ```
    addu $sp,framesize
    ```
5. Return

    ```
    j $31
    ```

Roadmap

- Runtime storage organization
- Procedure call conventions
- **Instruction selection**
- Register allocation
- Example: generating Java bytecode

Instruction selection

> ***Simple approach:***
  - Macro-expand each IR tuple/subtree to machine instructions
  - Expanding independently leads to poor code quality
  - Mapping may be many-to-one
  - “Maximal munch” works well with RISC

> ***Interpretive approach:***
  - Model target machine state as IR is expanded

Register and temporary allocation

- Limited # hard registers
  - assume *pseudo-register* for each temporary
  - register allocator chooses temporaries to spill
  - allocator generates mapping
  - allocator inserts code to spill/restore pseudo-registers to/from storage as needed

IR tree patterns

- A *tree pattern* characterizes a fragment of the IR corresponding to a machine instruction
  - Instruction selection means *tiling* the IR tree with a minimal set of tree patterns

MIPS tree patterns (example)

—	r_i			TEMP
—	r_0			CONST 0
li	Rd	I		CONST
la	Rd	label		NAME
move	Rd	Rs		MOVE(•, •)
add	Rd	Rs$_1$	Rs$_2$	+(•, •)
	Rd	Rs$_1$	I_{16}	+(•, CONST$_{16}$), +(CONST$_{16}$, •)
mulo	Rd	Rs$_1$	Rs$_2$	×(•, •)
	Rd	Rs	I_{16}	×(•, CONST$_{16}$), ×(CONST$_{16}$, •)
and	Rd	Rs$_1$	Rs$_2$	AND(•, •)
	Rd	Rs$_1$	I_{16}	AND(•, CONST$_{16}$), AND(CONST$_{16}$, •)
or	Rd	Rs$_1$	Rs$_2$	OR(•, •)
	Rd	Rs$_1$	I_{16}	OR(•, CONST$_{16}$), OR(CONST$_{16}$, •)
xor	Rd	Rs$_1$	Rs$_2$	XOR(•, •)
	Rd	Rs$_1$	I_{16}	XOR(•, CONST$_{16}$), XOR(CONST$_{16}$, •)
sub	Rd	Rs$_1$	Rs$_2$	−(•, •)
	Rd	Rs	I_{16}	−(•, CONST$_{16}$)
div	Rd	Rs$_1$	Rs$_2$	/(•, •)
	Rd	Rs	I_{16}	/(•, CONST$_{16}$)
srl	Rd	Rs$_1$	Rs$_2$	RSHIFT(•, •)
	Rd	Rs	I_{16}	RSHIFT(•, CONST$_{16}$)
sll	Rd	Rs$_1$	Rs$_2$	LSHIFT(•, •)
	Rd	Rs	I_{16}	LSHIFT(•, CONST$_{16}$)
	Rd	Rs	I_{16}	×(•, CONST$_{2^k}$)
sra	Rd	Rs$_1$	Rs$_2$	ARSHIFT(•, •)
	Rd	Rs	I_{16}	ARSHIFT(•, CONST$_{16}$)
	Rd	Rs	I_{16}	/(•, CONST$_{2^k}$)
lw	Rd	I_{16}(Rb)		MEM(+(•, CONST$_{16}$)), MEM(+(CONST$_{16}$, •)), MEM(CONST$_{16}$), MEM(•)

Notation:

r_i	register i
Rd	destination register
Rs	source register
Rb	base register
I	32-bit immediate
I_{16}	16-bit immediate
label	code label

Addressing modes:

- register: R
- indexed: I_{16}(Rb)
- immediate: I_{16}

. . .

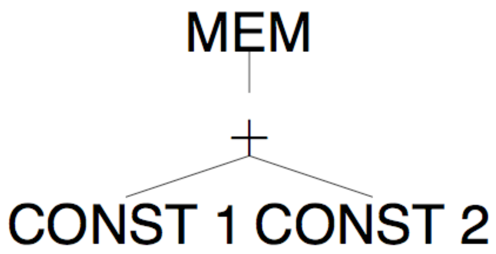
Optimal tiling

- **“Maximal munch”**
  - Start at root of tree
  - Tile root with largest tile that fits
  - Repeat for each subtree

- *NB:* (locally) optimal ≠ (global) optimum
  - *optimum*: least cost instructions sequence (shortest, fewest cycles)
  - *optimal*: no two adjacent tiles combine to a lower cost tile
  - CISC instructions have complex tiles ⇒ optimal ≠ optimum
  - RISC instructions have small tiles ⇒ optimal ≈ optimum

Optimum tiling

- Dynamic programming
  - Assign cost to each tree node — sum of instruction costs of best tiling for that node (including best tilings for children)



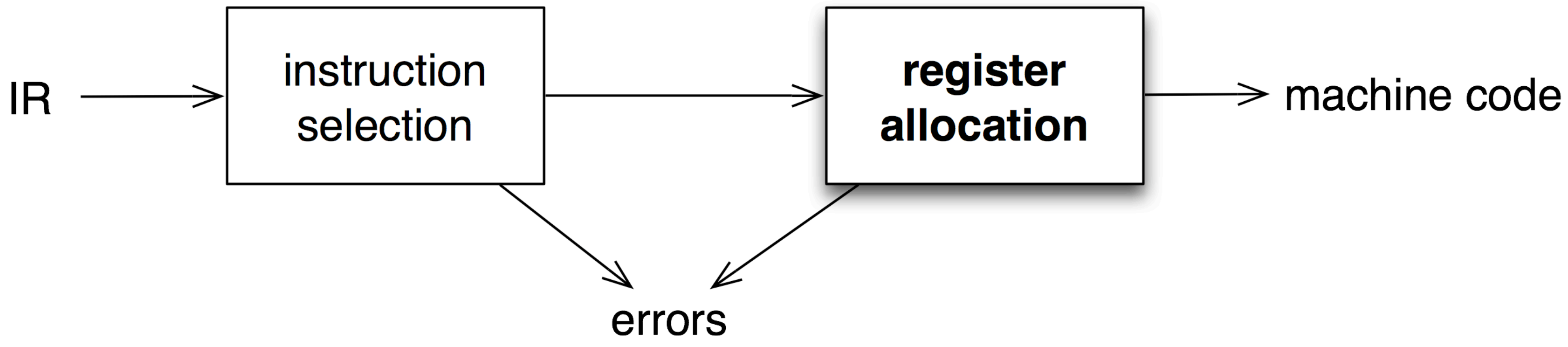
Tile	Instruction	Tile Cost	Leaves Cost	Total Cost
+(•, •)	add	1	1+1	3
+(•, CONST 2)	add	1	1+0	2
+(CONST 1, •)	add	1	0+1	2

http://en.wikipedia.org/wiki/Dynamic_programming

Roadmap

- > Runtime storage organization
- > Procedure call conventions
- > Instruction selection
- > **Register allocation**
- > Example: generating Java bytecode

Register allocation



- > Want to have value in register when used
  - — limited resources
  - — changes instruction choices
  - — can move loads and stores
  - — optimal allocation is difficult (NP-complete)

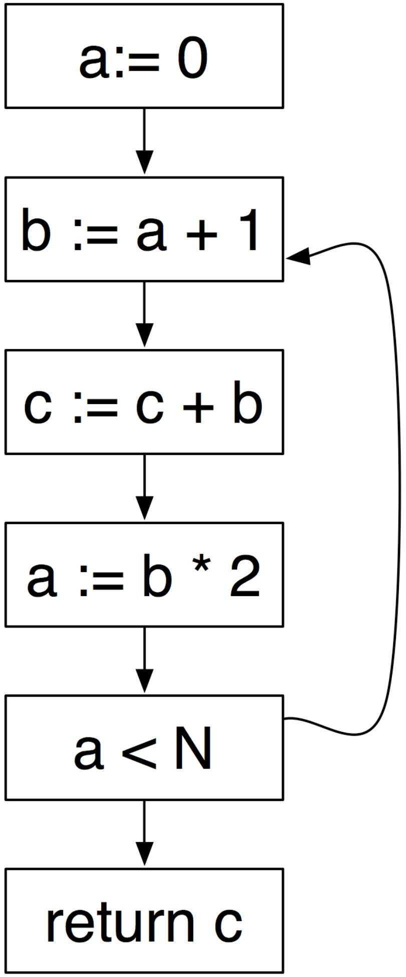
Liveness analysis

- ***Problem:***
  - IR has unbounded # temporaries
  - Machines has bounded # registers
- ***Approach:***
  - Temporaries with disjoint *live* ranges can map to same register
  - If not enough registers, then *spill* some temporaries (i.e., keep in memory)
- The compiler must perform *liveness analysis* for each temporary
  - It is *live* if it holds a value that may still be needed

Control flow analysis

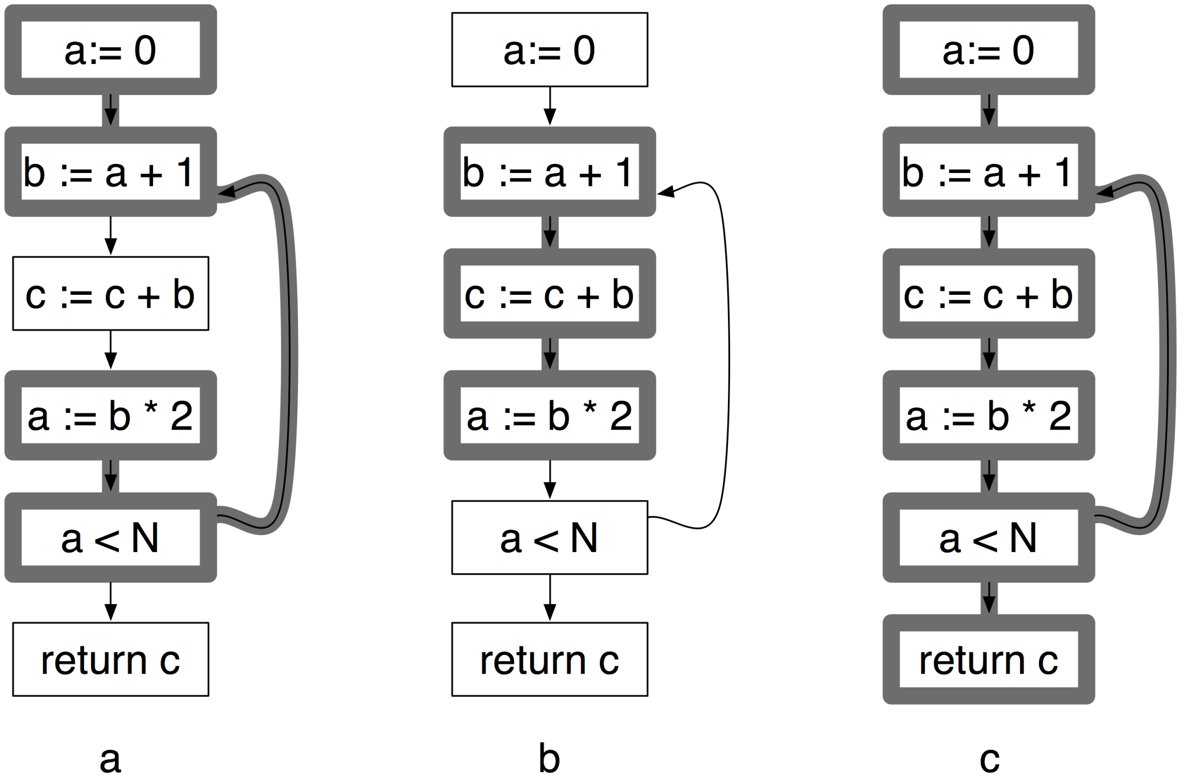
- Liveness information is a form of data flow analysis over the *control flow graph* (CFG):
  - Nodes may be individual program statements or basic blocks
  - Edges represent potential flow of control

```
       $a \leftarrow 0$
$L_1$: $b \leftarrow a + 1$
       $c \leftarrow c + b$
       $a \leftarrow b \times 2$
       if $a < N$ goto $L_1$
       return $c$
```



Liveness

A variable *v* is *live* on edge *e* if there is a path from *e* to a use of *v* not passing through a definition of *v*



a and b are never live at the same time, so two registers suffice to hold a, b and c

Roadmap

- Runtime storage organization
- Procedure call conventions
- Instruction selection
- Register allocation
- **Example: generating Java bytecode**

The visitor

```
package compiler;
...
public class CompilerVisitor extends DepthFirstVisitor {
  Generator gen;

  public CompilerVisitor(String className) {
    gen = new Generator(className);
  }

  public void visit(Assignment n) {
    n.f0.accept(this);
    n.f1.accept(this);
    n.f2.accept(this);
    String id = n.f0.f0.tokenImage;
    gen.assignValue(id);
  }

  public void visit(PrintStm n) {
    n.f0.accept(this);
    gen.prepareToPrint();
    n.f1.accept(this);
    n.f2.accept(this);
    n.f3.accept(this);
    gen.stopPrinting();
  }
  ...
}
```

This time the visitor is responsible for generating bytecode.

Bytecode generation with BCEL

```
package compiler;
...
import org.apache.bcel.generic.*;
import org.apache.bcel.Constants;

public class Generator {
  private Hashtable<String,Integer> symbolTable;
  private InstructionFactory factory;
  private ConstantPoolGen cp;
  private ClassGen cg;
  private InstructionList il;
  private MethodGen method;
  private final String className;

  public Generator (String className) {
    this.className = className;
    symbolTable = new Hashtable<String,Integer>();
    cg = new ClassGen(className, "java.lang.Object", className + ".java",
        Constants.ACC_PUBLIC | Constants.ACC_SUPER, new String[] {});

    cp = cg.getConstantPool();
    factory = new InstructionFactory(cg, cp);

    il = new InstructionList();
    method = new MethodGen(Constants.ACC_PUBLIC | Constants.ACC_STATIC,
        Type.VOID, new Type[] { new ArrayType(Type.STRING, 1) },
        new String[] { "arg0" }, "main", className, il, cp);
  }
...
```

We introduce a separate class to introduce a higher-level interface for generating bytecode

Creates a class with a static main!

Invoking print methods

```
private void genPrintTopNum() {
  il.append(factory.createInvoke("java.io.PrintStream", "print",
      Type.VOID, new Type[] { Type.INT }, Constants.INVOKEVIRTUAL));
}

private void genPrintString(String s) {
  pushSystemOut();
  il.append(new PUSH(cp, s));
  il.append(factory.createInvoke("java.io.PrintStream", "print",
      Type.VOID, new Type[] { Type.STRING }, Constants.INVOKEVIRTUAL));
}

private void pushSystemOut() {
  il.append(factory.createFieldAccess(
      "java.lang.System", "out",
      new ObjectType("java.io.PrintStream"), Constants.GETSTATIC));
}

public void prepareToPrint() {
  pushSystemOut();
}

public void printValue() {
  genPrintTopNum();
  genPrintString(" ");
}

public void stopPrinting() {
  genPrintTopNum();
  genPrintString("\n");
}
```

To print, we must push System.out on the stack, push the arguments, then invoke print.

Binary operators

```
public void add() {
  il.append(new IADD());
}

public void subtract() {
  il.append(new ISUB());
}

public void multiply() {
  il.append(new IMUL());
}

public void divide() {
  il.append(new IDIV());
}

public void pushInt(int val) {
  il.append(new PUSH(cp, val));
}
```

Operators simply consume the top stack items and push the result back on the stack.

Variables

```
public void assignValue(String id) {
  il.append(factory.createStore(Type.INT, getLocation(id)));
}

public void pushId(String id) {
  il.append(factory.createLoad(Type.INT, getLocation(id)));
}

private int getLocation(String id) {
  if(!symbolTable.containsKey(id)) {
    symbolTable.put(id, 1+symbolTable.size());
  }
  return symbolTable.get(id);
}
```

Variables must be translated to locations. BCEL keeps track of the needed space.

Code generation

```
public void generate(File folder) throws IOException {
  il.append(InstructionFactory.createReturn(Type.VOID));
  method.setMaxStack();
  method.setMaxLocals();
  cg.addMethod(method.getMethod());
  il.dispose();
  OutputStream out =
    new FileOutputStream(new File(folder, className + ".class"));
  cg.getJavaClass().dump(out);
}
```

Finally we generate the return statement, add the method, and dump the bytecode.

Generated class files

```
public class Eg3 {
  public static void main(java.lang.String[] arg0);
     0  getstatic java.lang.System.out : java.io.PrintStream [12]
     3  iconst_1
     4  istore_1
     5  iload_1
     6  iload_1
     7  iload_1
     8  imul
     9  iadd
    10  iload_1
    11  iadd
    12  istore_1
    13  iload_1
    14  invokevirtual java.io.PrintStream.print(int) : void [18]
    17  getstatic java.lang.System.out : java.io.PrintStream [12]
    20  ldc <String " "> [20]
    22  invokevirtual java.io.PrintStream.print(java.lang.String) : void [23]
    25  getstatic java.lang.System.out : java.io.PrintStream [12]
    28  iload_1
    29  iconst_1
    30  iadd
    31  invokevirtual java.io.PrintStream.print(int) : void [18]
    34  getstatic java.lang.System.out : java.io.PrintStream [12]
    37  ldc <String "\n"> [25]
    39  invokevirtual java.io.PrintStream.print(java.lang.String) : void [23]
    42  return
}
```

Generated from:

```
"print((a := 1; a := a+a*a+a, a),a+1)"
```

What you should know!

- ✎ *How is the run-time stack typically organized?*
- ✎ *What is the “procedure linkage contract”?*
- ✎ *What is the difference between the FP and the SP?*
- ✎ *What are storage classes for variables?*
- ✎ *What is “maximal munch”?*
- ✎ *Why is liveness analysis useful to allocate registers?*
- ✎ *How does BCEL simplify code generation?*

Can you answer these questions?

- ✎ *Why does the run-time stack grow down and not up?*
- ✎ *In Java, which variables are stored on the stack?*
- ✎ *Does Java support downward or upward exposure of local variables?*
- ✎ *Why is optimal tiling not necessarily the optimum?*
- ✎ *What semantic analysis have we forgotten to perform in our straightline to bytecode compiler?*

License