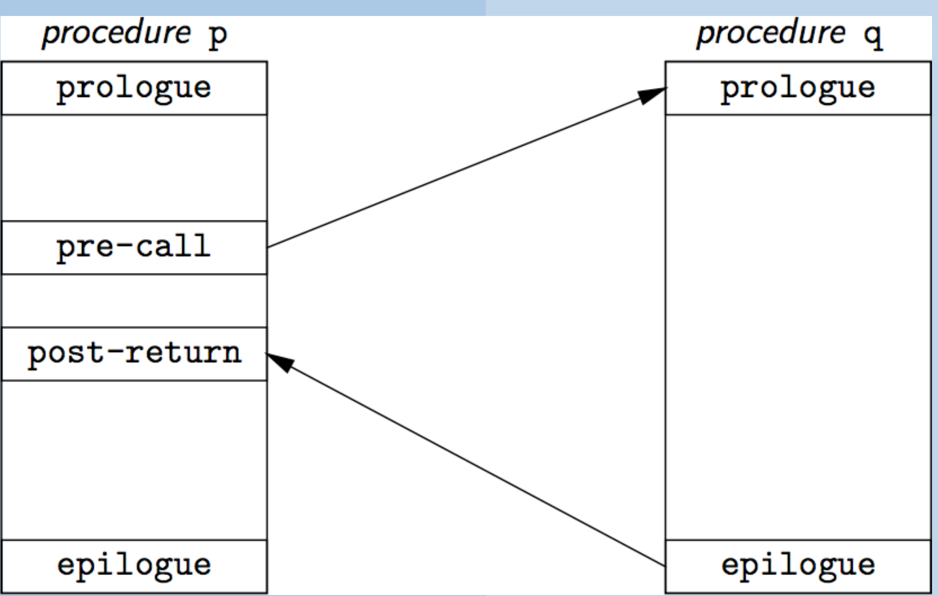
7. Code Generation

Oscar 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.

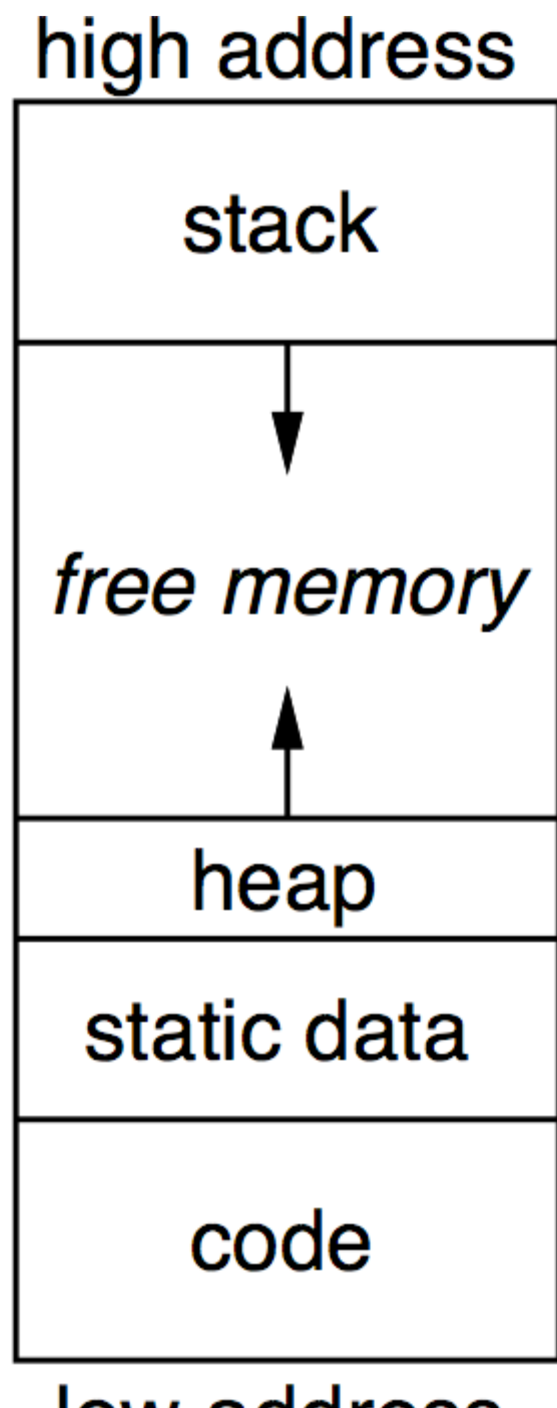
Roadmap

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

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.



In a 32-bit architecture, memory addresses range from from 0 to 4GB, broken into pages, of which only the low and the high pages are actually allocated. The low pages hold the compiled program code, static data, and heap data. The heap “grows” upwards as needed. High memory addresses refer to the run-time stack. They “grow” downward with each procedure call and “shrink” upward with each return.

Certain memory pages (e.g., holding compiled code) may possibly be protected against modification.

The Procedure Abstraction

- The *procedure abstraction* supports separate compilation
  - build large programs
  - keep compile times reasonable
  - independent procedures
- The linkage convention (calling convention):
  - *a social contract* — procedures inherit a valid run-time environment *and* restore one for their parents
  - *platform dependent* — code generated at compile time

Procedures as abstractions

```
function foo()
{
    int a, b;
    ...
    bar(a);
    ...
}
```

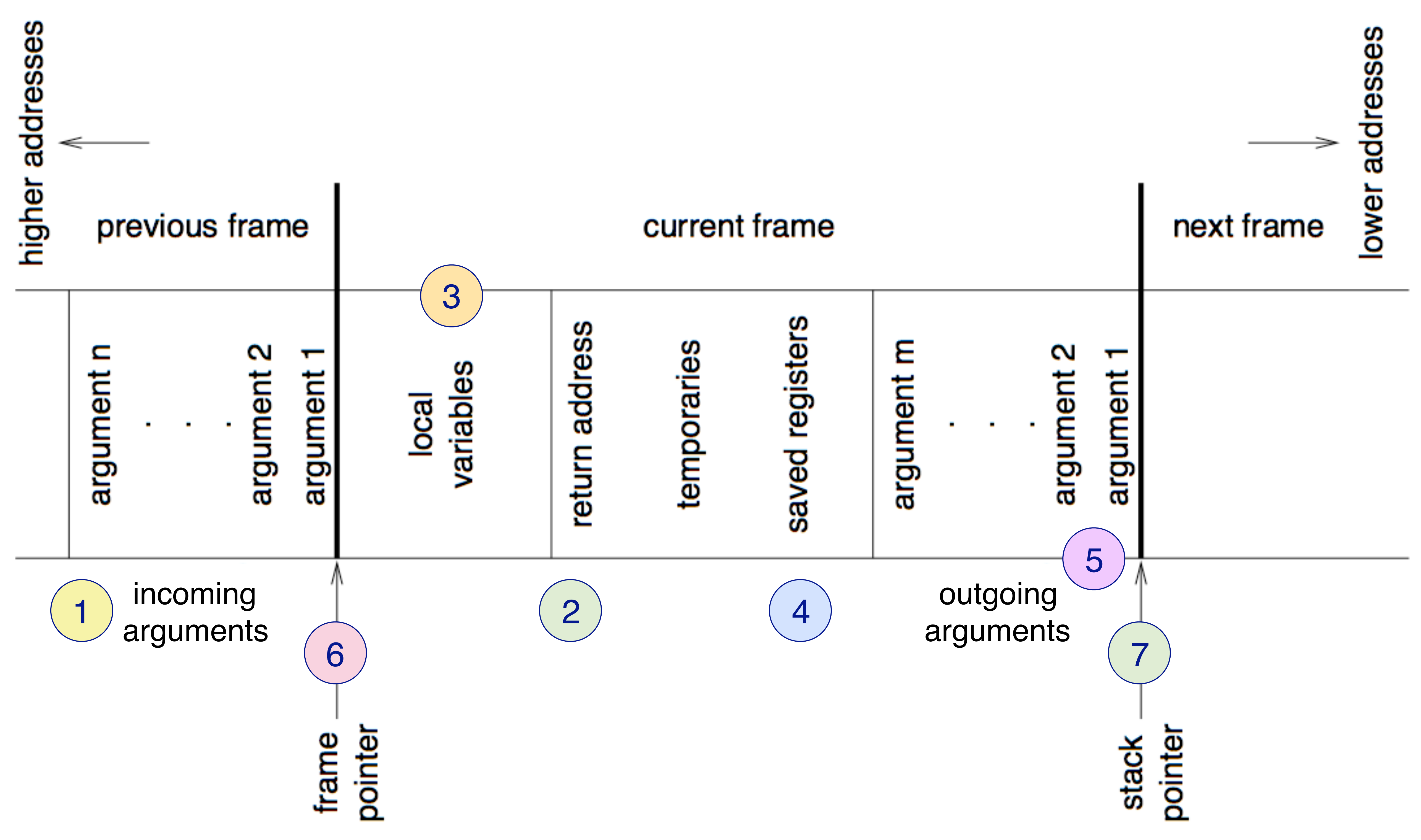
```
function bar(int a)
{
    int x;
    ...
    bar(x);
    ...
}
```

bar() must preserve foo()'s state while executing.
what if bar() is recursive?

Every procedure requires memory to store its arguments, local variables, and other information (e.g., where to return to). In FORTRAN, which originally did not support recursion, the memory was allocated statically. With the introduction of recursion, each procedure invocation needs its own private memory region which is dynamically allocated (and released). This naturally leads to the construct of the run-time procedure call stack.

See also: https://en.wikipedia.org/wiki/Call_stack

Activation records



Each procedure activation has an "*activation record*" or "*stack frame*". Details may vary depending on the implementation. In this example:

1. the callee's *incoming arguments* are stored in the caller's stack frame
2. the callee stores the *return address* in the code to jump to on completion
3. each frame has space to hold the *local variables* of the procedure
4. the callee saves the values of the *registers* (to be restored on return)
5. *outgoing arguments* for further calls are stored here (just like 1 above)
6. the *frame pointer* identifies the start of the *current frame* on the stack
7. the *stack pointer* points to the *end of the stack*

Registers

- Typical machine has many of them
- Caller-save vs. Callee-save
  - Convention depending on architecture
  - Used for nifty optimizations
    - *When value is not needed after call the caller puts the value in a caller-save register*
    - *When value is needed in multiple called functions the caller saves it only once*
- Parameter passing put first *k* arguments in registers (*k=4..6*)
  - avoids needless memory traffic because of
    - *leaf procedures (many)*
    - *interprocedural register allocation*
  - same with the return address

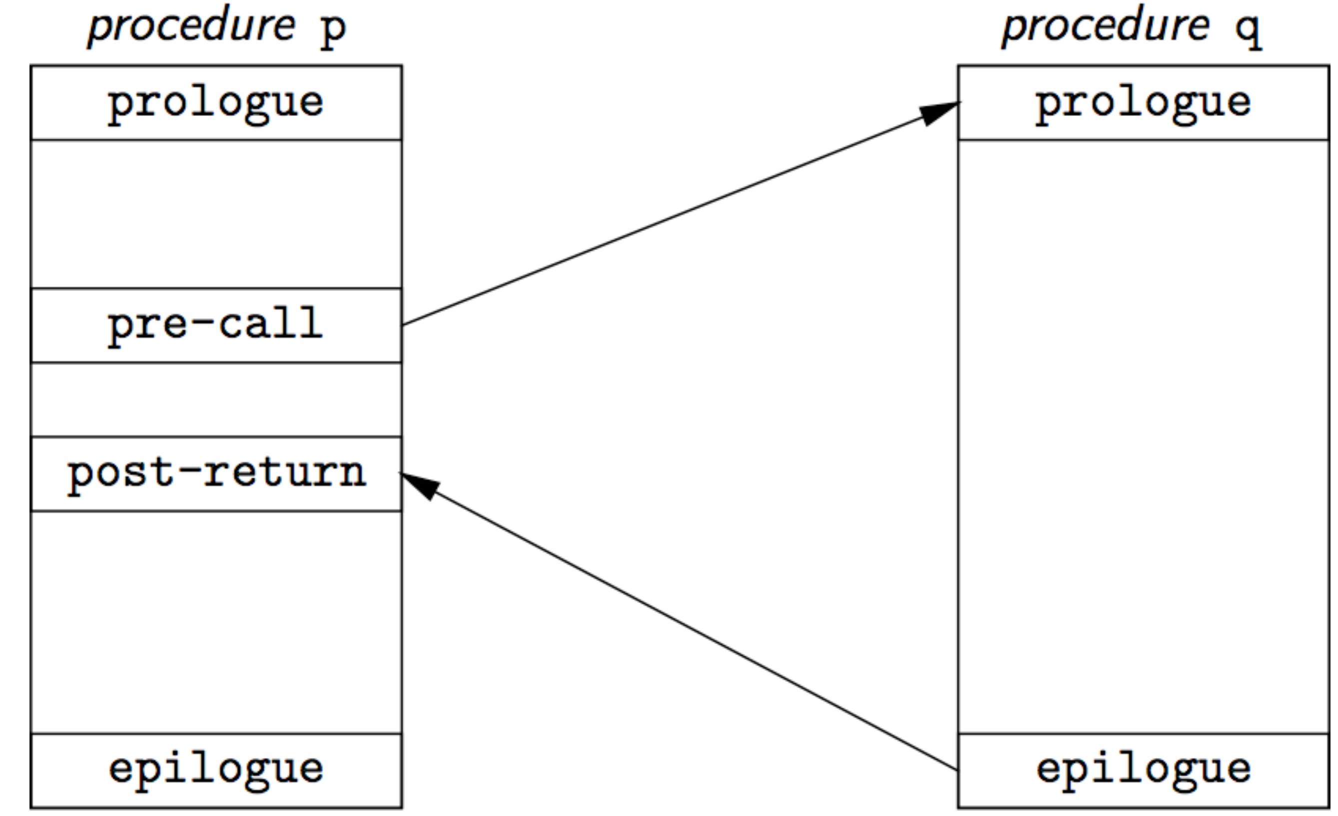
Recall that a limited number of registers constitute the working memory of the CPU. Values must be explicitly *loaded from memory* to perform operations with them, or *stored back to memory* when they are modified.

Whenever a procedure calls another one, the current values of the registers must be saved so that the callee may freely use all the registers, and upon return they must be restored so that the caller can safely assume that the registers are still valid.

The responsibility for saving and restoring registers may lie either with the caller or the callee (with various tradeoffs).

Procedures as control abstractions

- **On entry**, establish *p*'s environment
- **During a call**, preserve *p*'s environment
- **On exit**, tear down *p*'s environment

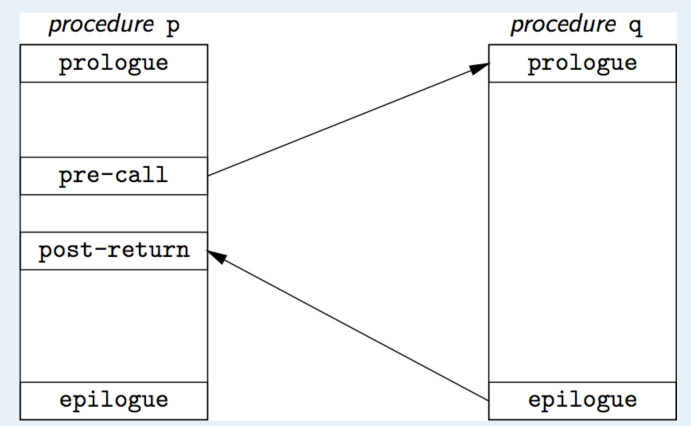


The prologue of a procedure p prepares the stack and registers for the body of the procedure to run. Typical actions are to push the current stack and frame pointers to allocate a new frame.

The epilogue performs the reverse clean up actions.

See also: http://en.wikipedia.org/wiki/Function_prologue

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

At compile time, generate code to do this.

At run time, the code manipulates the frame and data areas.

A *basic frame* does not (yet) have space for local data.

The *static link* is for nested functions – the static link points to the frame of the enclosing function (if any) [Appel p 124].

Variable scoping

Who sees local variables? Where can they be allocated?

Downward exposure

- called procedures see caller variables
- dynamic scoping vs lexical scoping

Upward exposure

- procedures can return references to variables
- functions that return functions

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

With *downward exposure*, a procedure can pass (a reference to) a local variable to a called procedure. The variable can be allocated on the stack since it will be guaranteed to be available to all called procedures.

With *dynamic scoping*, callees have access to the environment of the caller (this is evil, and generally avoided in modern programming languages).

With *upward exposure*, if a called procedure returns a reference to a local variable, that variable must be allocated on the heap, as the stack frame of the callee will be popped after the return.

Functions that return functions (i.e., blocks or anonymous functions) are just a generalization of this idea. Such functions may capture the environment in which they are defined, in which case the whole environment may need to persist after the callee returns.

Higher-order functions

```
fun add(x)
  let fun sum(y) = x+y
  return sum
end

val inc = add(1)
val dec = add(-1)

val x = inc(5)
val y = dec(6)
```

Nested functions

+

Functions returned as values

=

Higher-order functions

Pascal has nested functions but no functions returned as values. C has functions as values but not nested. ML, Scheme, Smalltalk, Java etc. all have higher-order functions.

In the example, function `add()` returns locally-defined function `sum()`. Note that `sum()` uses the argument `x` passed to `add()`, hence this value must continue to be available after `add()` returns `summ()` to its caller. The environment of `add()` therefore cannot exist purely on the run-time stack, as it will disappear after `add()` returns.

Lexically nested scopes

- view variables as (level, offset) pairs
  - reflects scoping
- helps 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
  - table of access links (display)

Modern languages are lexically scoped. A nested scope has access to its surrounding scope. A nested block has its own local variables, but can access the variables of its surroundings blocks. Similarly, nested functions have their own local variables, but may access the variables of their surrounding functions.

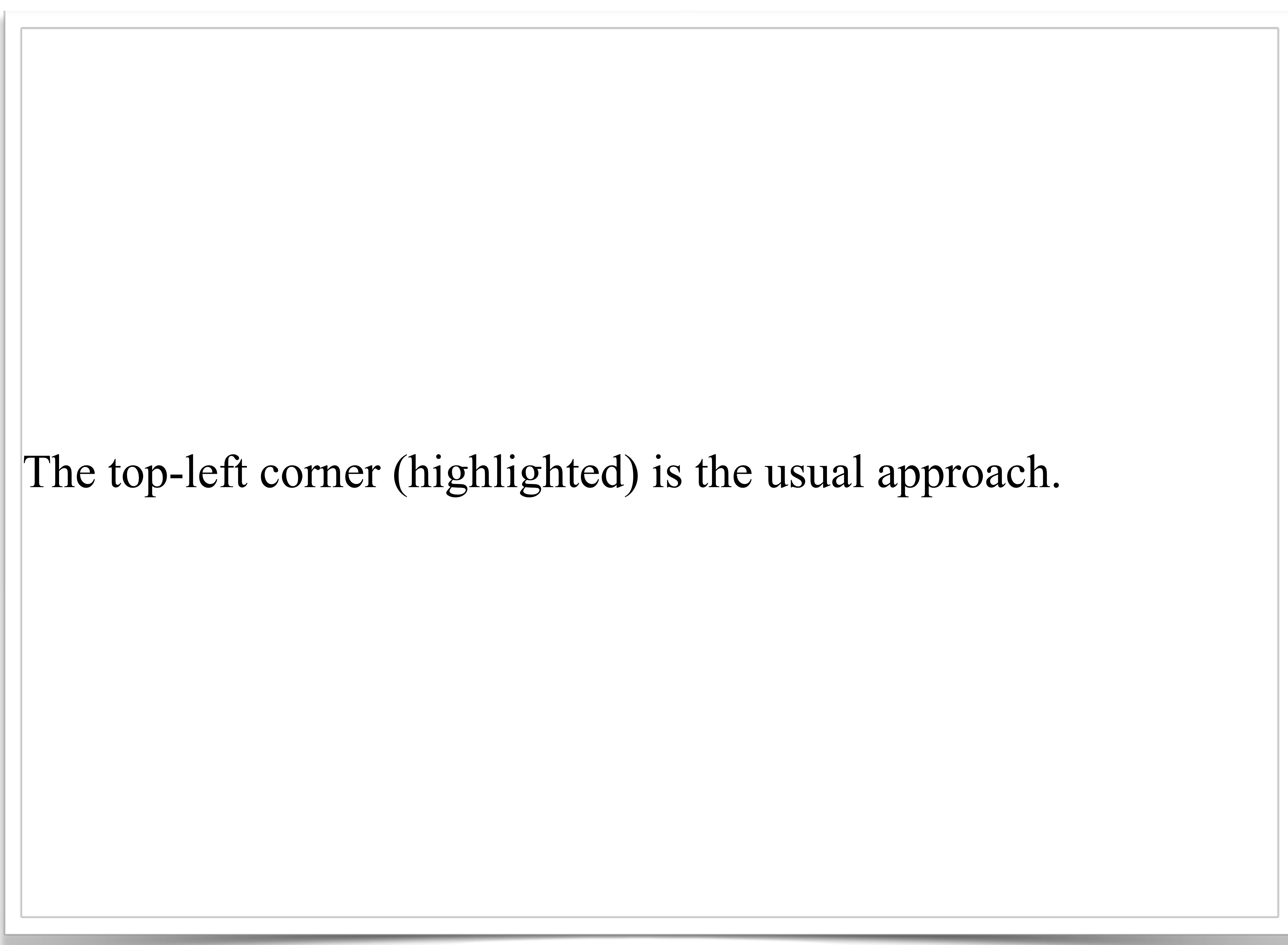
By encoding variables as (level,offset) pairs, the run-time system can keep track of which scope variables are in, and whether they are to be found in the current stack frame or an earlier one.

Roadmap

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

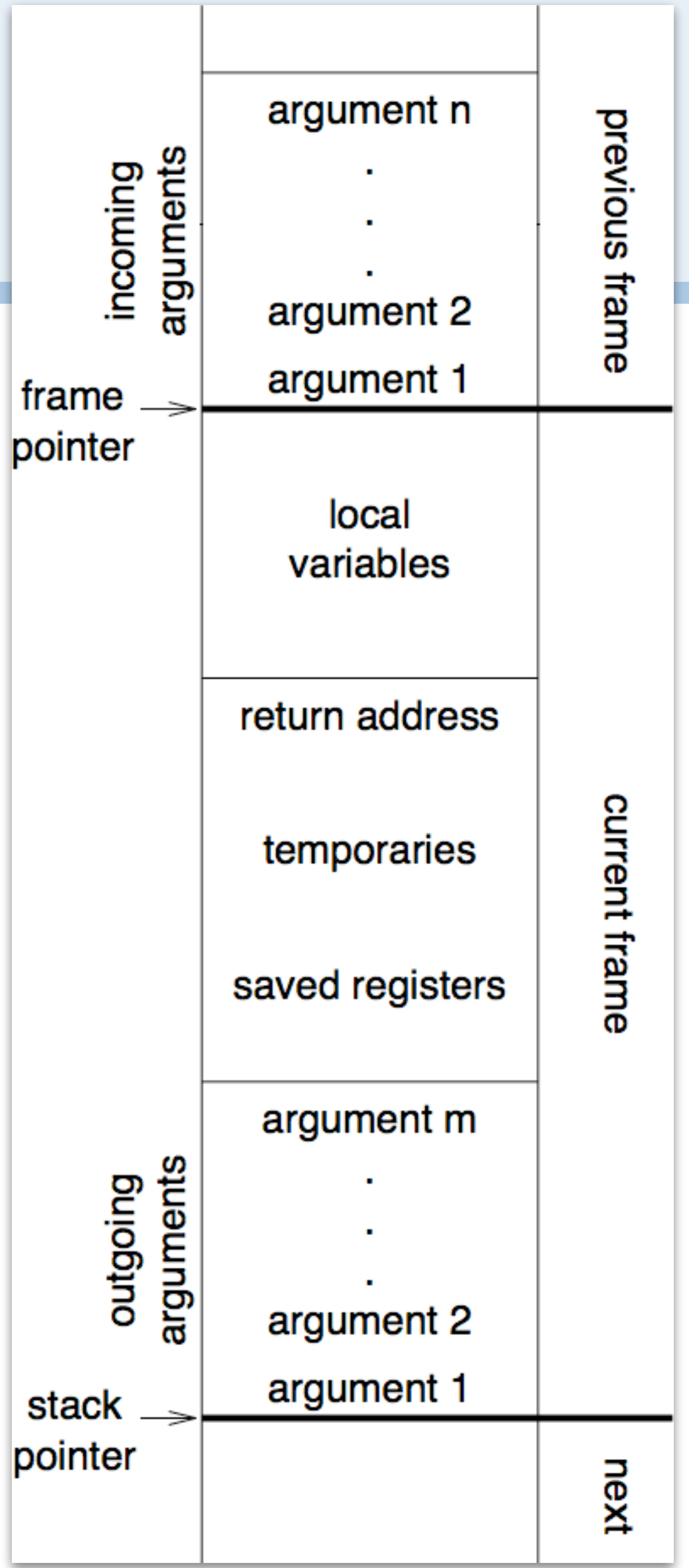
Calls: Saving and restoring registers

	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.


The top-left corner (highlighted) is the usual approach.

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



See also slide 10 (Procedure linkage contract)

Note that the stack in the figure is upside down (grows "downward", so the "top" is at the bottom). The caller leaves space at the "top" for the return value. The "incoming arguments" here belong to the caller, but they can also be part of the callee's stack frame. (The terminology is confusing, as "incoming" is also used to refer to parameters that the callee can use to store results.)

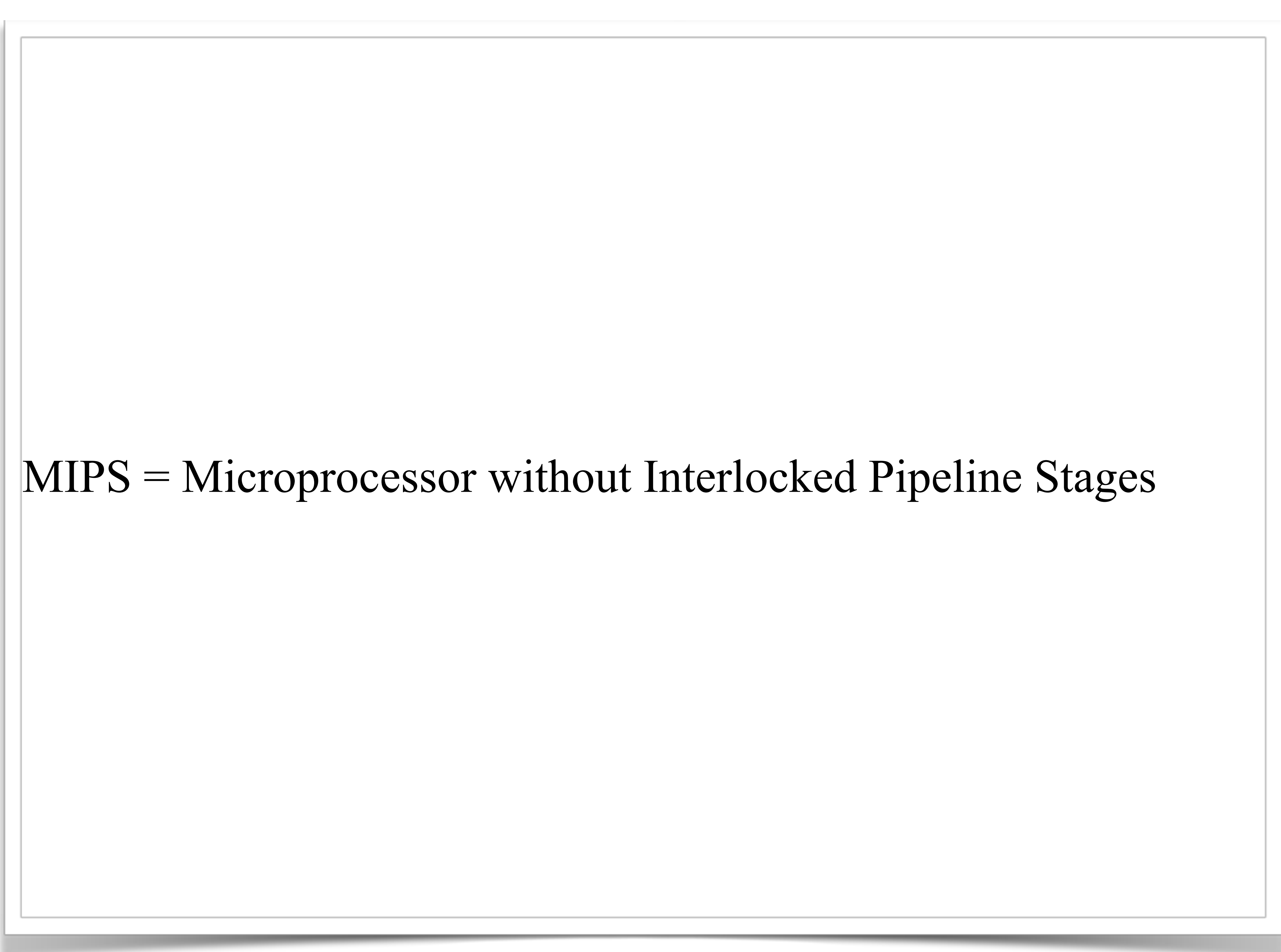
The caller here pushes space for locals etc onto the callee's stack frame, and evaluates and pushes the actual arguments onto the stack (presumably the "incoming" arguments).

The callee saves the registers in the area allocated by the caller, and restores them on return. When the callee returns, it leaves just the return value on the stack. The epilogue of the caller (invoked at the return address) will clean up the stack and frame pointers.

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 = Microprocessor without Interlocked Pipeline Stages

MIPS procedure call convention

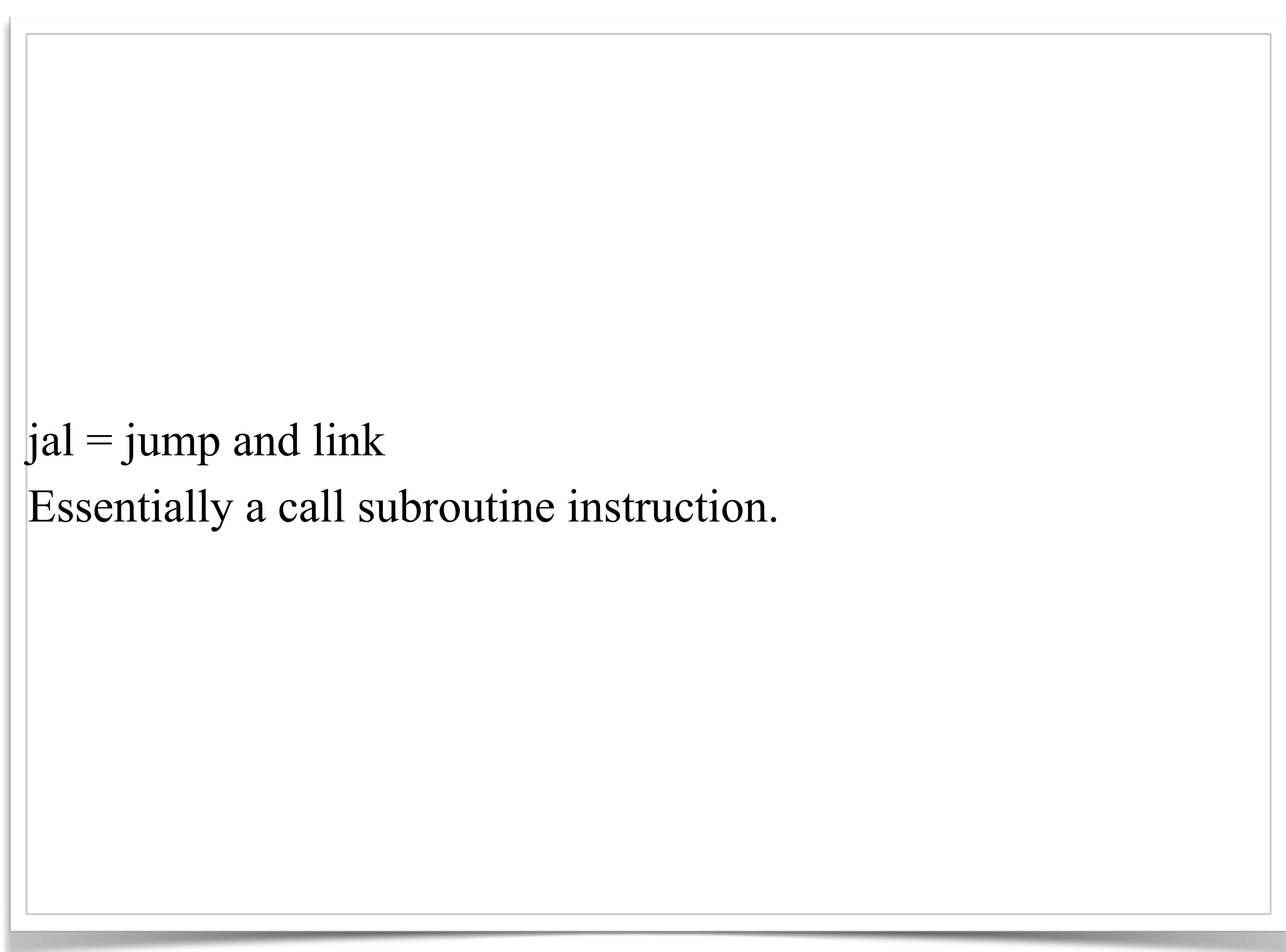
- ***Philosophy:***
  - Use full, general calling sequence only when necessary
  - Omit portions of it where possible
    (e.g., avoid using FP register whenever possible)

- ***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*


jal = jump and link
Essentially a call subroutine instruction.

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

- subu = subtract unsigned
  - This updates the stack pointer by the current framesize. (Recall that the stack grows downward.)
- sw = store word
  - $31 is the return address, $16, $17 are temporaries etc. Store them on the stack at the corresponding offset from the stack pointer.

MIPS procedure call convention

> ***Epilogue:***

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

   ```
   lw reg, framesize+frameoffset-N($sp)
   ```

3. Get return address

   ```
   lw $31, framesize+frameoffset($sp)
   ```

4. Clean up stack

   ```
   addu $sp,framesize
   ```

5. Return

   ```
   j $31
   ```

lw = load word

addu = add unsigned

j = jump

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

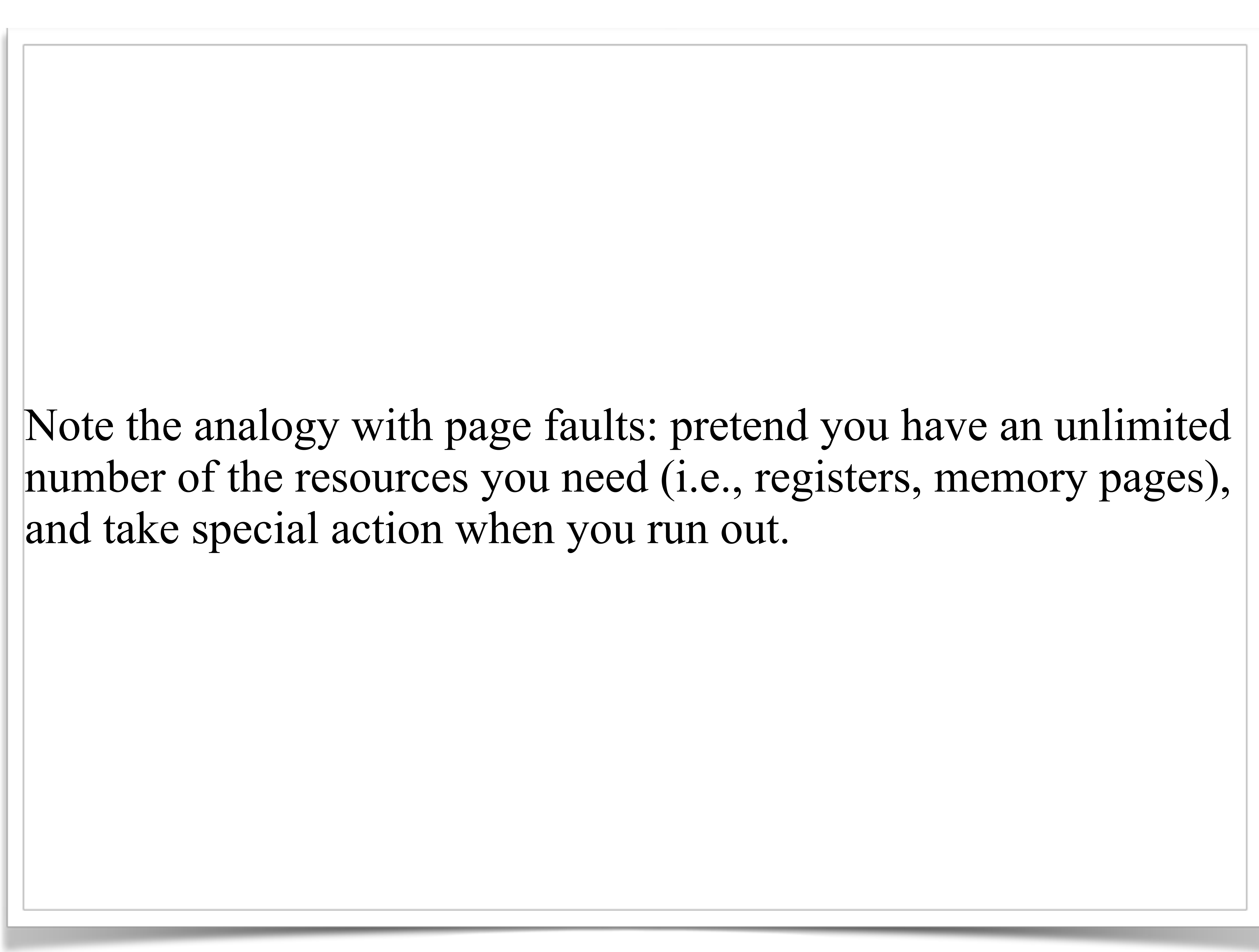
The “maximal munch” principle is the rule that as much of the input as possible should be processed when creating some construct.

In this case, try to macro-expand the largest IR munch that you can match.

See: https://en.wikipedia.org/wiki/Maximal_munch

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


Note the analogy with page faults: pretend you have an unlimited number of the resources you need (i.e., registers, memory pages), and take special action when you run out.

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}

...

At right are tree patterns to match; at left is the code to be emitted.

(The rest of the example is elided.)

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

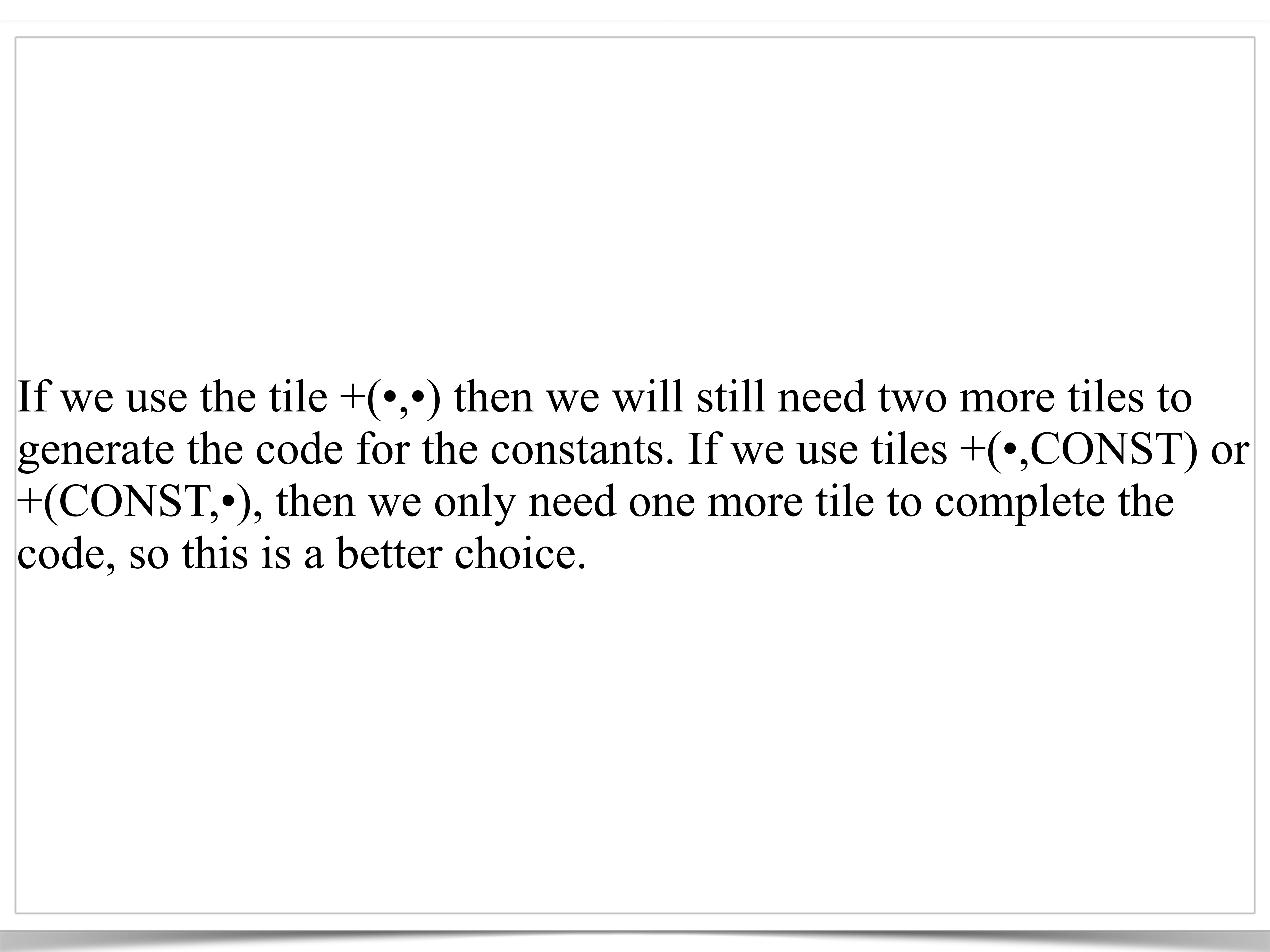
Optimum tiling

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

```
        MEM
         |
         +
        / \
CONST 1    CONST 2
```

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

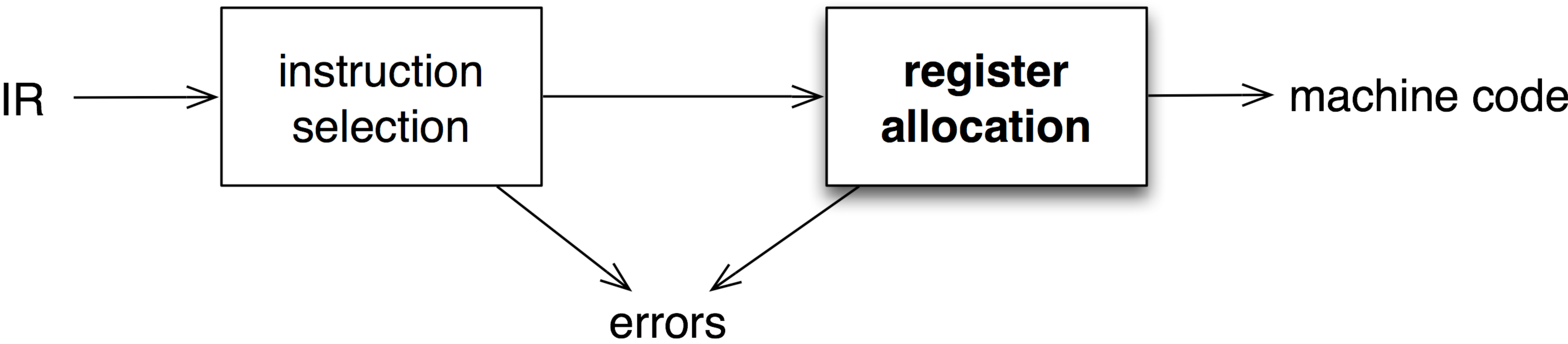
http://en.wikipedia.org/wiki/Dynamic_programming


If we use the tile +(•,•) then we will still need two more tiles to generate the code for the constants. If we use tiles +(•,CONST) or +(CONST,•), then we only need one more tile to complete the code, so this is a better choice.

Roadmap

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

Register allocation



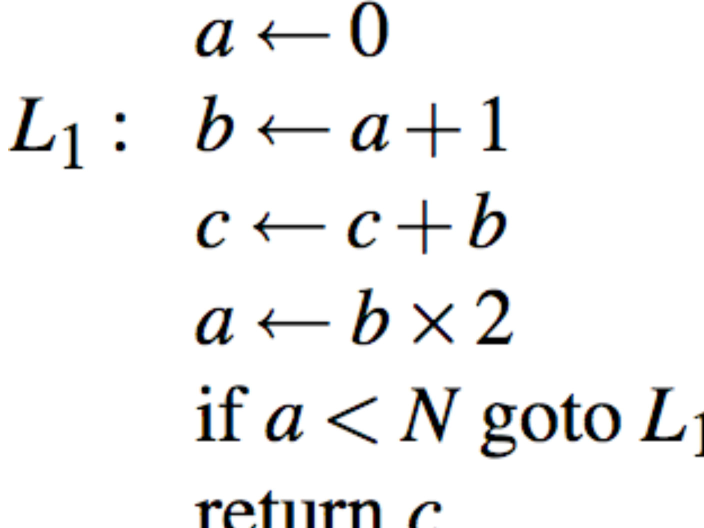
> Want to have value in register when used
  - limited resources
  - changes instruction choices
  - can move loads and stores
  - optimal allocation is difficult (NP-complete)

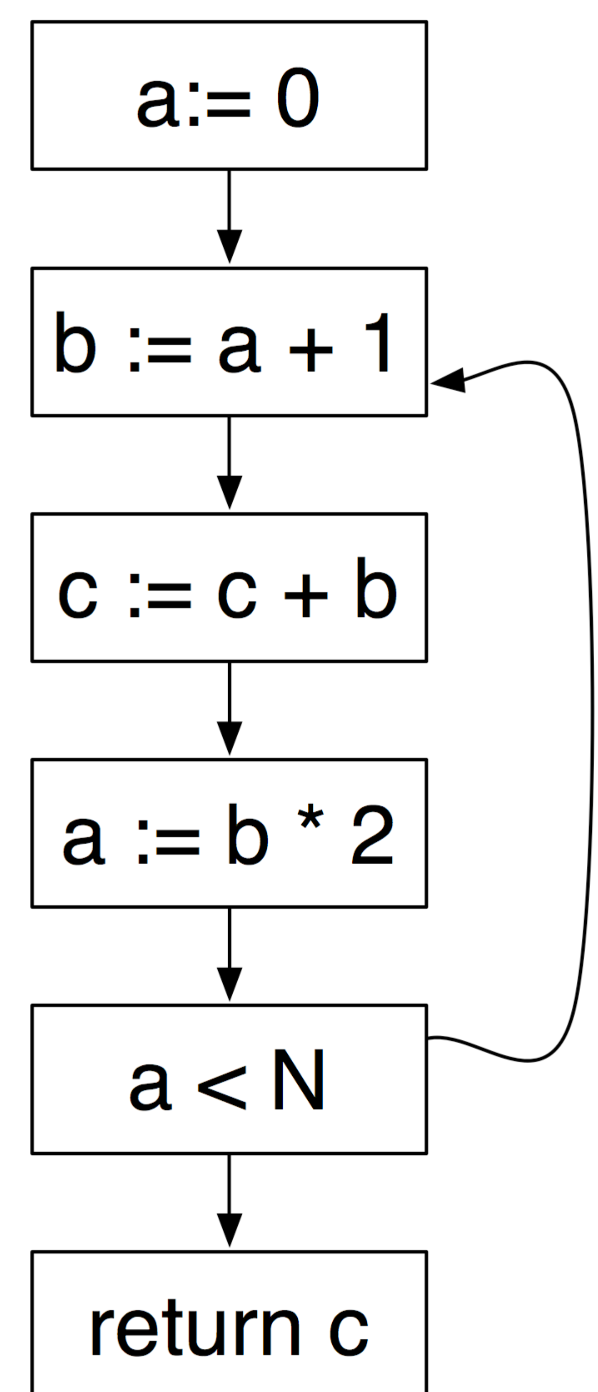
Liveness analysis

- ***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

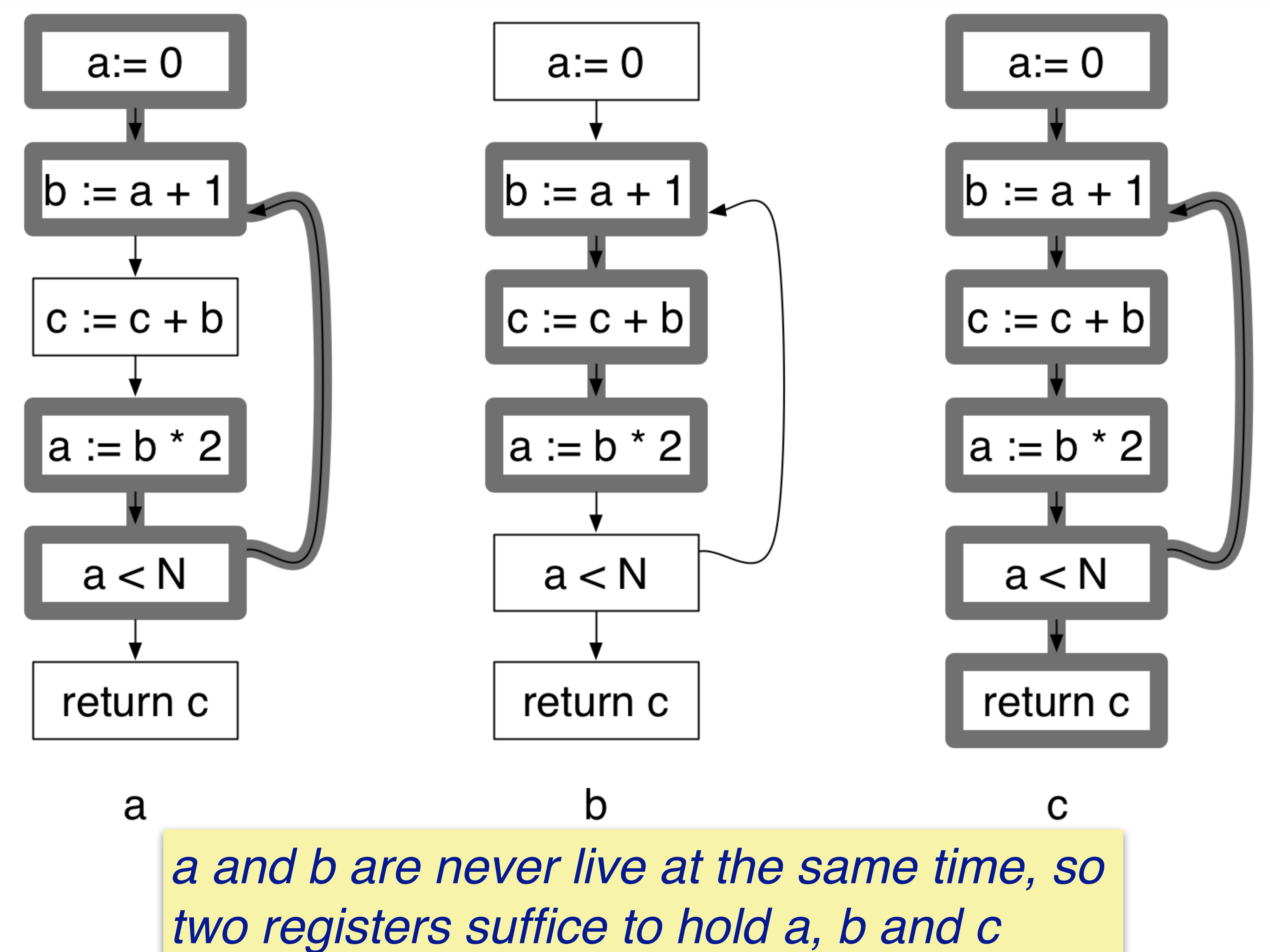
> Liveness information is a form of data flow analysis over the *control flow graph* (CFG):
  - Nodes may be individual program statements or basic blocks
  - Edges represent potential flow of control





Liveness (review)

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

Here we see that a and b are not live at the same time, so two registers suffice: one for both a and b and the other for c.

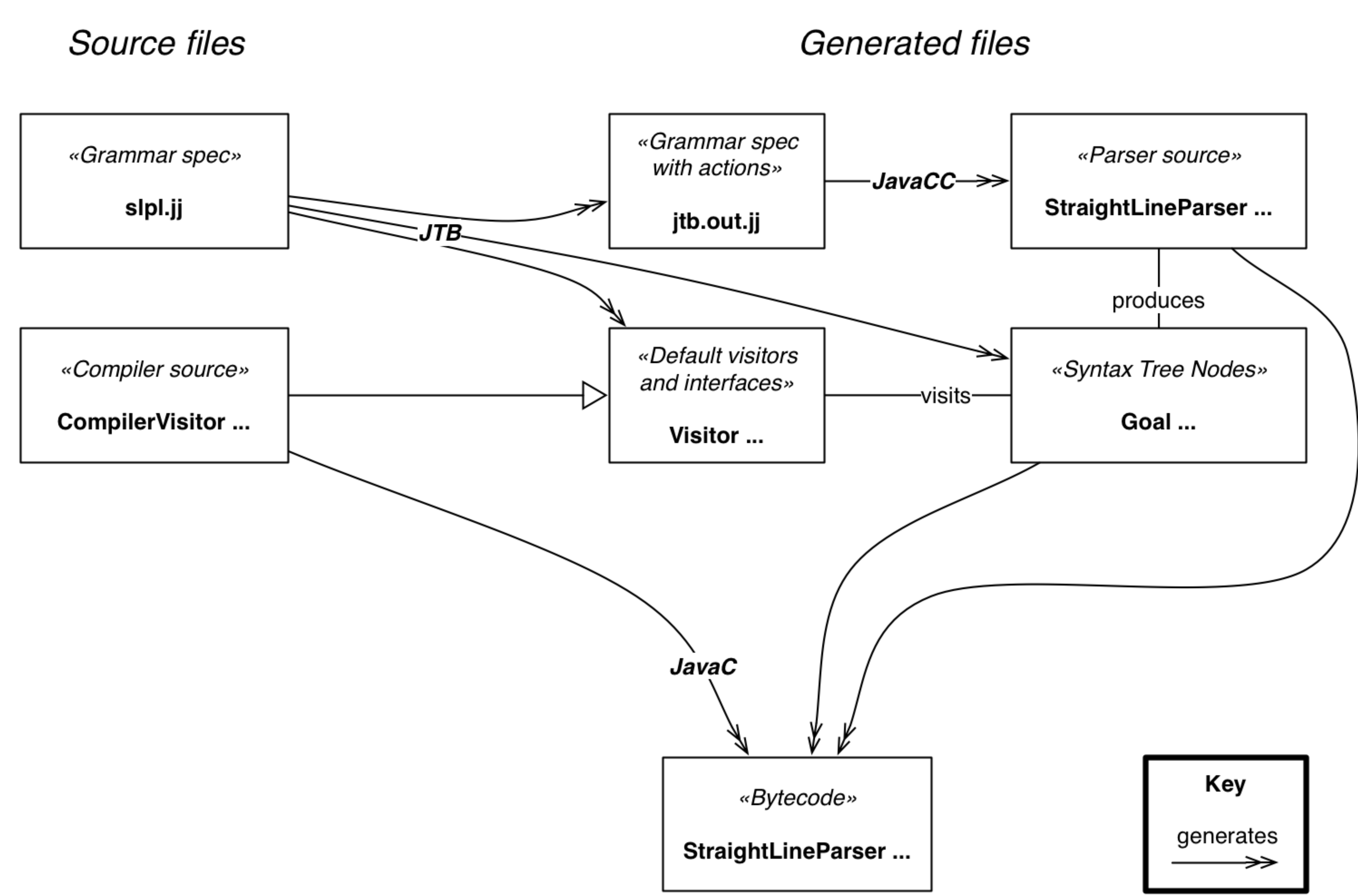
See chapter 10 of Appel (2nd edition) for this example and details of algorithms.

NB: liveness analysis might also reveal errors — e.g., if c is a local, then it has not been initialized.

Roadmap

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

Straightline Compiler Files



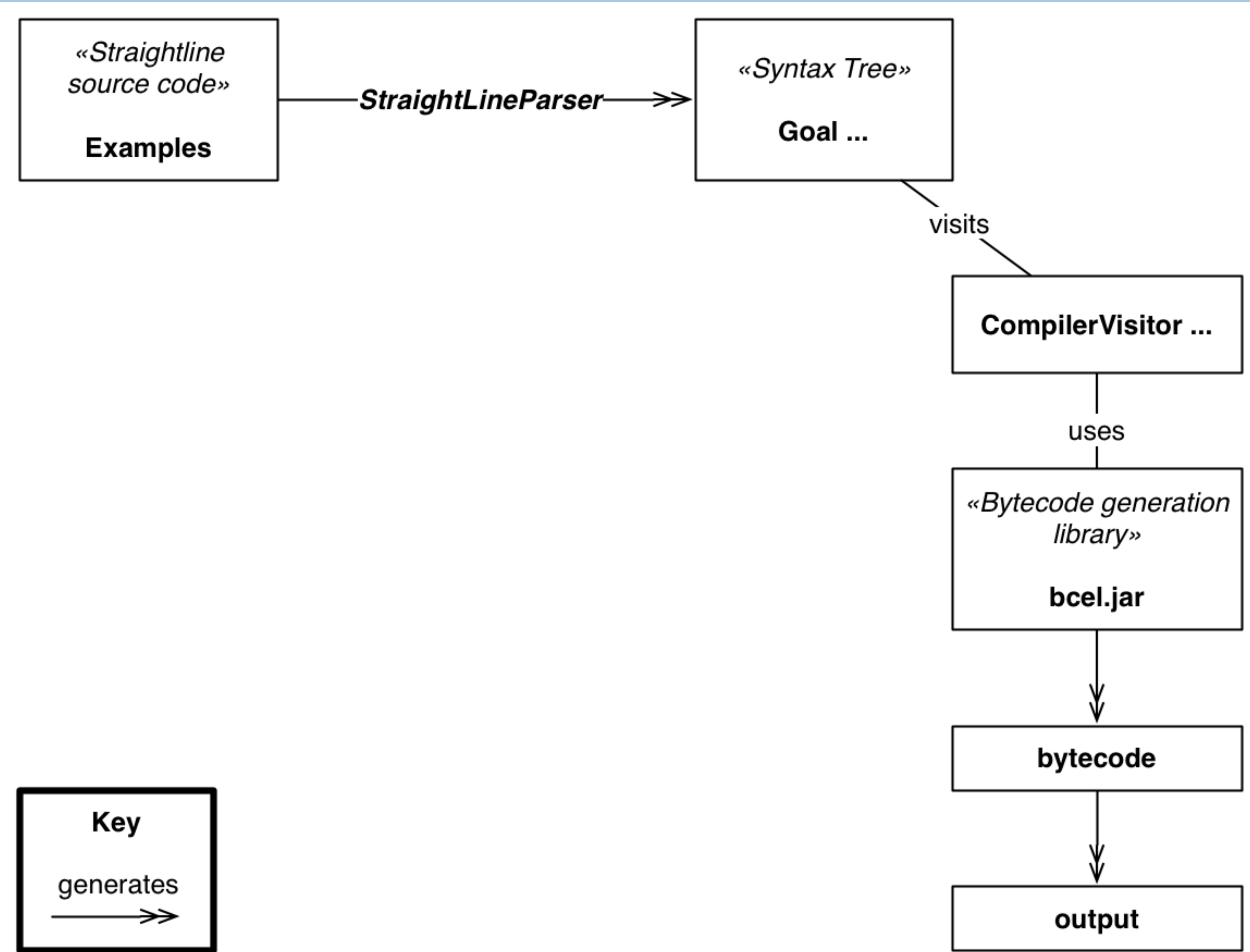
As we have seen (lecture 1: intro, and lecture 4: parsing in practice), we use Java Tree Builder to generate (concrete) syntax tree nodes from the grammar specification of our toy straightline language. JTB also generates default visitors for the syntax tree, and default actions to build the tree in the expanded grammar spec.

Our compiler extends the default visitor, visiting the syntax tree nodes and generating code.

Recall that the source code is available here:

```
git clone git://scg.unibe.ch/lectures-cc-examples
```

Straightline Compiler Runtime



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.

Since the syntax tree generated from our grammar by JTB is rather generic, it has ugly generic names for the field holding subnodes.

We make use of a “Generator” object that keeps track of semantic information and emits the actual bytecode with the help of the BCEL framework.

When we visit a print statement, we first visit the “print” token (f0), then the nodes for “()”, the expression list, and “)” (f1 through f3). We prepare for printing, then visit the individual expression subnodes, and finally complete the printing action.

As we shall see, `gen.prepareToPrint()` pushes the Java `print` command onto the stack. Visiting the expression list will leave the result of each expression on the stack. Finally `gen.stopPrinting()` will cause the `print` method to be executed.

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!

Since our toy language is not object-oriented, we simply generate a class with a single static main method to represent our program.

The Generator holds a symbol table and a number of objects needed to interact with BCEL (such as the `InstructionFactory` and the `InstructionList`).

The Java VM is stack-based machine, so the instructions we generate push values onto the stack, or evaluate instructions that pop values from the top of the stack and leave behind the result.

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.

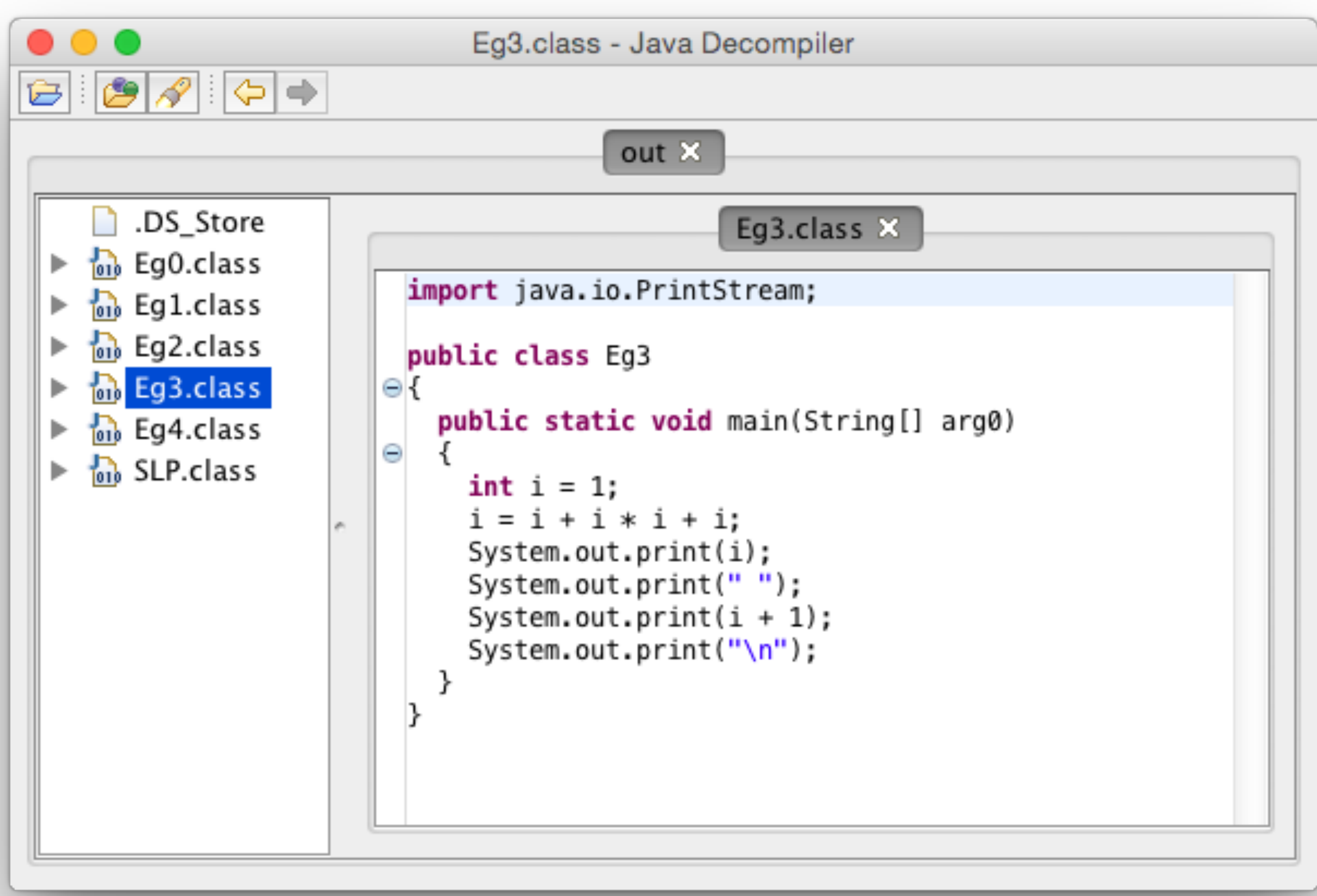
Generated class files

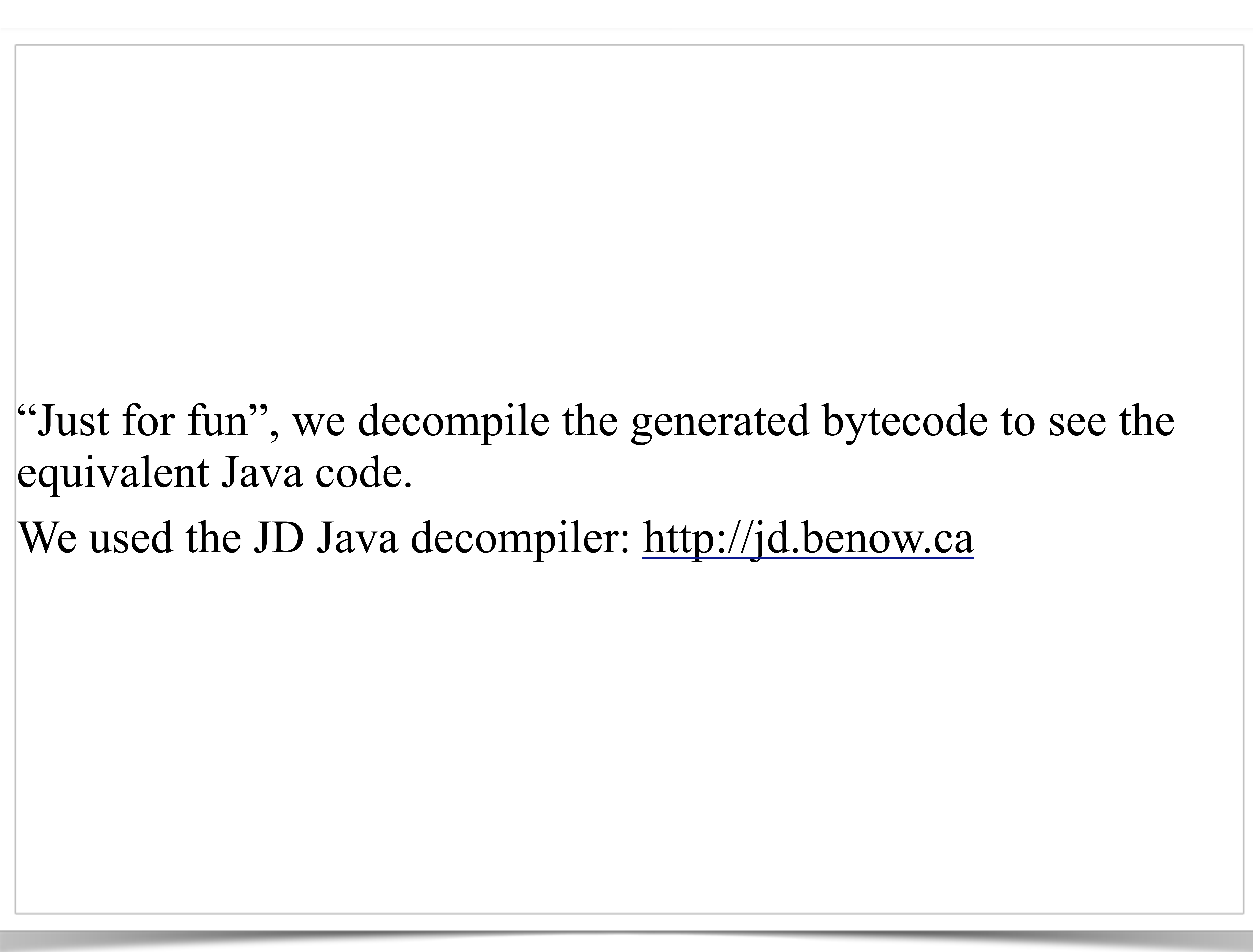
```
public class Eg3 {
  public static void main(java.lang.String[] arg0);
     0  getstatic java.lang.System.out : java.io.PrintStream [12]
     3  iconst_1
     4  istore_1
     5  iload_1
     6  iload_1
     7  iload_1
     8  imul
     9  iadd
    10  iload_1
    11  iadd
    12  istore_1
    13  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)"
```

Decompiling the generated class files




“Just for fun”, we decompile the generated bytecode to see the equivalent Java code.
We used the JD Java decompiler: http://jd.benow.ca

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?*

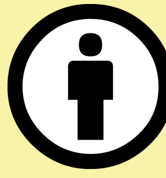
Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

You are free to:

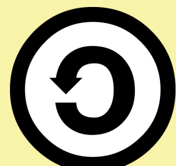
Share — copy and redistribute the material in any medium or format
Adapt — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:



Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.



ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

http://creativecommons.org/licenses/by-sa/4.0/