Lecture #15: Introduction to Runtime Organization

Status

- Lexical analysis
  - Produces tokens
  - Detects & eliminates illegal tokens
- Parsing
  - Produces trees
  - Detects & eliminates ill-formed parse trees
- Static semantic analysis
  - Produces *decorated tree* with additional information attached
  - Detects & eliminates remaining static errors
- Next are the dynamic "back-end" phases: $\Longleftarrow$ *we are here*
  - Code generation (at various semantic levels)
  - Optimization

Run-time environments

Before discussing code generation, we need to understand what we are trying to generate.

- We'll use the term *virtual machine* to refer to the compiler's target.
- Can be just a bare hardware architecture (small embedded systems).
- Can be an interpreter, as for Java, or an interpreter that does additional compilation at execution, as in modern Java JITs
- Can even be a "machine" whose machine language is another programming language such as C, Java, or Javascript.
- For now, we'll stick to hardware + conventions for using it (the *API: application programmer's interface*) + some *runtime-support library.*

Code Generation Goals and Considerations

- *Correctness:* execution of generated code must be consistent with the programs' specified dynamic semantics.
- In general, however, these semantics do not completely specify behavior, often to allow compiler to accomplish other goals, such as...
- *Speed:* produce code that executes as quickly as possible, or reliably meets certain timing constraints (as in real-time systems).
- *Size:* minimize size of generated program or of runtime data structures.
- Speed and size optimization can be conflicting goals. Why?
- *Compilation speed:* especially during development or when using JITs.
- Most complications in code generation come from trying to be fast as well as correct, because this requires attention to special cases.

Subgoals and Constraints

- Subgoals for improving speed and size:
  - Minimize instruction counts.
  - Keep data structure static, known at compilation (e.g., known constant offsets to fields). Contrast Java and Python.
  - Maximize use of registers ("top of the memory hierarchy").
- Subgoals for improving compilation speed:
  - Try to keep analyses as *local* as possible (single statement, block, procedure), because their compilation-time cost tends to be non-linear.
  - Simplify assumptions about control flow: procedure calls "always" return, statements generally execute in sequence. (Where are these violated?)

Example of a Virtual Machine: Java Interpreter

- The Java compiler (`javac`) converts Java programs (`.java`) into *class files* (`.class`).
- A Java interpreter, or a JIT (Just In Time) compiler then executes these.
- The class-file format is platform-independent, avoiding the problem of producing the different versions for different architectures (x86_64, ARM, RISC V, ...) and operating systems (MacOS, Linux, BSD Unix, Windows, Android, ...).
- The class file uses a rather high-level machine language (compared to, say, x86_64).
- Its abstraction is that of a *stack machine,* in which operations implicitly pop operands from and push results onto an execution stack.
- In addition, class file can carry information about types, as well as debugging informatiom such as line numbers and local variable names.

Java Class Example

[See accompanying files.]

Classfile Verification

- The Java interpreter does not assume that the class files you give it won't break anything.
- Instead, it *statically verifies* that each class file it is given conforms to type rules.
- So, for example, it won't allow this:

```
...
7: iinc    0, 1  // Increment args parameter (String[])
8: aload_0       // Get String[] args parameter.
9: arraylength
```

  which would nonsensically add an integer (1) to a pointer (the reference to String[]) and then try to take its length.

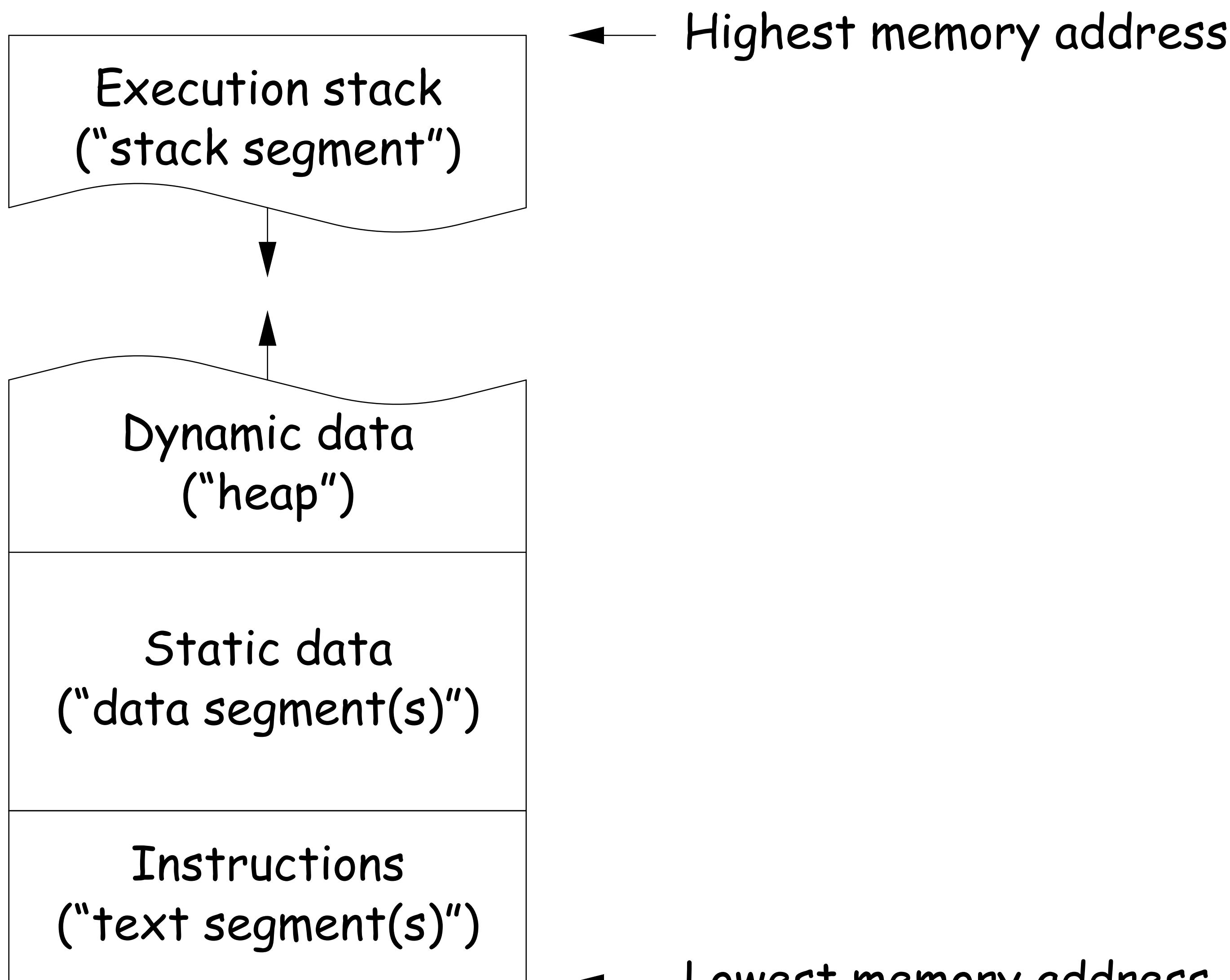
- We'll take a look at this kind of verification in a later unit.

Activations and Lifetimes (Extents)

- An invocation of procedure P is an *activation* of P.
- The *lifetime of an activation* of P is all the steps to execute P, including all the steps in procedures P calls.
- The *lifetime (extent) of a variable* is the portion of execution during which that variable exists (whether or not the code currently executing can reference it).
- Lifetime is a dynamic (run-time) concept, as opposed to scope, which is static.
- Lifetimes of procedure activations and local variables properly nest (in a single thread), suggesting a *stack* data structure for maintaining their runtime state.
- Other variables have extents that are not coordinated with procedure calls and returns.

Memory Layout

Characteristics of procedure activations and variables suggest an idealized layout for a process's data:



Memory Layout II

- However, reality is generally much more complicated.
- The `layout.txt` file in the accompanying lecture files shows a snapshot of the layout from a running Linux process.
- As you'll see, there is a great variety of information, much of it shared among processes.

Runtime Support for Functions

Bare Machine to Virtual Machine

- Typical architectures provide simple instructions to support subprograms (functions and procedures).
- Typically, we have some sort of "branch and link" instruction that branches to an instruction, and puts the address of the instruction after the branch itself—the *return address*—in some well-defined place.
- But there is more to subprogram calls than that, such as local variables, parameters, dealing with nested calls, etc.
- To deal with these other things, compilers generate code for, in effect, a virtual machine with a more elaborate call instruction.
- Explicit in the JVM's `invokevirtual` instruction.
- For conventional generation of machine code, use various programming conventions.

Activation Records

- The information needed to manage one procedure activation is called an *activation record (AR)* or *(stack) frame.*
- If procedure F (the *caller*) calls G (the *callee),* typically G's activation record contains a mix of data about F and G:
  - *Return address* to instructions in F.
  - *Dynamic link* to the AR for F.
  - Space to save registers needed by F.
  - Space for G's local variables.
  - Information needed to find non-local variables needed by G.
  - Temporary space for intermediate results, arguments to and return values from functions that G calls.
  - Assorted machine status needed to restore F's context (signal masks, floating-point unit parameters).
- Depending on architecture and compiler, registers typically hold part of AR (at times), especially parameters, return values, locals, and pointers to the current stack top and frame.

Calling Conventions

- Many variations are possible:
  - Can rearrange order of frame elements.
  - Can divide caller/callee responsibilities differently.
  - Don't need to use an array-like implementation of the stack: can use a linked list of ARs.
- An organization is better if it improves execution speed or simplifies code generation
- The compiler must determine, at compile-time, the layout of activation records and generate code that correctly accesses locations in the activation record.
- Furthermore, it is common to compile procedures separately and without access of each other's details, which motivates the imposition of *calling conventions.*

Static Storage

- Here, *static storage* refers to variables whose extent is an entire execution and whose size is typically fixed before execution.
- Not generally stored in an activation record, but assigned a fixed address once.
- In *C/C++* variables with file scope (declared `static` in *C*) and with external linkage ("global") are in static storage.
- Java's "static" variables are an odd case: they don't really fit this picture (why?)

Heap Storage

- Variables whose extent is greater than that of the AR in which they are created can't be kept there:

```
Bar foo() { return new Bar(); }
```

- Call such storage *dynamically allocated*.
- Typically allocated out of an area called the *heap* (confusingly, not the same as the heap used for priority queues!)

Achieving Runtime Effects—Functions

- Language design and runtime design interact. Semantics of functions make good example.
- Levels of function features:
  1. Plain: no recursion, no nesting, fixed-sized data with size known by compiler.
  2. Add recursion.
  3. Add variable-sized unboxed data.
  4. Allow nesting of functions, up-level addressing.
  5. Allow function values w/ properly nested accesses only.
  6. Allow general closures.
  7. Allow continuations.
- Tension between these effects and structure of machines:
  - Machine languages typically only make it easy to access things at addresses like $R + C$, where R is an address in a register and C is a relatively small integer constant.
  - Therefore, fixed offsets good, data-dependent offsets bad.

1: No recursion, no nesting, fixed-sized data

- Total amount of data is bounded, and there is only one instantiation of a function at a time.
- So all variables, return addresses, and return values can go in fixed locations.
- No stack needed at all.
- Characterized FORTRAN programs in the early days.
- In fact, can dispense with call instructions altogether: expand function calls in-line. E.g.,

```
def f (x):
    x *= 42
    y = 9 + x;
    g (x, y)

f (3)
```

$\Longrightarrow$ becomes $\Longrightarrow$

```
x_1 = 3
x_1 *= 42
y_1 = 9 + x_1
g (x_1, y_1)
```

- However, program may get bigger than you want. Typically, one in-lines only small, frequently executed functions.

1: Calling conventions

- If we don't use function inlining, will need to save return address, parameters.
- There are many options. Here's one example, from the IBM 360, of calling function F from G and passing values 3 and 4:

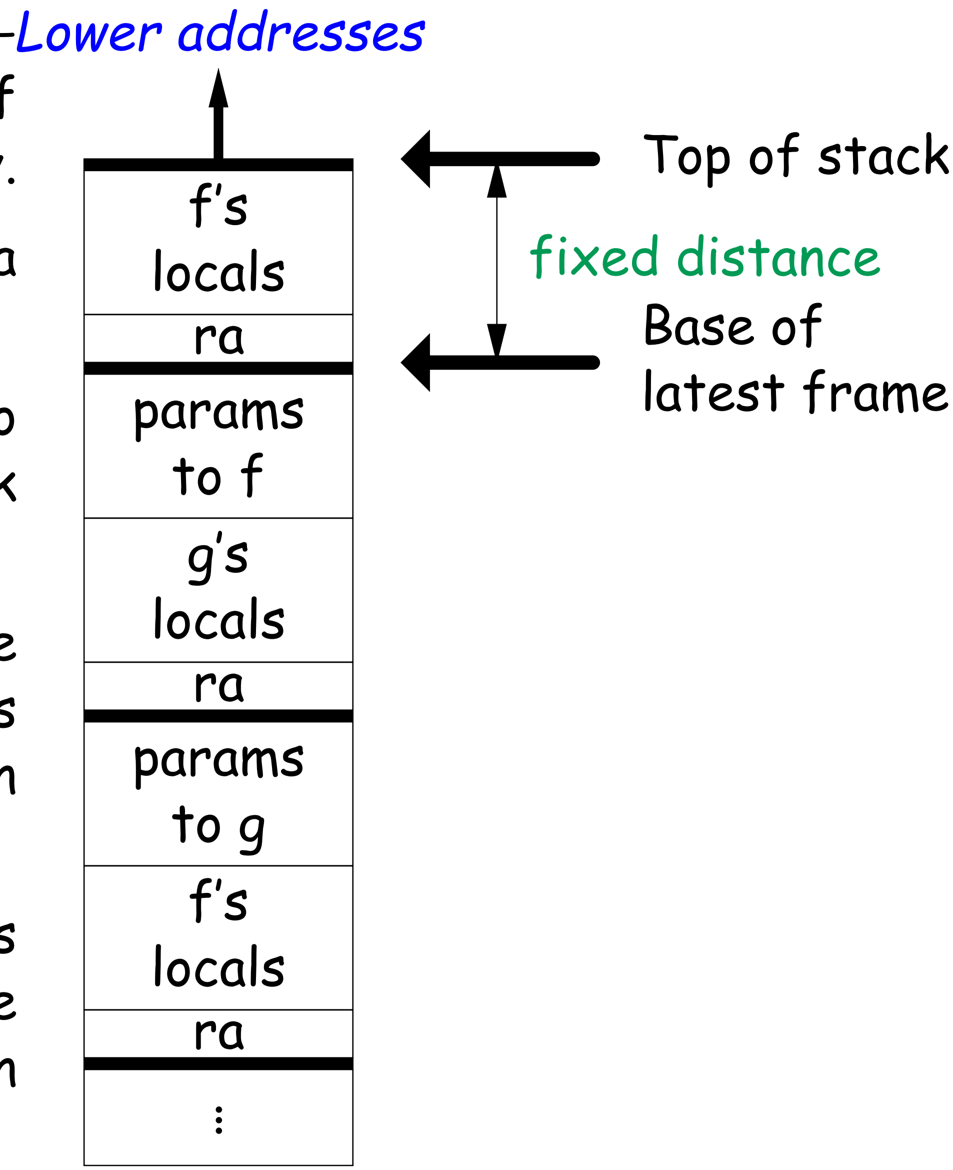
```
GArgs  DS    2F           Reserve 2 4-byte words of static storage */
       ...
       ENTRY G
G      ...
       LA    R1,GArgs     Load Address of arguments into register 1
       LA    R0,3         Store 3 and 4 in GArgs+0 and GArgs+4
       ST    R0,GArgs
       LA    R0,4
       ST    R0,GArgs+4
       BAL   R14,F        Call ("Branch and Link") to F, R14 gets return point
```

and F might contain

```
FRet   DS    F
       ENTRY F
F      ST    R14,FRet     Save return address
       L     R2,0(R1)     Load first argument.
       ...
       L     R14,FRet     Get return address
       BR    R14          Branch to it
```

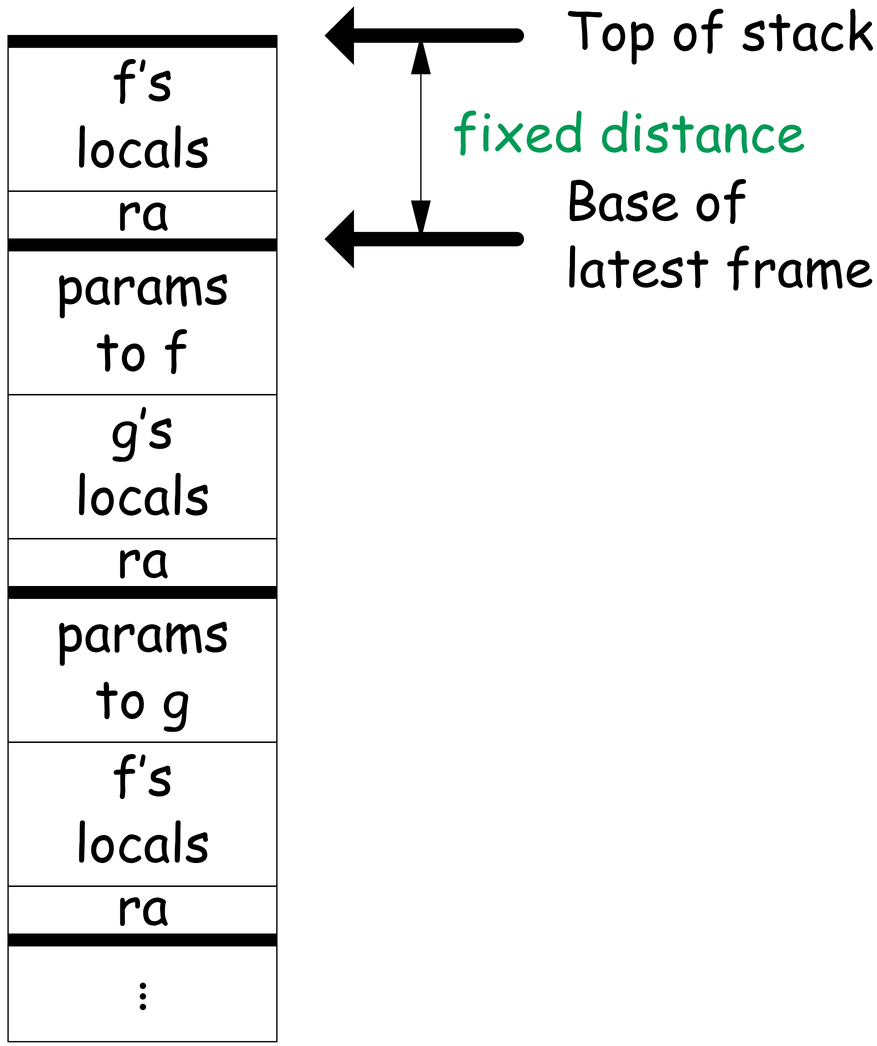
2: Add recursion

- Now, total amount of data is unbounded, and several instantiations of a function can be active simultaneously.
- Calls for some kind of expandable data structure: a stack.
- However, variable sizes still fixed, so size of each activation record (stack frame) is fixed.
- All local-variable addresses and the value of dynamic link are known offsets from stack pointer, which is typically in a register.
- (The diagram shows the conventions we'll use in Project 3, where we'll define a stack frame as starting at the return address or dynamic link.)



2: Calling Sequence when Frame Size is Fixed

- So dynamic links not really needed.
- Suppose f calls g calls f, as at right.
- When called, the initial code of g (its *prologue*) decrements the stack pointer by the size of g's activation record.
- g's exit code (its *epilogue*):
  - increments the stack pointer by this same size,
  - pops off the return address, and
  - branches to address just popped.



2: Possible calling sequence for Risc V

C code:

```
int
dist2(int x, int y)
{
  return x**2 + y**2;
}

int
g(int q)
{
  return dist2(q, 5);
}
```

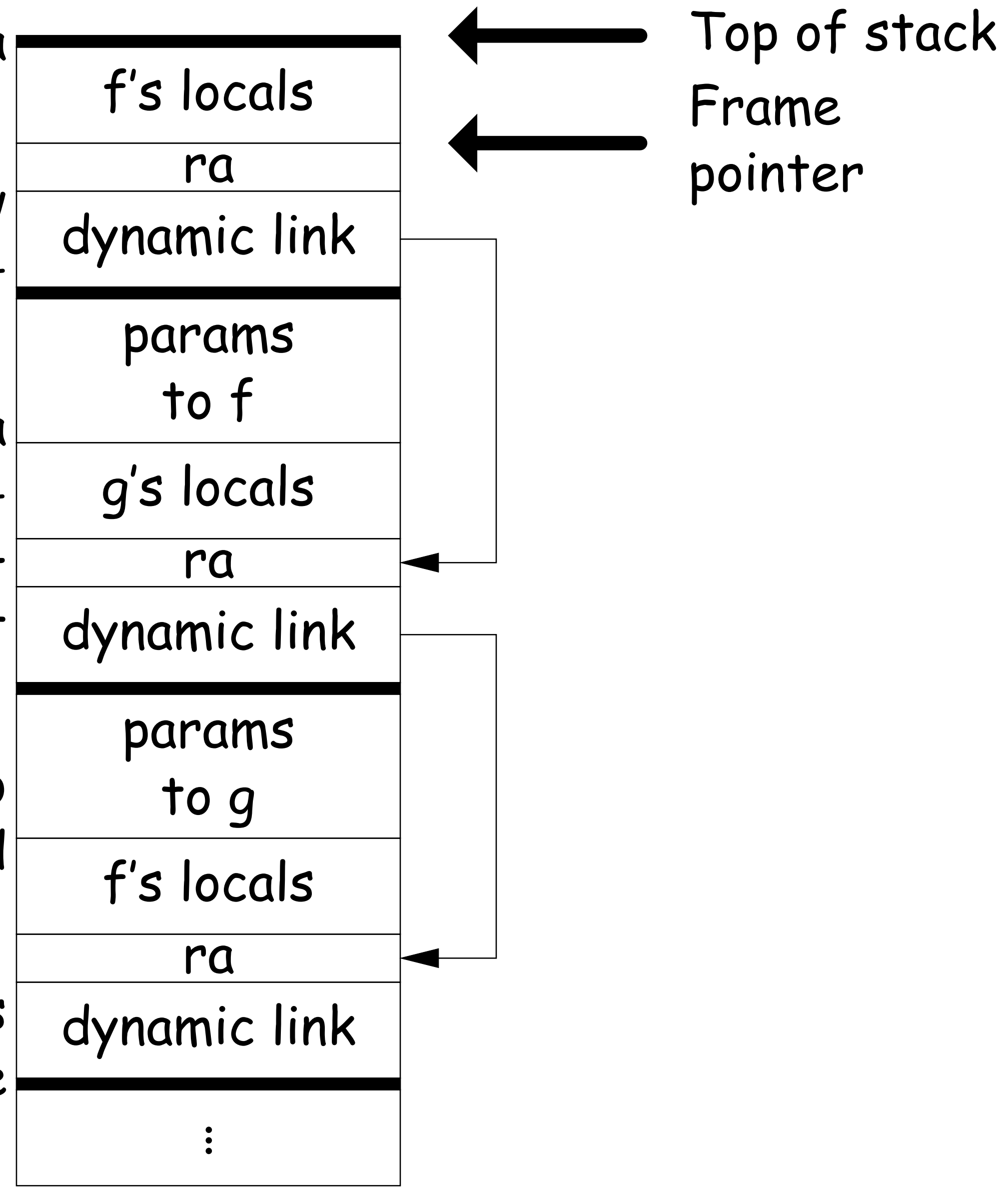
Assembly excerpt:

```
dist2:  # Leaf procedure (no need to save ra)
  lw t0, 8(sp)    # x
  mul t0, t0, t0  # x*x
  lw t1, 4(sp)    # y
  mul t1, t1, t1  # y*y
  add a0, t0, t1  # x*x+y*y
  jr ra

g: # Non-leaf procedure
  sw ra, 0(sp)    # Save return address
  addi sp, sp, -4 # Adjust SP
  lw t0, 8(sp)    # q
  sw t0, 0(sp)    # Argument 1
  li t0, 5
  sw t0, -4(sp)   # Argument 2
  addi sp, sp, -8 # Put SP below params
  jal dist2       # Call
  addi sp, sp, 8  # Return SP to pre-dist2 call
  lw ra, 4(sp)    # Retrieve return address
  addi sp, sp, 4  # Return SP to pre-g call
  jr ra
```

2: Frame pointers

- In the previous example, took all data relative to a (varying) stack pointer.
- The compiler "knows" at each point how to restore the stack pointer before return (fixed-size adjustments).
- Sometimes, it is convenient to have a pointer to a fixed location in the activation record—called a *frame pointer*—that the callee (called function) must set and restore.
- For one thing, this makes it easier to write general procedures that unwind the stack.
- Frame pointer in register. Previous value must be saved by each callee (the *dynamic link* or *control link*.)



2: Alternative Calling Sequence with Frame Pointer

C code:

```
int
dist2(int x, int y)
{
  return x**2 + y**2;
}

int
g(int q)
{
  return dist2(q, 5);
}
```

```
dist2:  # Leaf procedure (as before)
    lw t0, 8(sp)    # x
    mul t0, t0, t0  # x*x
    lw t1, 4(sp)    # y
    mul t1, t1, t1  # y*y
    add a0, t0, t1  # x*x+y*y
    jr ra

g: # Non-leaf procedure (use fp, save ra, old fp---DL).
    sw fp, 0(sp)    # Save old frame pointer
    sw ra, -4(sp)   # Save return address
    addi sp, sp, -8 # Adjust SP to allocate frame
    addi fp, sp, 4  # fp now points to saved return address
    lw t0, 8(fp)    # q
    sw t0, 0(sp)    # Argument 1
    li t0, 5
    sw t0, -4(sp)   # Argument 2
    addi sp, sp, -8 # Put SP below params
    jal dist2       # Call
    addi sp, sp, 8  # Return SP to pre-dist2 call
    lw ra, 0(fp)    # Get saved ra.
    addi sp, fp, 4  # Return sp to pre-g call
    lw fp, 4(fp)    # Return fp to pre-g call
    jr ra
```

3: Add Variable-Sized Unboxed Data

- "Unboxed" means "not on heap."
- Boxing allows all quantities on stack to have fixed size.
- So Java implementations have fixed-size stack frames.
- But does cost heap allocation, so some languages also provide for placing variable-sized data directly on stack ("heap allocation on the stack")
- `alloca` in C, e.g.
- Now we do need dynamic link (DL).
- But can still insure fixed offsets of data from frame base (*frame pointer*) using pointers.
- To right, f calls g, which has variable-sized unboxed array (see right).