Checkin 27

Show the layout of an activation record with two 64-bit locals. Write the function prologue and epilogue corresponding to that function

Announcements

Administrivia

P6 is out

Q3 imminent!

University of Kansas | Drew Davidson

Statement Code Generation

Last Lecture

Activation Records

Managing the Stack

- Managing data
- Managing control

<u>You Should Know</u>

How to code up stack frames
The function prologue
The function epilogue

Architecture

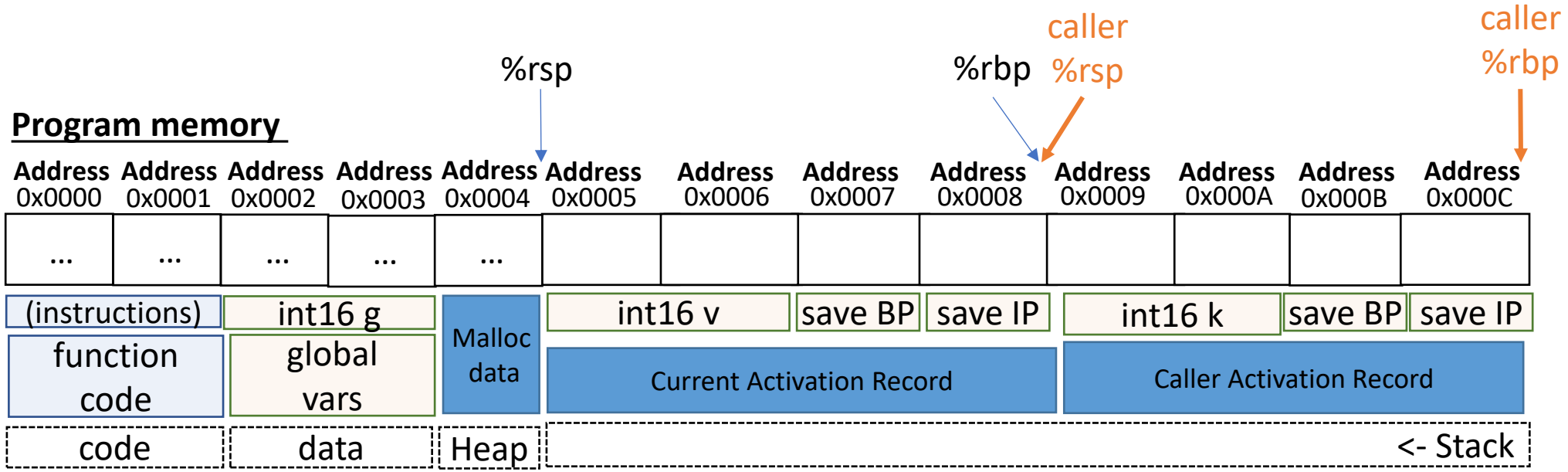
Call Stack Bookkeeping

Review: Stack Frames

We need to store (on the stack):

- The call site to resume execution after call
- The base pointer to restore the old stack frame after call

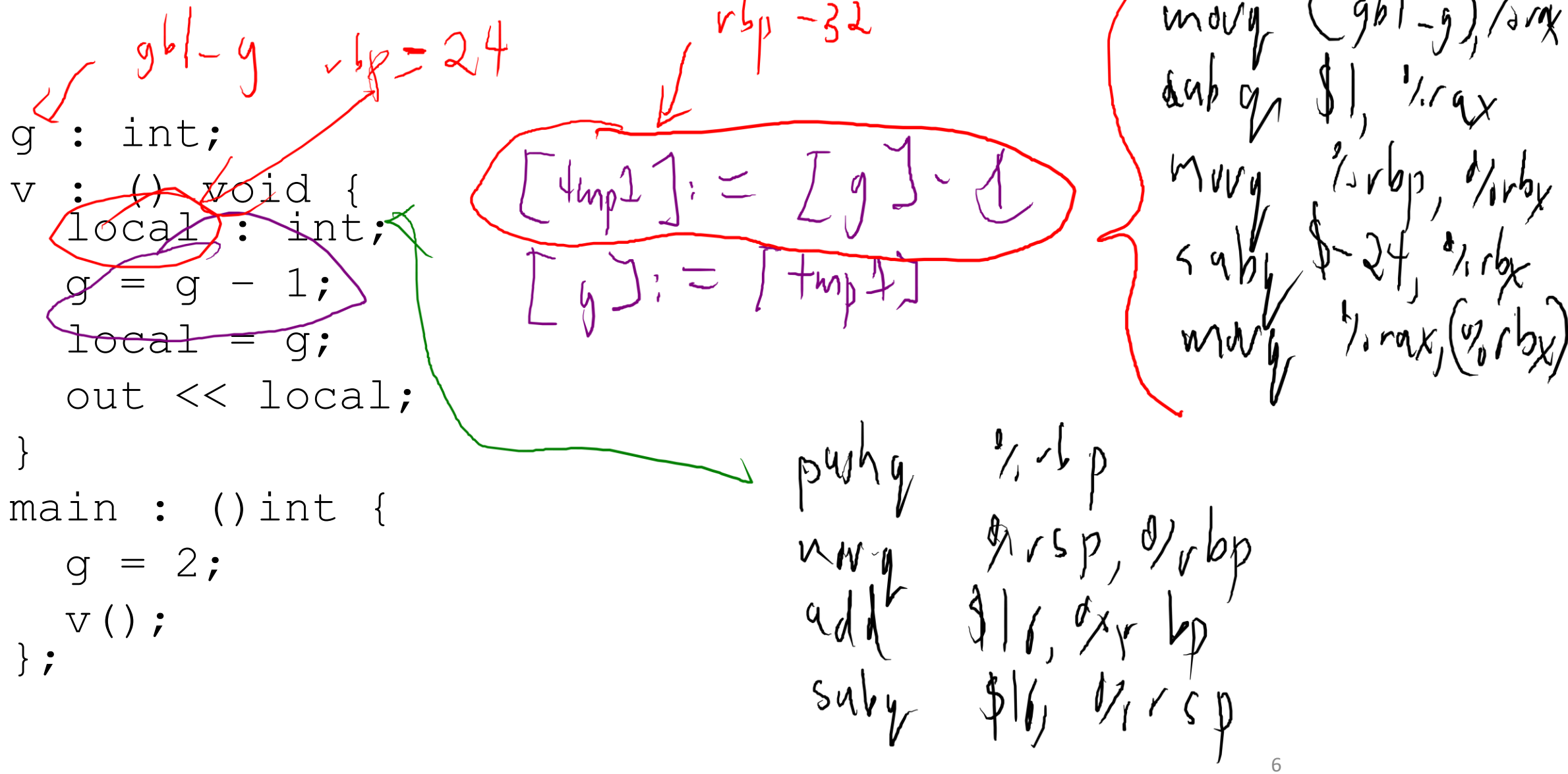
bookkeeping space at the beginning of the AR



A Less-Trivial x64 Program

Review: Stack Frames

```
g : int;
v : () void {
  local : int;
  g = g - 1;
  local = g;
  out << local;
}
main : ()int {
  g = 2;
  v();
};
```



Addressing modes

Toward Local Variables

Some Nice "Shortcuts"

- Often want to read memory at a fixed offset from some register

  "the memory at 8 bytes before %rbp"

- Good news! X64 can do that:

```
movq -8(%rbp), %rax
```

- This is a very handy addressing mode

```
leaq -8(%rbp), %rax
```

"Move the value AT %rbp – 8 into %rax" =

```
movq %rbp, %rdx
subq $8, %rdx
movq (%rdx), %rax
```

"Move the value OF %rbp – 8 into %rax" =

```
movq %rbp, %rdx
subq $8, %rdx
movq %rdx, %rax
```

Where We're At

Progress Pics

Assembled quite a few x64 concepts

- Basic data manipulation (movq)
- Basic math (addq, idivq, etc)
- Global data (.data, .quad, .byte)
- Local data
- Function calls

You can now hand-code some non-trivial programs


COMPILER
Code Generation
Optimization
Architecture
Runtime Environments
Semantic Analysis
Syntactic Definition
Parsing
Syntax-Dir Translation
Regular Languages
Lexical Analysis
LAND
GLOPPY
CANDY CASTLE
KING KANDY
QUEEN FROSTINE
GRAMMA NUTT
PRINCESS LOLLY
LORD LICORICE
LICORICE CASTLE
JOLLY
PLUMPY

Lecture Outline

Statement Code Generation

From Quads to Assembly

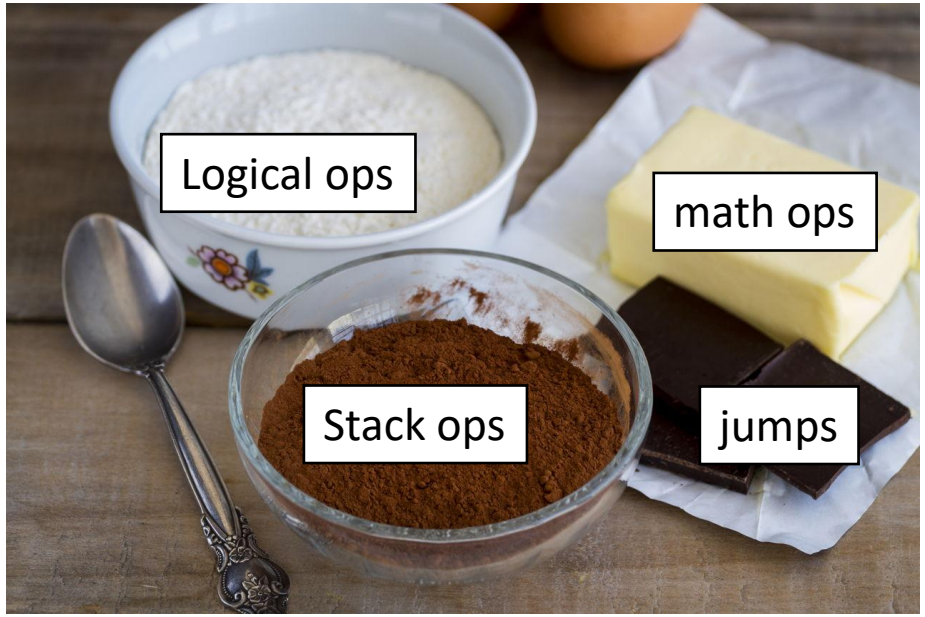
- Approach Overview
- Planning out memory
- Writing out x64

Code generation

Representing Abstract Constructs

Statement Code Generation

Combine (simple) target language constructs…



…to build (complex) source language constructs

Our Approach: Small Steps

Code Generation

2 passes over IRProgram (like passes over AST)

1. Allocate memory for opds (data pass)
2. Generate code for quads (code pass)

Code Generation Objectives

Designing Code Generators

- Two high level goals:
  - Generate correct code ⟸ Top priority
  - Generate efficient code

- It can be difficult to achieve both at once
  - Efficient code can be harder to understand
  - Efficient code may have unanticipated side effects

Our Approach: Small Steps

Code Generation

2 passes over IRProgram (like passes over AST)

1. Allocate memory for opds (data pass)
2. Generate code for quads (code pass)

Preparing the 3AC memory layout

Variable Allocation

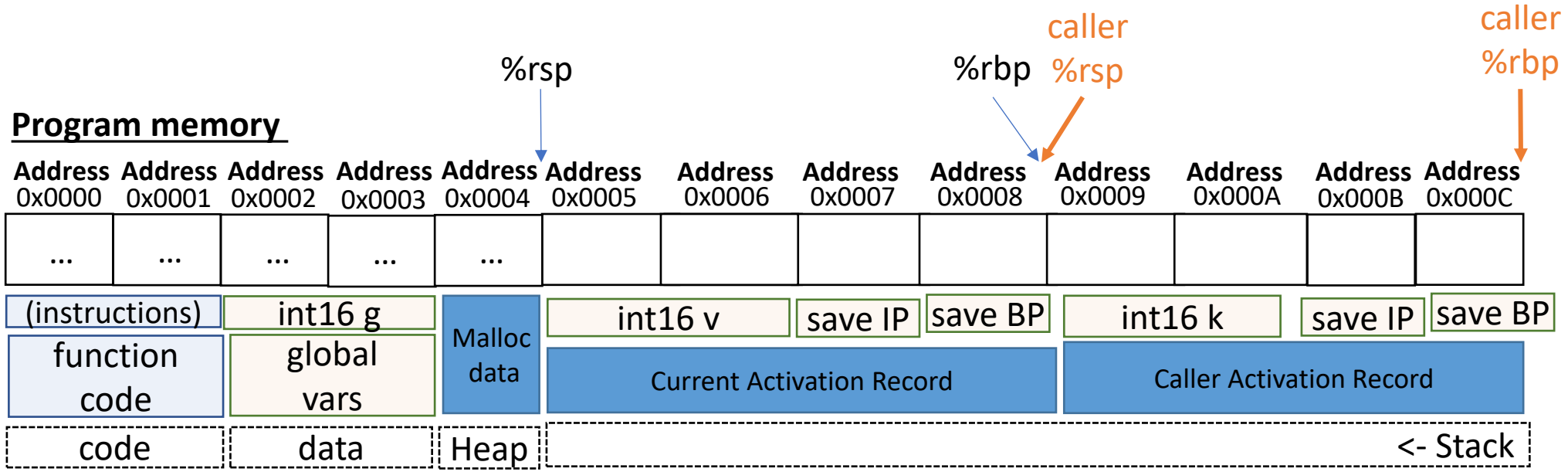
Code Generation

Big picture:

- Every variable needs space in enough space in memory for its type
- Every quad using that variable needs to access the same location

Need a mix of static/dynamic allocation

- Put globals/strings at fixed addresses in memory (absolute locations)
- Put locals/formals at stack offsets in memory (relative locations)



Allocation: In Code (suggestion)

Code Generation

Add a location field (std::string) to semantic symbols

- All related SymOpds have pointers to the same symbol

Location can be a string

- For globals, the label that you'll write
- For locals, the stack offset you'll arrange

Variable Allocation: Globals

Code Generation

3AC Code

```
[g] := 4
```

location: At label gbl_g

Where g is a global int

X64 Code

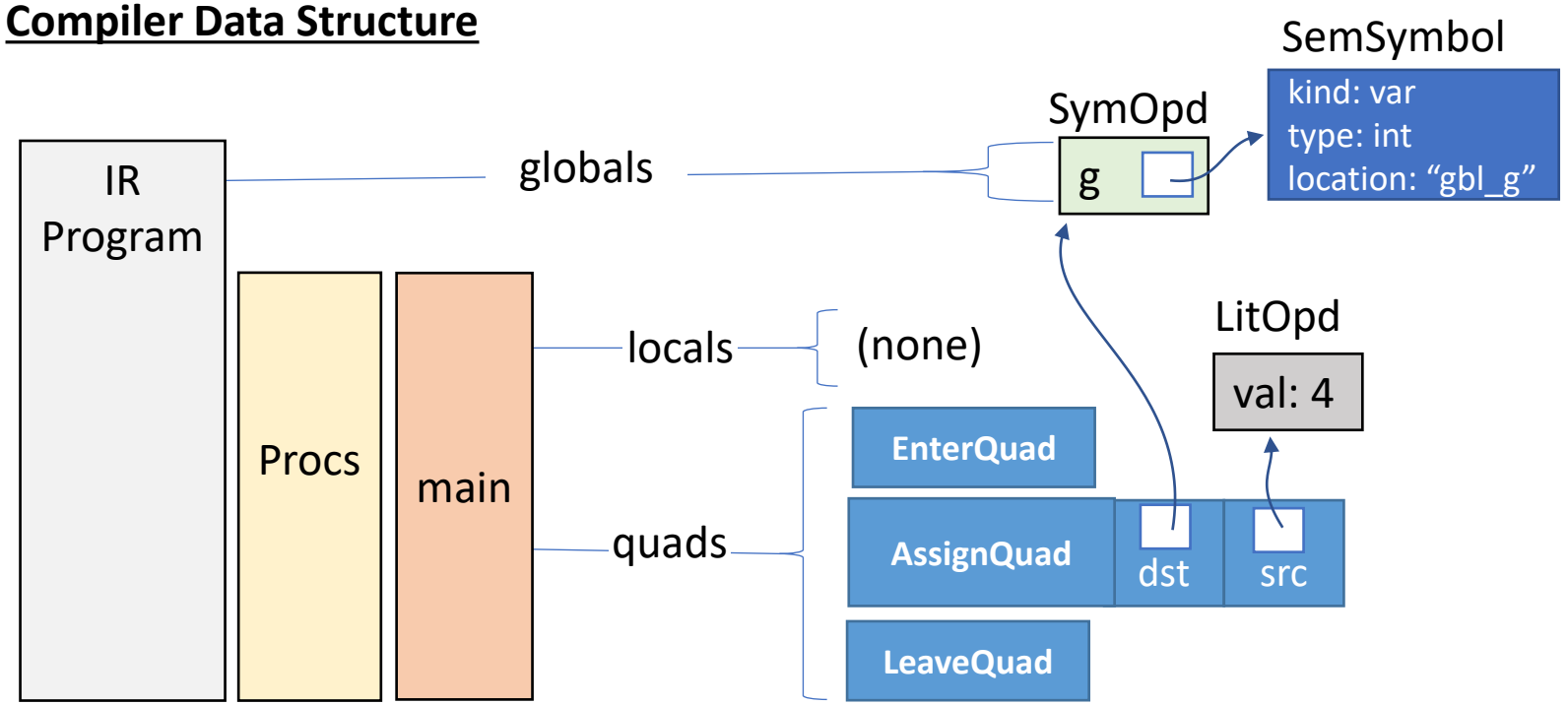
… in .data section …

```
gbl_g:      .quad 0
```

… somewhere in .text section …

```
movq $4, (gbl_g)
```

Compiler Data Structure



Variable Allocation: Locals

Code Generation

3AC Code

```
[v] := 7
```

Where v is a local int

location:
At offset
-24(%rbp)

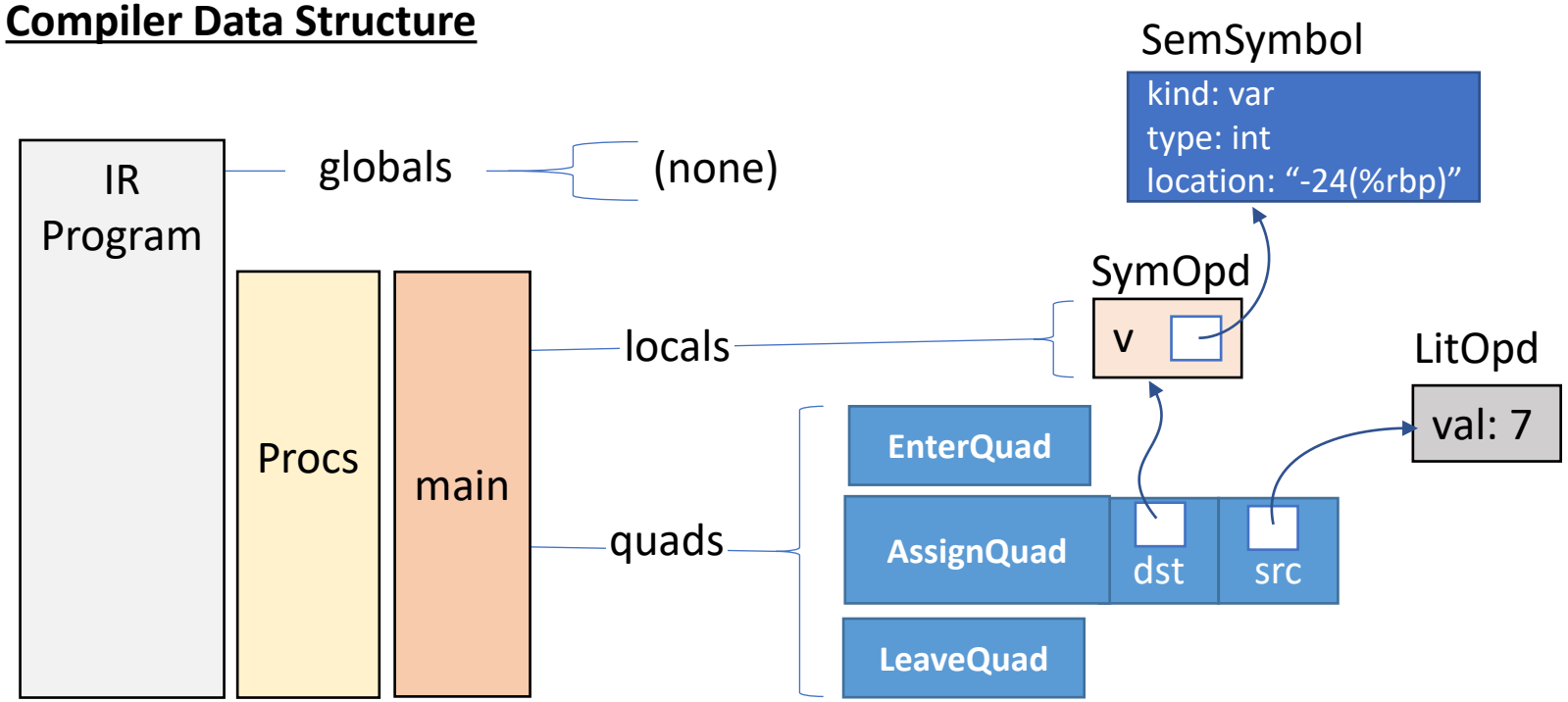
X64 Code

… assume stack frame setup …

… somewhere in main's asm …

```
movq $7, -24(%rbp)
```

Compiler Data Structure



Our Approach: Small Steps

Code Generation

2 passes over IRProgram (like passes over AST)

1. Allocate memory for opds (data pass)
2. Generate code for quads (code pass)

Write the assembly file

Assembly Directives/Initialization

Code Generation

We're gonna write the whole file in one shot

- Aided greatly by our preparatory layout pass
- Also aided by the assembler

Write out .data section:

```
.data
.globl: main
<global1_label> : <global1_type> <global1_val>
…
<global1_label> : <global1_type> <global1_val>
```

Walk each 3AC Procedure, output each quad

```
enter main
```

Generating Code for Quads

Code Generation

Generating Code for Quads

Code Generation

enter <proc>

leave <proc>

<opd> := <opd>

<opd> := <opr> <opd>

<opd> := <opd> <opr> <opd>

<lbl>: <INSTR>

ifz <opd> goto <lbl>

goto Li

nop

call <name>

setin <int> <operand>

getin <int> <operand>

setout <int> <operand>

getout <int> <operand>

Generating Code for Quads: enter/leave

Code Generation

On entry to the function:

- Set up the activation record

On exit from the function

- Break down the activation record

Make function prologue/epilogue

enter <proc>

leave <proc>

Prologue

```
pushq %rbp
movq %rsp, %rbp
addq $16, %rbp
subq $X, %rsp
```

Epilogue

```
addq $X, %rsp
popq %rbp
retq
```

Generating Code for Quads: enter/leave

Code Generation

src code	**3ac code**	**asm code**

int main(){
}
	enter main
	leave main
		lbl_main: pushq %rbp
		movq %rsp, %rbp
		addq $16, %rbp
		subq $0, %rsp
		addq $0, %rsp
		pushq %rbp
		retq

enter <proc>

leave <proc>

Prologue	**Epilogue**

pushq %rbp
movq %rsp, %rbp
addq $16, %rbp
subq $X, %rsp
	addq $X, %rsp
	popq %rbp
	retq

Generating Code for Quads

Code Generation

- enter <proc>
- leave <proc>

<opd> := <opd>

<opd> := <opr> <opd>

<opd> := <opd> <opr> <opd>

<lbl>: <INSTR>

ifz <opd> goto <lbl>

goto Li

nop

call <name>

setin <int> <operand>

getin <int> <operand>

setout <int> <operand>

getout <int> <operand>

For assignment-style quads…

1) Load operand src locations into registers

2) Compute a value to register

3) Store result at dst location

Assignment-Style Quads

Code Generation

3AC

[a] := [b] + 4

SymOpd
Symbol location: "gbl_a"

SymOpd
Symbol location: "-24(%rbp)

ASM

1) `movq -24(%rbp), %rax`
1) `movq $4, %rbx`
2) `addq %rbx %rax`
3) `movq %rax (gbl_a)`

For assignment-style quads…

1) Load operand src locations into registers
2) Compute a value to register
3) Store result at dst location

Questions?

Code Generation