6.035

Unoptimized Code Generation

- Last time we left off on the procedure abstraction …

The Stack

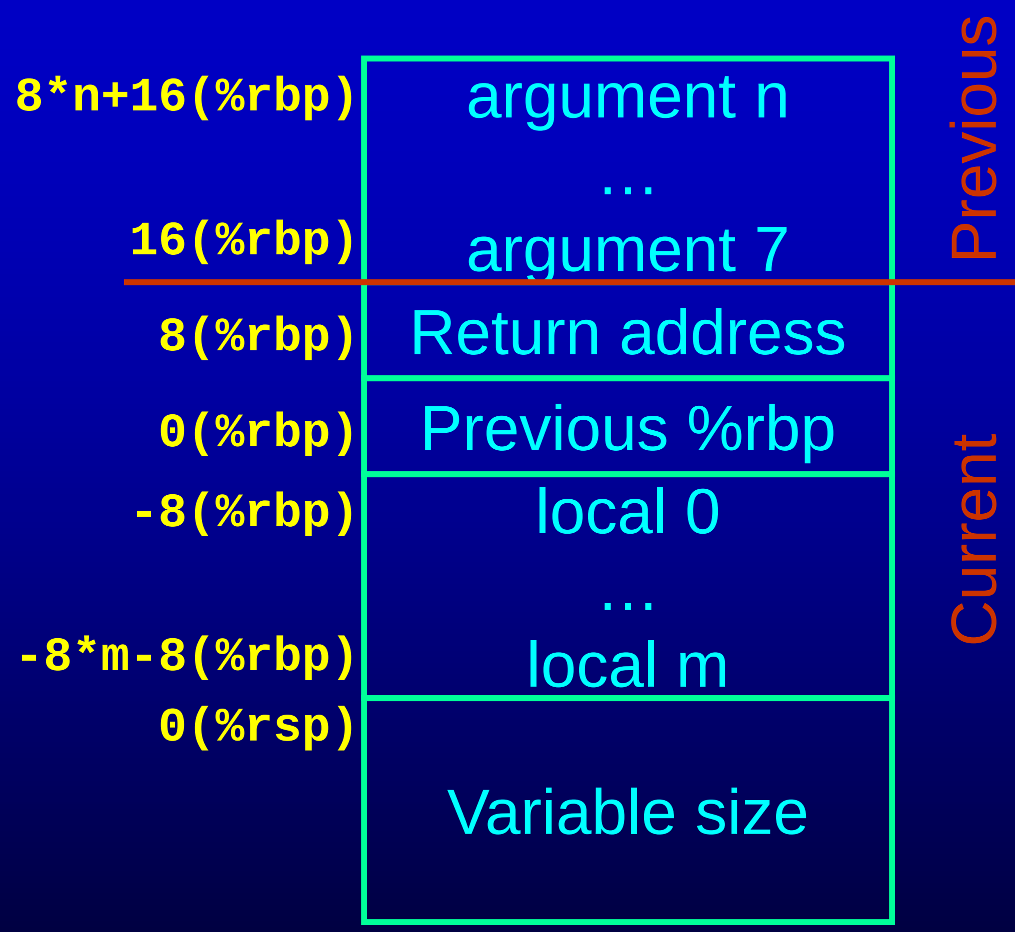
- Arguments 0 to 6 are in:
  - %rdi, %rsi, %rdx, %rcx, %r8 and %r9

%rbp

- marks the beginning of the current frame

%rsp

- marks the end

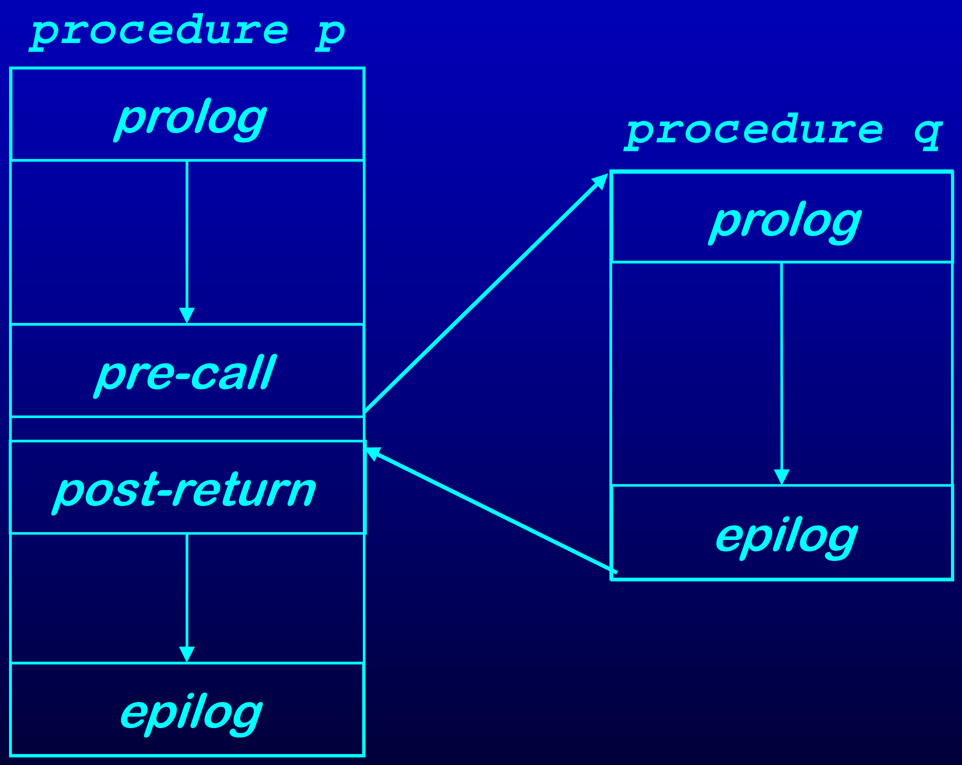


Question:

- Why use a stack? Why not use the heap or pre-allocated in the data segment?

Procedure Linkages

Standard procedure linkage



Pre-call:

- Save caller-saved registers
- Push arguments

Prolog:

- Push old frame pointer
- Save calle-saved registers
- Make room for temporaries

Epilog:

- Restore callee-saved
- Pop old frame pointer
- Store return value

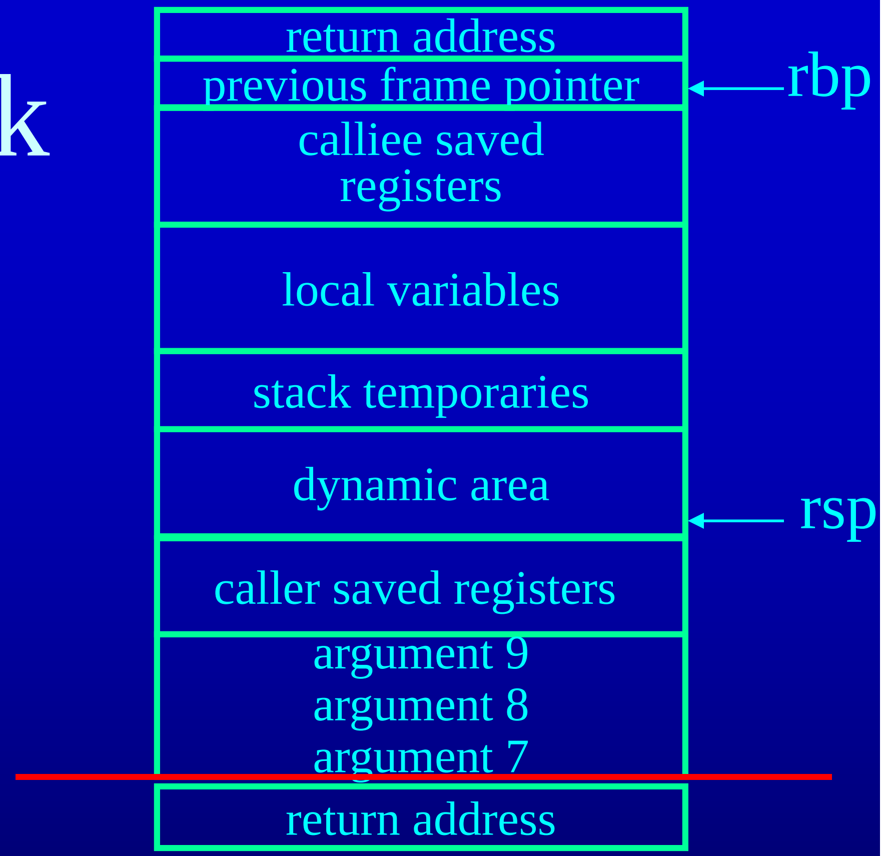
Post-return:

- Restore caller-saved
- Pop arguments

Stack

- Calling: Caller
  - Assume %rcx is live and is caller save
  - Call foo(A, B, C, D, E, F, G, H, I)
    - A to I are at -8(%rbp) to -72(%rbp)

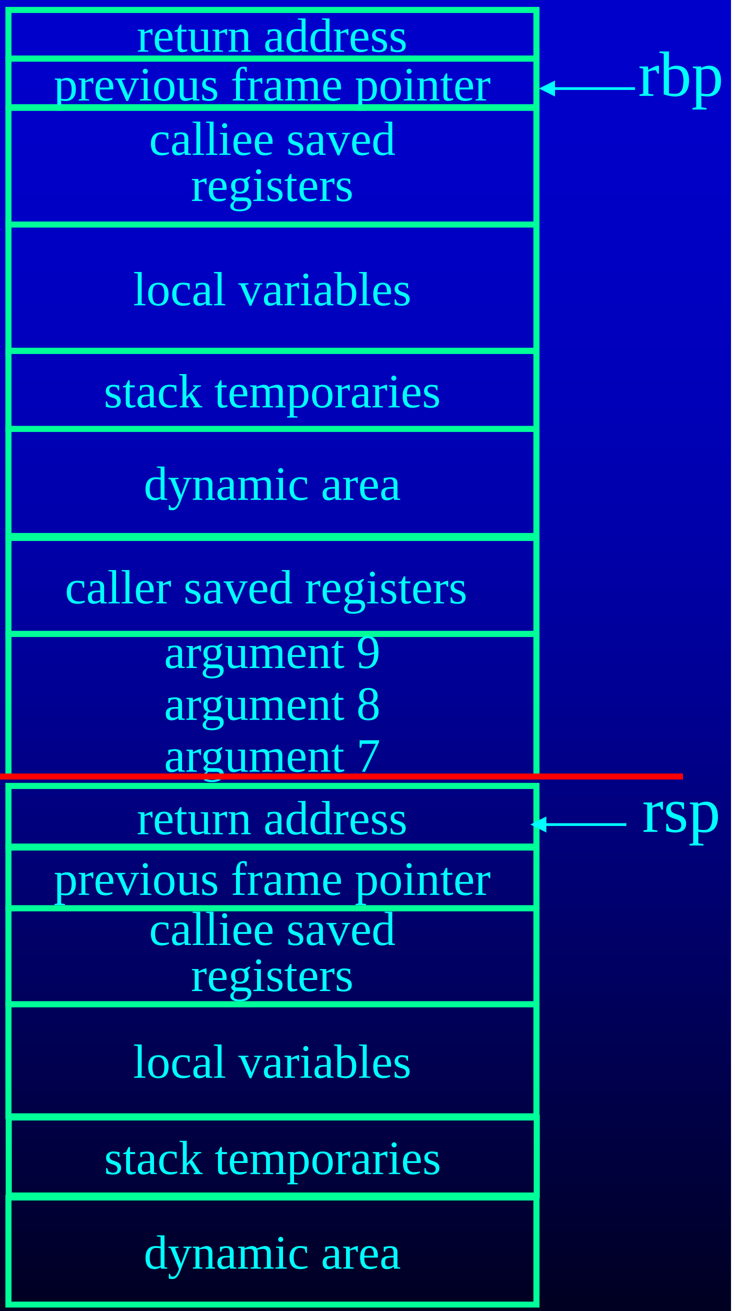
```
push    %rcx
push    -72(%rbp)
push    -64(%rbp)
push    -56(%rbp)
mov      -48(%rbp), %r9
mov      -40(%rbp), %r8
mov      -32(%rbp), %rcx
mov      -24(%rbp), %rdx
mov      -16(%rbp), %rsi
mov      -8(%rbp), %rdi
call    foo
```



Stack

- Calling: Calliee
  - Assume %rbx is used in the function and is calliee save
  - Assume 40 bytes are required for locals

```
foo:
    push    %rbp
    mov     %rsp, %rbp
    enter   $48, $0
    sub     $48, %rsp
    mov     %rbx, -8(%rbp)
```



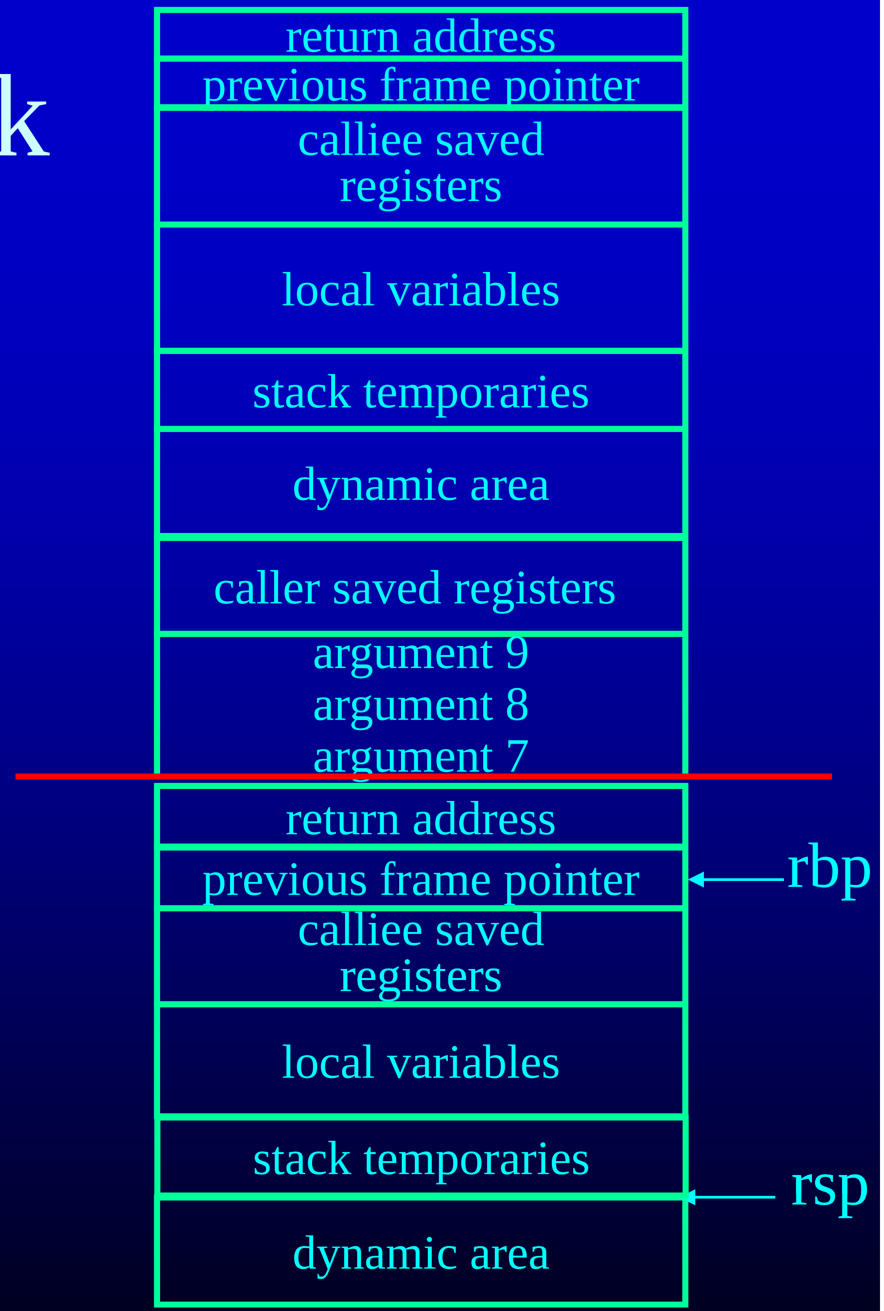
Stack

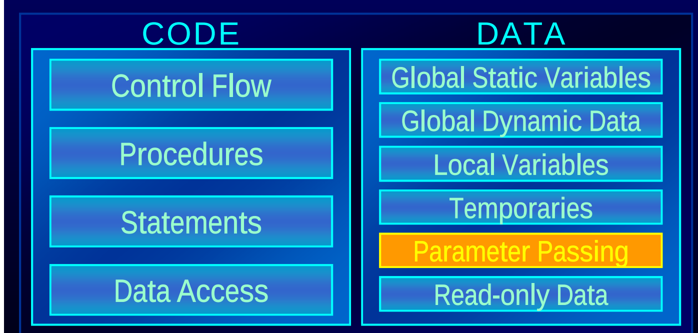
- Arguments
- Call foo(A, B, C, D, E, F, G, H, I)
  - Passed in by pushing before the call

```
push    -72(%rbp)
push    -64(%rbp)
push    -56(%rbp)
mov     -48(%rbp), %r9
mov     -40(%rbp), %r8
mov     -32(%rbp), %rcx
mov     -24(%rbp), %rdx
mov     -16(%rbp), %rsi
mov     -8(%rbp), %rdi
call    foo
```

  - Access A to F via registers
    - or put them in local memory
  - Access rest using 16+xx(%rbp)

```
mov     16(%rbp), %rax
mov     24(%rbp), %r10
```





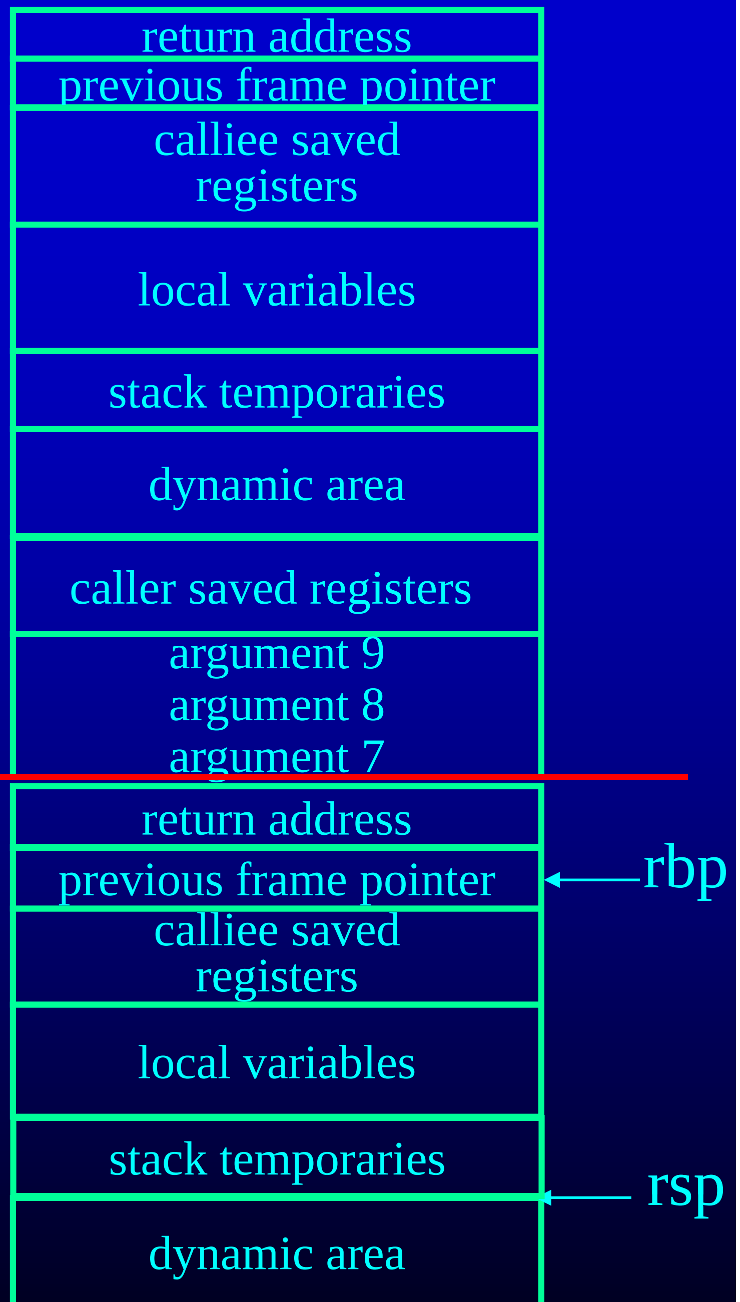
Stack

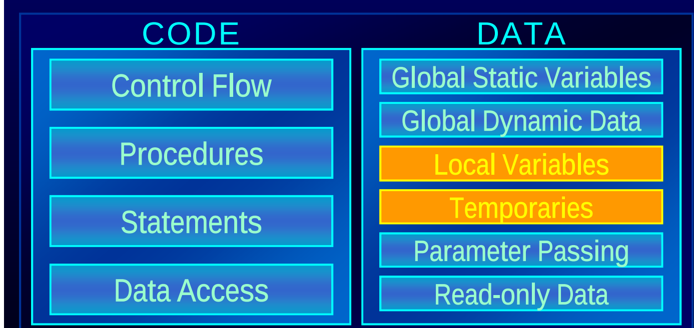
- Locals and Temporaries
  - Calculate the size and allocate space on the stack

```
    sub      $48, %rsp
or  enter    $48, 0
```

  - Access using -8-xx(%rbp)

```
    mov      -28(%rbp), %r10
    mov      %r11, -20(%rbp)
```

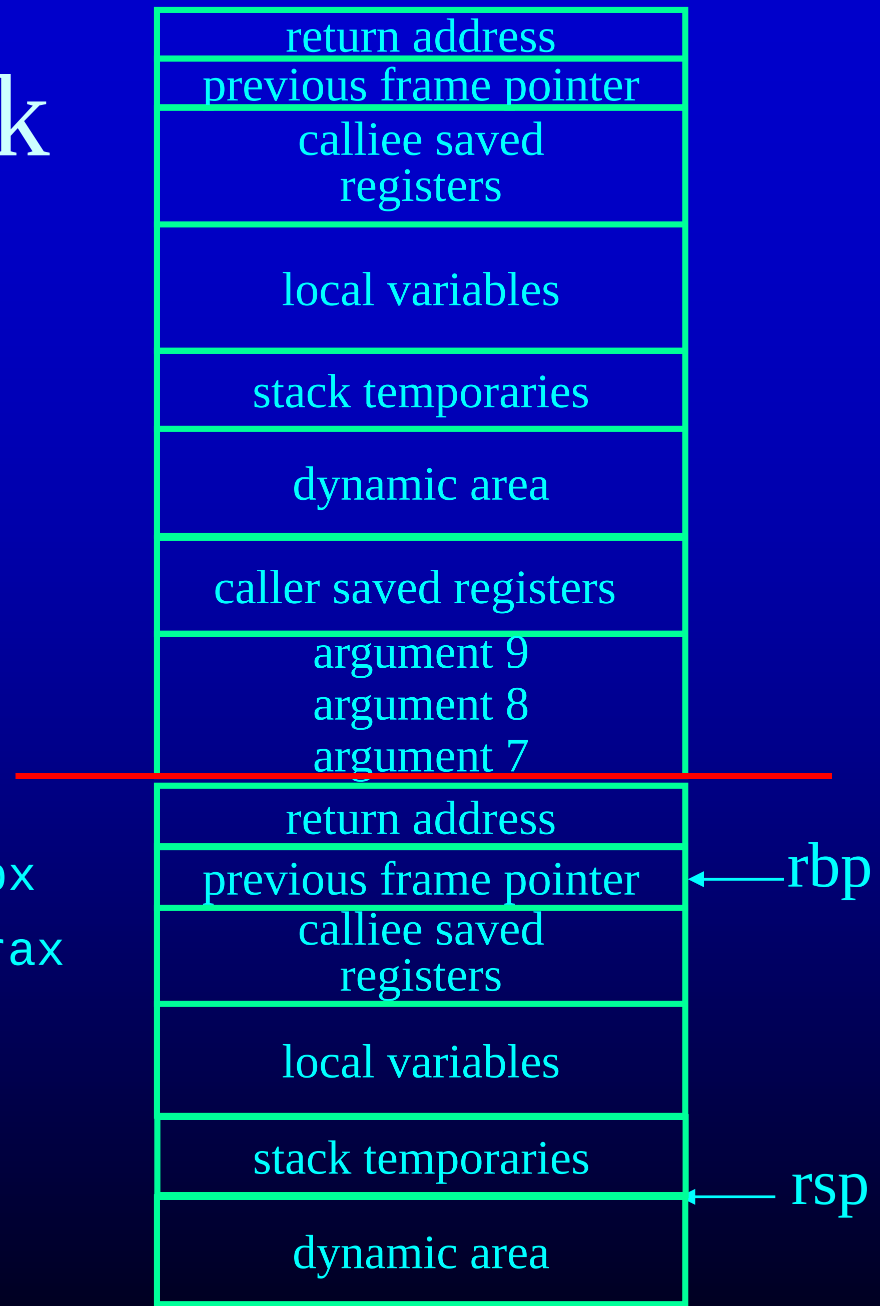




Stack

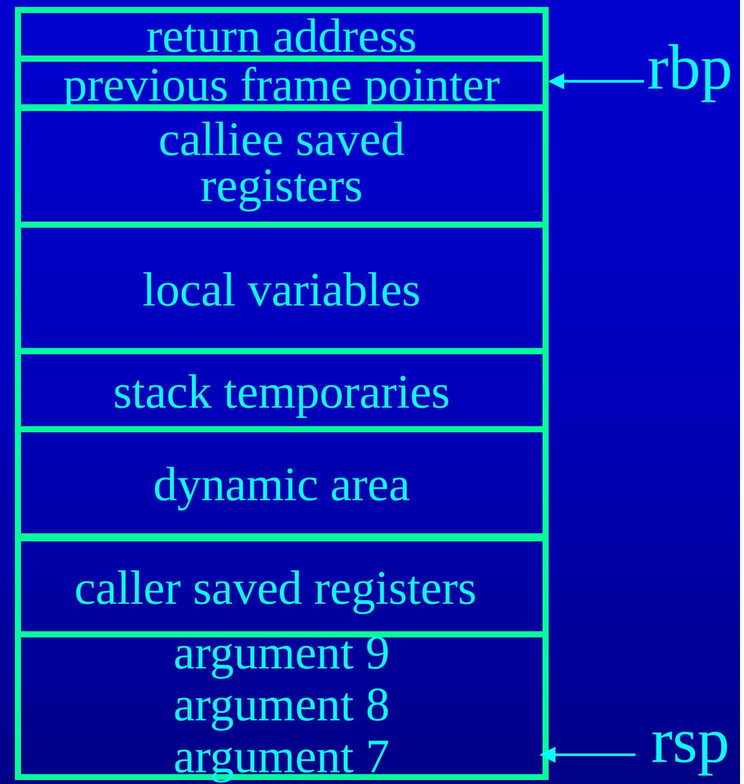
- Returning Calliee
  - Assume the return value is the first temporary
  - Restore the caller saved register
  - Put the return value in %rax
  - Tear-down the call stack

```
mov     -8(%rbp), %rbx
mov     -16(%rbp), %rax
mov     %rbp, %rsp
pop     %rbp
leave
ret
```

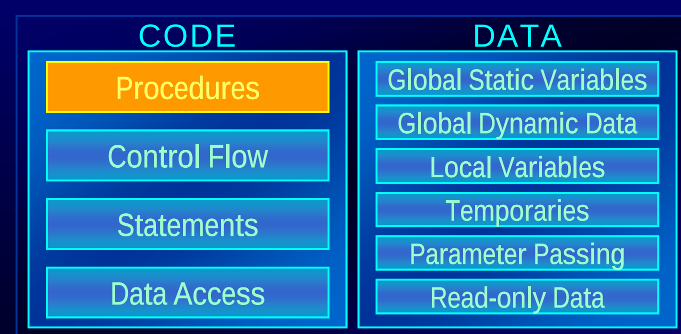


Stack

- Returning Caller
- (Assume the return value goes to the first temporary)
  - Restore the stack to reclaim the argument space
  - Restore the caller save registers
  - Save the return value




```
call    foo
add     $24, %rsp
pop     %rcx
mov     %rax, 8(%rbp)
…
```



Question:

- Do you need the \$rbp?
- What are the advantages and disadvantages of having \$rbp?

So far we covered..

CODE	DATA
Procedures	Global Static Variables
Control Flow	Global Dynamic Data
Statements	Local Variables
Data Access	Temporaries
	Parameter Passing
	Read-only Data

Outline

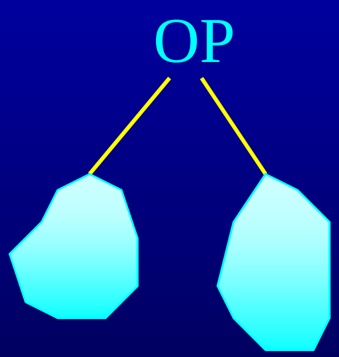
- Generation of expressions and statements
- Generation of control flow
- x86-64 Processor
- Guidelines in writing a code generator

Expressions

- Expressions are represented as trees
  - Expression may produce a value
  - Or, it may set the condition codes (boolean exprs)
- How do you map expression trees to the machines?
  - How to arrange the evaluation order?
  - Where to keep the intermediate values?
- Two approaches
  - Stack Model
  - Flat List Model

Evaluating expression trees

- Stack model
  - Eval left-sub-tree
    Put the results on the stack
  - Eval right-sub-tree
    Put the results on the stack
  - Get top two values from the stack
    perform the operation OP
    put the results on the stack
- Very inefficient!



Evaluating expression trees

- Flat List Model
  - The idea is to linearize the expression tree
  - Left to Right Depth-First Traversal of the expression tree
    - Allocate temporaries for intermediates (all the nodes of the tree)
      - New temporary for each intermediate
      - All the temporaries on the stack (for now)
  - Each expression is a single 3-addr op
    - x = y op z
    - Code generation for the 3-addr expression
      - Load y into register %r10
      - Load z into register %r11
      - Perform `op %r10, %r11`
      - Store %r11 to x

Issues in Lowering Expressions

- Map intermediates to registers?
  - registers are limited
    - when the tree is large, registers may be insufficient ⇒ allocate space in the stack
- No machine instruction is available
  - May need to expand the intermediate operation into multiple machine ops.
- Very inefficient
  - too many copies
  - don’t worry, we’ll take care of them in the optimization passes
  - keep the code generator very simple

What about statements?

- Assignment statements are simple
  - Generate code for RHS expression
  - Store the resulting value to the LHS address

- But what about conditionals and loops?

Outline

- Generation of statements
- Generation of control flow
- Guidelines in writing a code generator

Two Approaches

- Template Matching Approach
  - Peephole Optimization
- Algorithmic Approach
- Both are based on structural induction
  - Generate a representation for the sub-parts
  - Combine them into a representation for the whole

Generation of control flow: Template Matching Approach

- Flatten the control structure
  - use a template
- Put unique labels for control join points
- Now generate the appropriate code

Template for conditionals

```
if (test)
  true_body
else
  false_body
```

```
        <do the test>
        joper lab_true
        <false_body>
        jmp   lab_end
lab_true:
        <true_body>
lab_end:
```

Example Program

```
if(ax > bx)
        dx = ax - bx;
else
        dx = bx - ax;
```

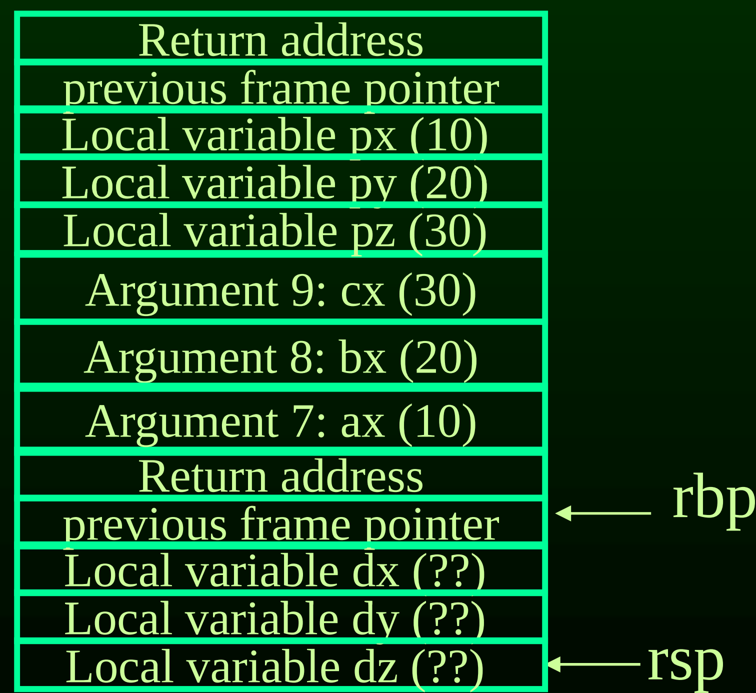
```
        <do test>

        joper  .L0
-----------------------------------
        <FALSE BODY>

        jmp     .L1
.L0:
-----------------------------------
        <TRUE BODY>

.L1:
```

Return address	
previous frame pointer	
Local variable px (10)	
Local variable py (20)	
Local variable pz (30)	
Argument 9: cx (30)	
Argument 8: bx (20)	
Argument 7: ax (10)	
Return address	
previous frame pointer	← rbp
Local variable dx (??)	
Local variable dy (??)	
Local variable dz (??)	← rsp

Example Program

```
if(ax > bx)
        dx = ax - bx;
else
        dx = bx - ax;
```

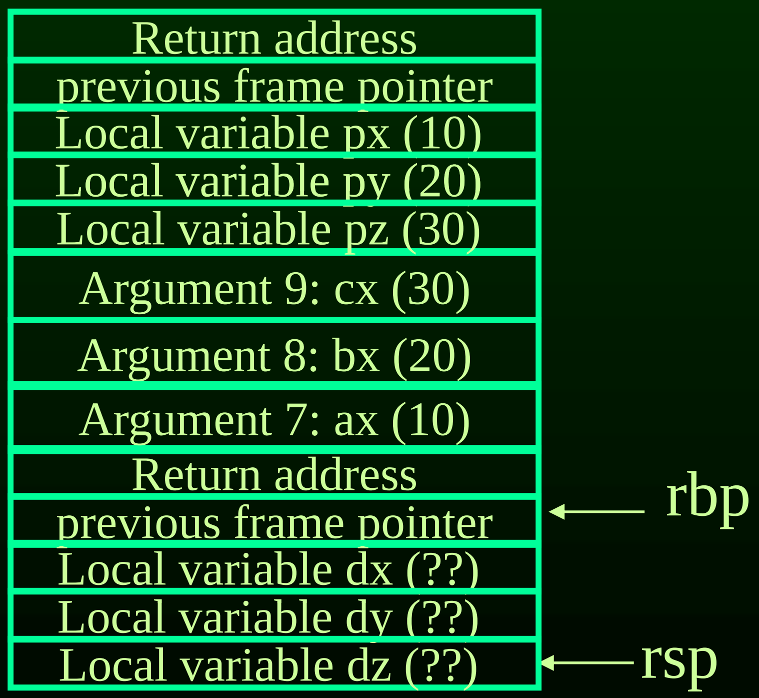
```
         movq    16(%rbp), %r10
         movq    24(%rbp), %r11
         cmpq    %r10, %r11
         jg      .L0

         <FALSE BODY>

         jmp     .L1
.L0:

         <TRUE BODY>

.L1:
```



Example Program

```
if(ax > bx)
        dx = ax - bx;
else
        dx = bx - ax;
```

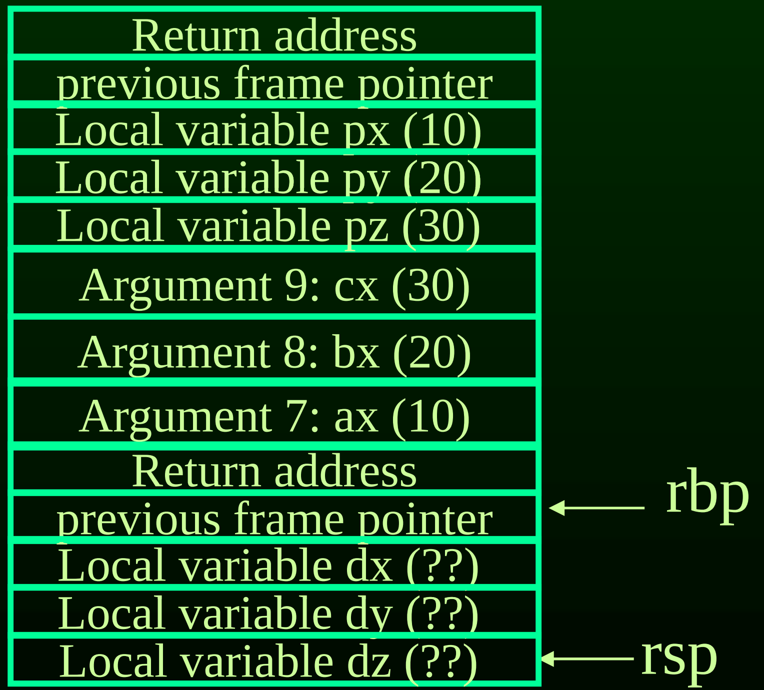
```
        movq    16(%rbp), %r10
        movq    24(%rbp), %r11
        cmpq    %r10, %r11
        jg      .L0
        movq    24(%rbp), %r10
        movq    16(%rbp), %r11
        subq    %r10, %r11
        movq    %r11, -8(%rbp)
        jmp     .L1
.L0:


       <TRUE BODY>


.L1:
```



Example Program

```
if(ax > bx)
        dx = ax - bx;
else
        dx = bx - ax;
```

```
        movq    16(%rbp), %r10
        movq    24(%rbp), %r11
        cmpq    %r10, %r11
        jg      .L0
        movq    24(%rbp), %r10
        movq    16(%rbp), %r11
        subq    %r10, %r11
        movq    %r11, -8(%rbp)
        jmp     .L1
.L0:
        movq    16(%rbp), %r10
        movq    24(%rbp), %r11
        subq    %r10, %r11
        movq    %r11, -8(%rbp)
.L1:
```



Template for while loops

```
while (test)
  body
```

Template for while loops

```
while (test)
  body
```

```
lab_cont:
      <do the test>
      joper lab_body
      jmp   lab_end
lab_body:
      <body>
      jmp   lab_cont
lab_end:
```

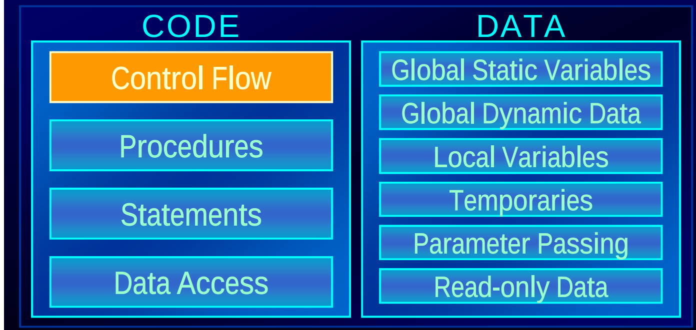
Template for while loops

```
while (test)
  body
```

```
lab_cont:
        <do the test>
        joper lab_body
        jmp   lab_end
lab_body:
        <body>
        jmp   lab_cont
lab_end:
```

- An optimized template

```
lab_cont:
        <do the test>
        joper lab_end
        <body>
        jmp   lab_cont
lab_end:
```



Question:

- What is the template for?

```
do
  body
while (test)
```

Question:

- What is the template for?

```
do
  body
while (test)
```

```
lab_begin:
     <body>
     <do test>
     joper lab_begin
```

Question:

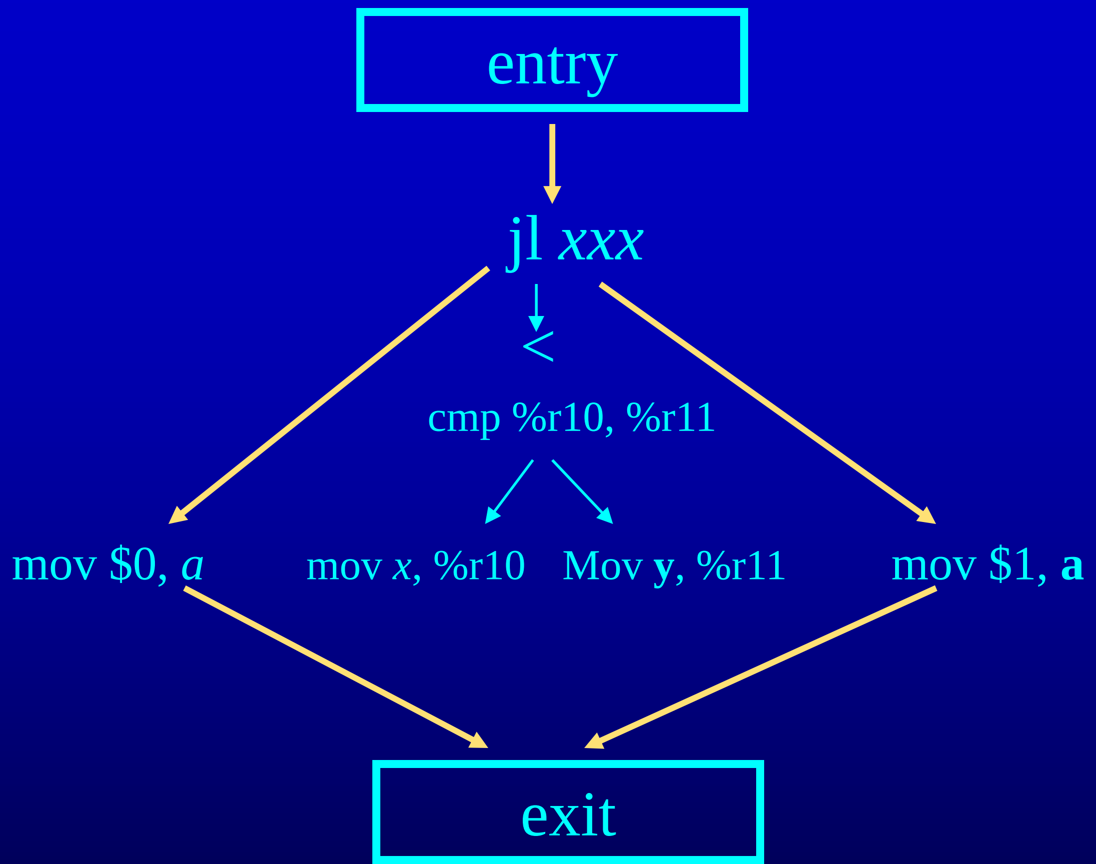
- **What is a drawback of the template based approach?**

Control Flow Graph (CFG)

- Starting point: high level intermediate format, symbol tables
- Target: CFG
  - CFG Nodes are Instruction Nodes
  - CFG Edges Represent Flow of Control
  - Forks At Conditional Jump Instructions

    Merges When Flow of Control Can Reach A Point Multiple Ways
  - Entry and Exit Nodes

```
if (x < y) {
        a = 0;
} else {
        a = 1;
}
```



Pattern for if then else

Short-Circuit Conditionals

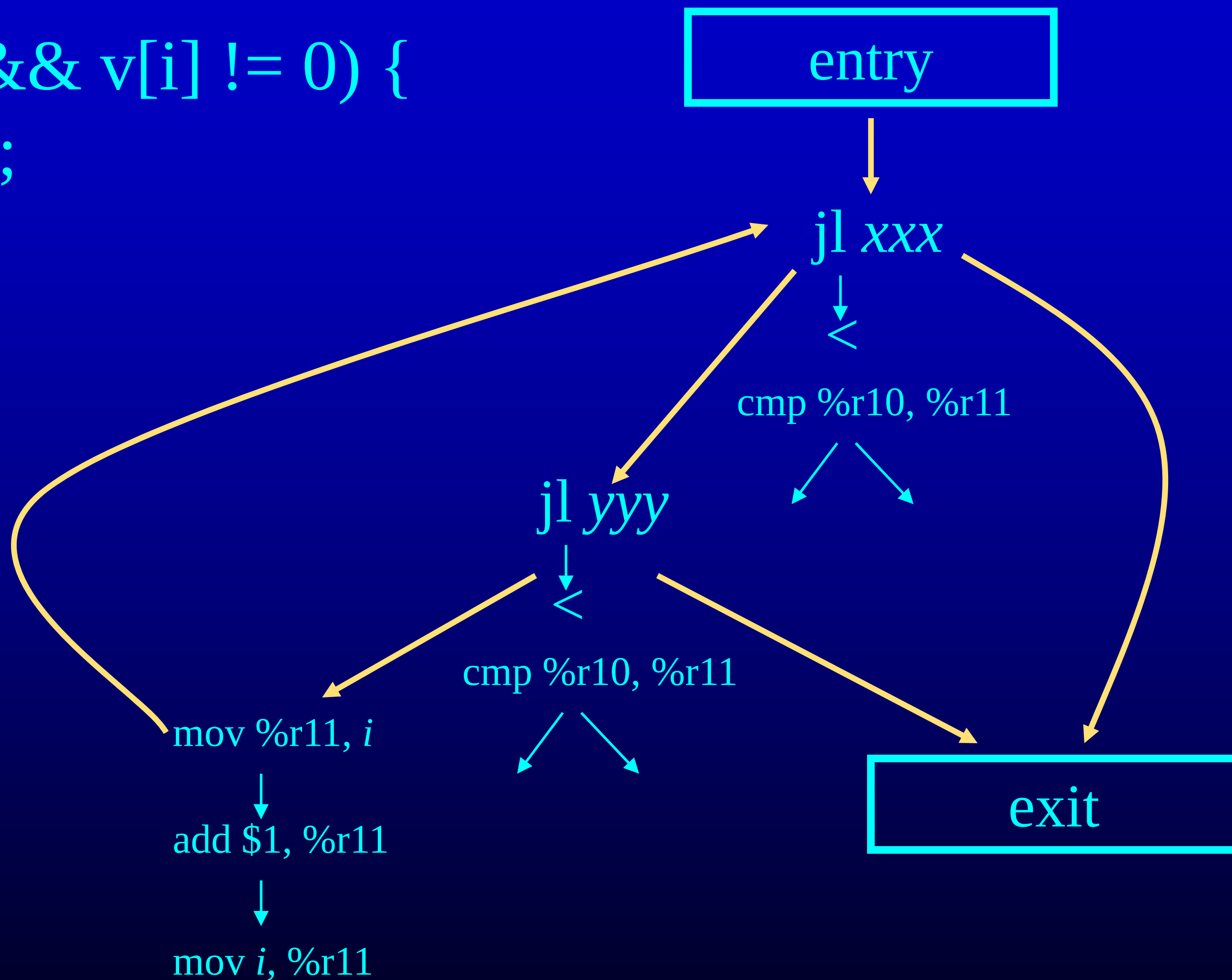
- In program, conditionals have a condition written as a boolean expression

  ((i < n) && (v[i] != 0)) || i > k)

- Semantics say should execute only as much as required to determine condition
  - Evaluate (v[i] != 0) only if (i < n) is true
  - Evaluate i > k only if ((i < n) && (v[i] != 0)) is false
- Use control-flow graph to represent this short-circuit evaluation

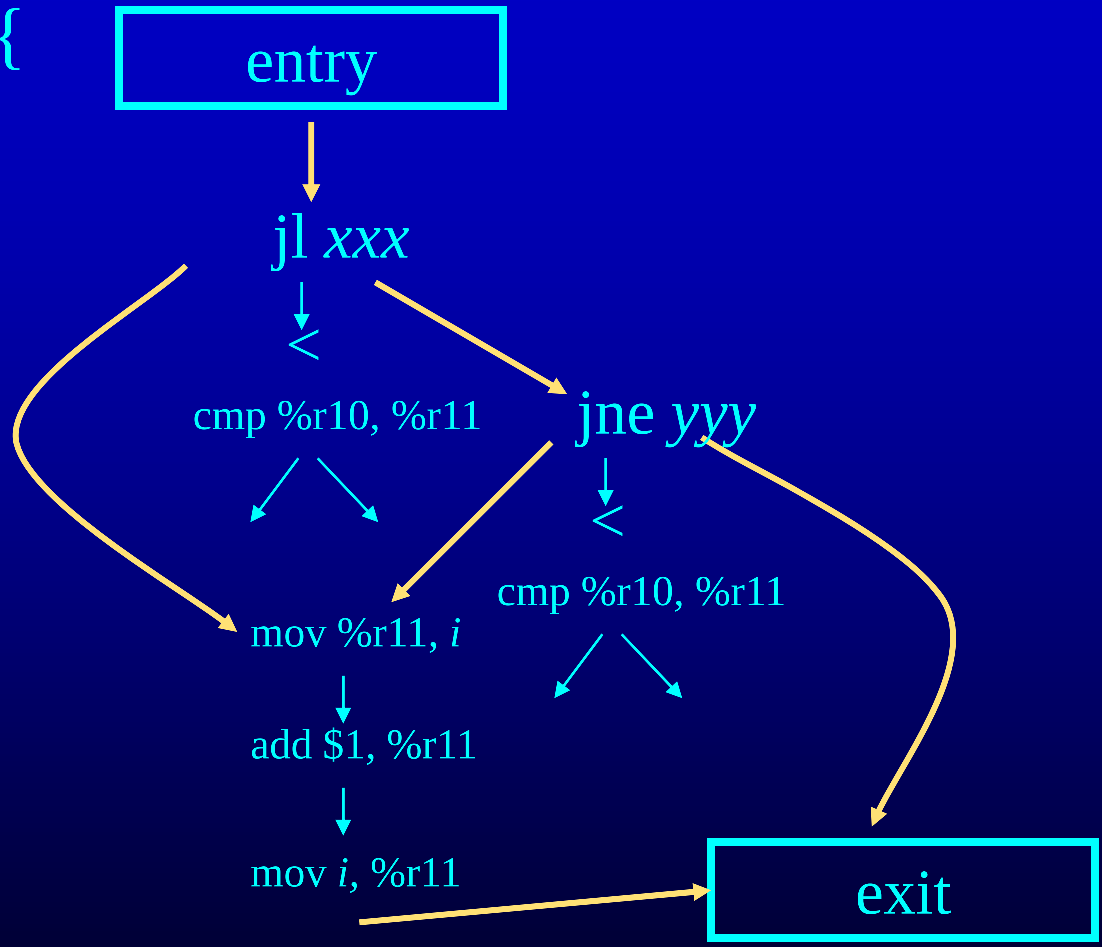
Short-Circuit Conditionals

```
while (i < n && v[i] != 0) {
        i = i+1;
}
```



More Short-Circuit Conditionals

```
if (a < b || c != 0) {
        i = i+1;
}
```



Routines for Destructuring Program Representation

destruct(n)

generates lowered form of structured code represented by n

returns (b,e) - b is begin node, e is end node in destructed form

shortcircuit(c, t, f)

generates short-circuit form of conditional represented by c

if c is true, control flows to t node

if c is false, control flows to f node

returns b - b is begin node for condition evaluation

new kind of node - nop node

Destructuring Seq Nodes

destruct(n)

generates lowered form of structured code represented by n

returns (b,e) - b is begin node, e is end node in destructed form

if n is of the form seq x y

Destructuring Seq Nodes

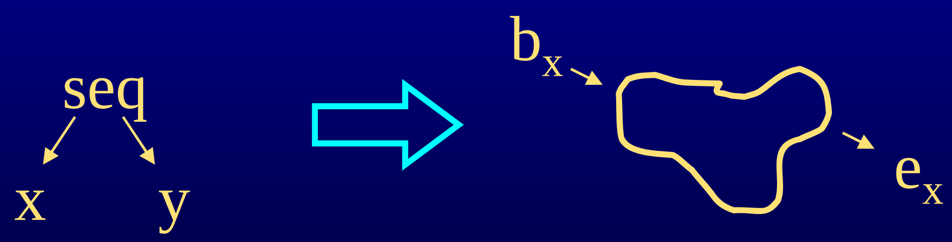
destruct(n)

generates lowered form of structured code represented by n

returns (b,e) - b is begin node, e is end node in destructed form

if n is of the form seq x y

1: (b_x,e_x) = destruct(x);



Destructuring Seq Nodes

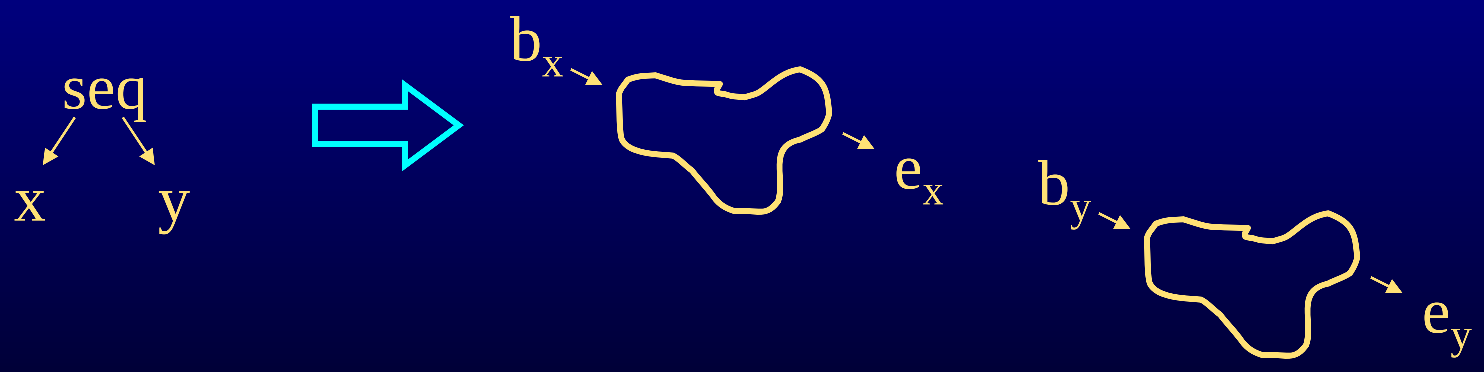
destruct(n)

generates lowered form of structured code represented by n

returns (b,e) - b is begin node, e is end node in destructed form

if n is of the form seq x y

1: (b_x,e_x) = destruct(x); 2: (b_y,e_y) = destruct(y);



Destructuring Seq Nodes

destruct(n)

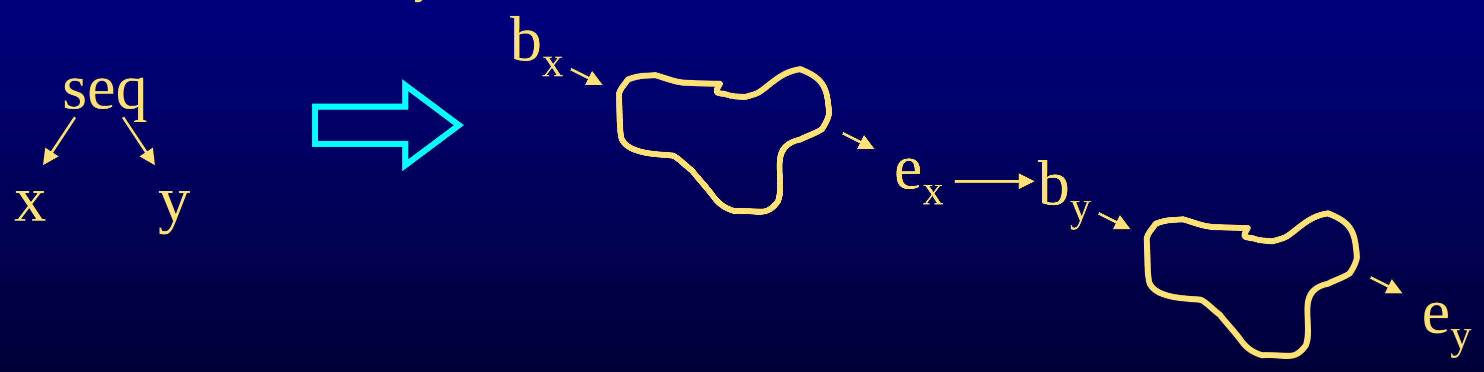
generates lowered form of structured code represented by n

returns (b,e) - b is begin node, e is end node in destructed form

if n is of the form seq x y

1: (b_x,e_x) = destruct(x); 2: (b_y,e_y) = destruct(y);

3: next(e_x) = b_y;



Destructuring Seq Nodes

destruct(n)

generates lowered form of structured code represented by n

returns (b,e) - b is begin node, e is end node in destructed form

if n is of the form seq x y

1: (b_x,e_x) = destruct(x); 2: (b_y,e_y) = destruct(y);

3: $next(e_x) = b_y$; 4: return (b_x, e_y);

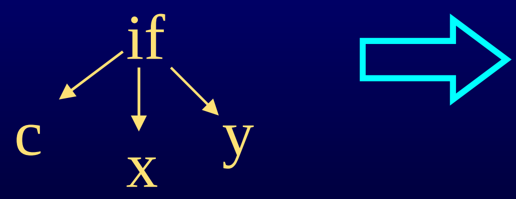
Destructuring If Nodes

destruct(n)

generates lowered form of structured code represented by n

returns (b,e) - b is begin node, e is end node in destructed form

if n is of the form if c x y



Destructuring If Nodes

destruct(n)

generates lowered form of structured code represented by n

returns (b,e) - b is begin node, e is end node in destructed form

if n is of the form if c x y

1: (b_x, e_x) = destruct(x);

Destructuring If Nodes

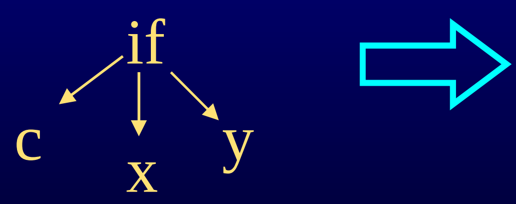
destruct(n)

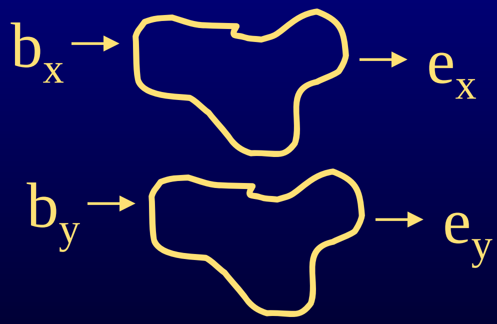
generates lowered form of structured code represented by n

returns (b,e) - b is begin node, e is end node in destructed form

if n is of the form if c x y

1: (b_x,e_x) = destruct(x); 2: (b_y,e_y) = destruct(y);





Destructuring If Nodes

destruct(n)

- generates lowered form of structured code represented by n
- returns (b,e) - b is begin node, e is end node in destructed form
- if n is of the form if c x y
  - 1: (b_x,e_x) = destruct(x); 2: (b_y,e_y) = destruct(y);
  - 3: e = new nop;

Destructuring If Nodes

destruct(n)

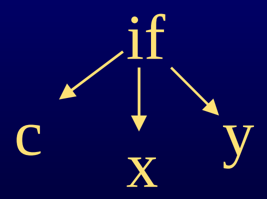
generates lowered form of structured code represented by n

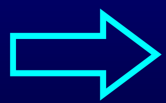
returns (b,e) - b is begin node, e is end node in destructed form

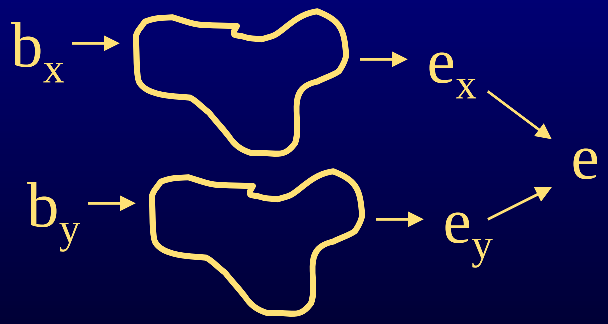
if n is of the form if c x y

1: (b_x,e_x) = destruct(x); 2: (b_y,e_y) = destruct(y);

3: e = new nop; 4: next(e_x) = e; 5: next(e_y) = e;







Destructuring If Nodes

destruct(n)

generates lowered form of structured code represented by n

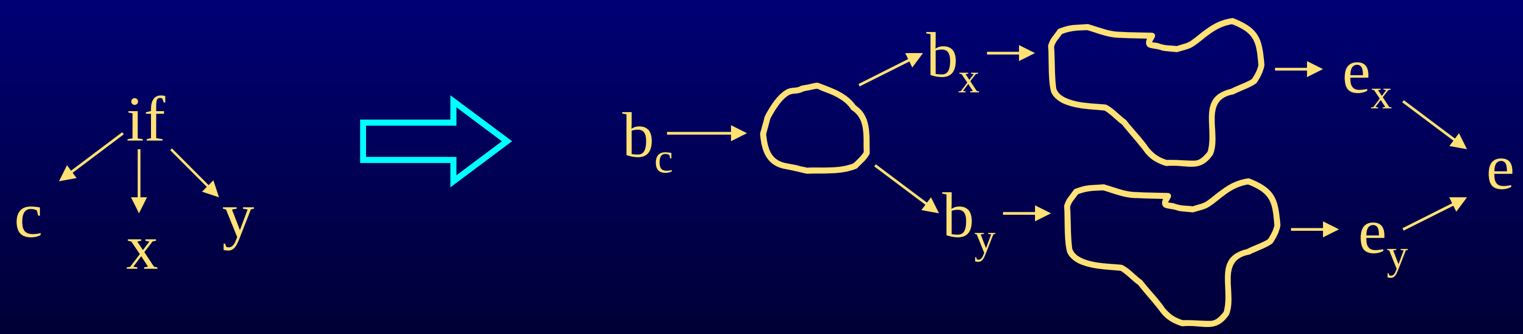
returns (b,e) - b is begin node, e is end node in destructed form

if n is of the form if c x y

1: (b_x,e_x) = destruct(x); 2: (b_y,e_y) = destruct(y);

3: e = new nop; 4: $next(e_x) = e$; 5: $next(e_y) = e$;

6: b_c = shortcircuit(c, b_x, b_y);



Destructuring If Nodes

destruct(n)

generates lowered form of structured code represented by n

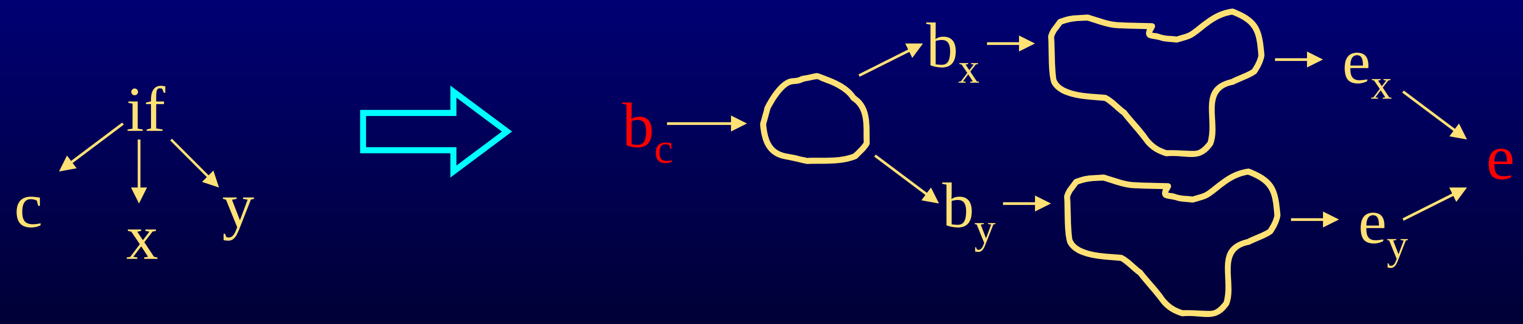
returns (b,e) - b is begin node, e is end node in destructed form

if n is of the form if c x y

1: (b_x,e_x) = destruct(x); 2: (b_y,e_y) = destruct(y);

3: e = new nop; 4: $next(e_x)$ = e; 5: $next(e_y)$ = e;

6: b_c = shortcircuit(c, b_x, b_y); 7: return (b_c, e);



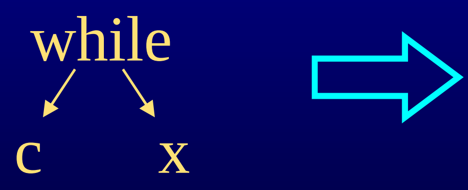
Destructuring While Nodes

destruct(n)

generates lowered form of structured code represented by n

returns (b,e) - b is begin node, e is end node in destructed form

if n is of the form while c x



Destructuring While Nodes

destruct(n)

generates lowered form of structured code represented by n

returns (b,e) - b is begin node, e is end node in destructed form

if n is of the form while c x

1: e = new nop;

Destructuring While Nodes

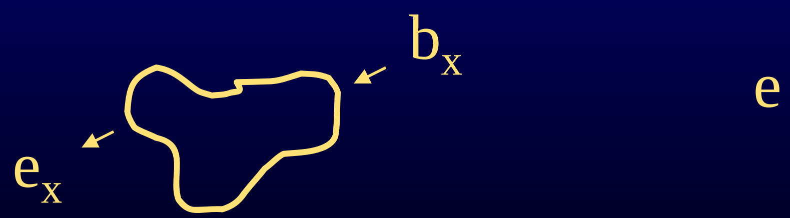
destruct(n)

generates lowered form of structured code represented by n

returns (b,e) - b is begin node, e is end node in destructed form

if n is of the form while c x

1: e = new nop; 2: (b_x,e_x) = destruct(x);



Destructuring While Nodes

destruct(n)

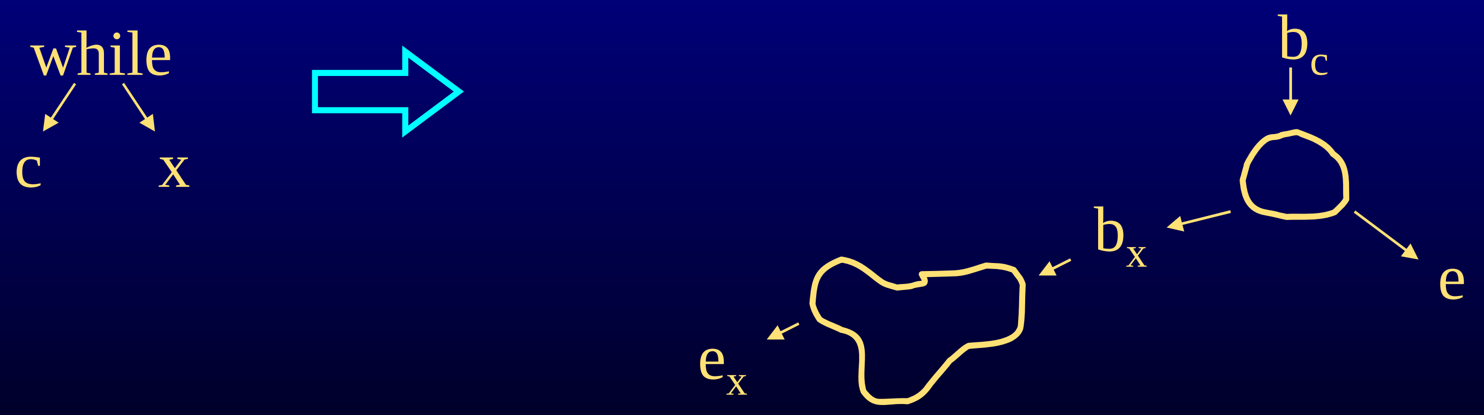
generates lowered form of structured code represented by n

returns (b,e) - b is begin node, e is end node in destructed form

if n is of the form while c x

1: e = new nop; 2: (b_x,e_x) = destruct(x);

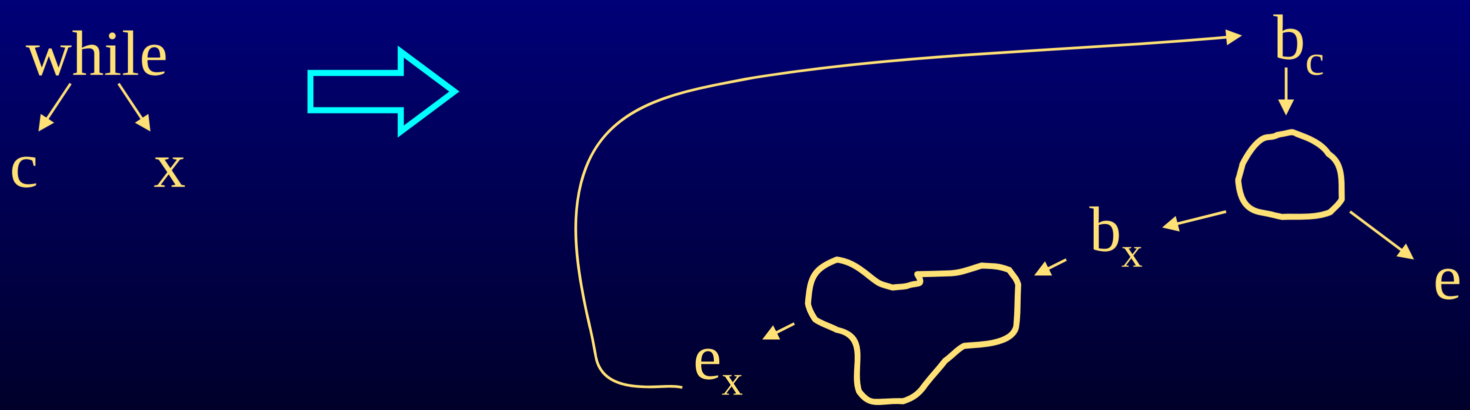
3: b_c = shortcircuit(c, b_x, e);



Destructuring While Nodes

destruct(n)

- generates lowered form of structured code represented by n
- returns (b,e) - b is begin node, e is end node in destructed form
- if n is of the form while c x
  - 1: e = new nop; 2: (b_x,e_x) = destruct(x);
  - 3: b_c = shortcircuit(c, b_x, e); 4: next(e_x) = b_c;



Destructuring While Nodes

destruct(n)

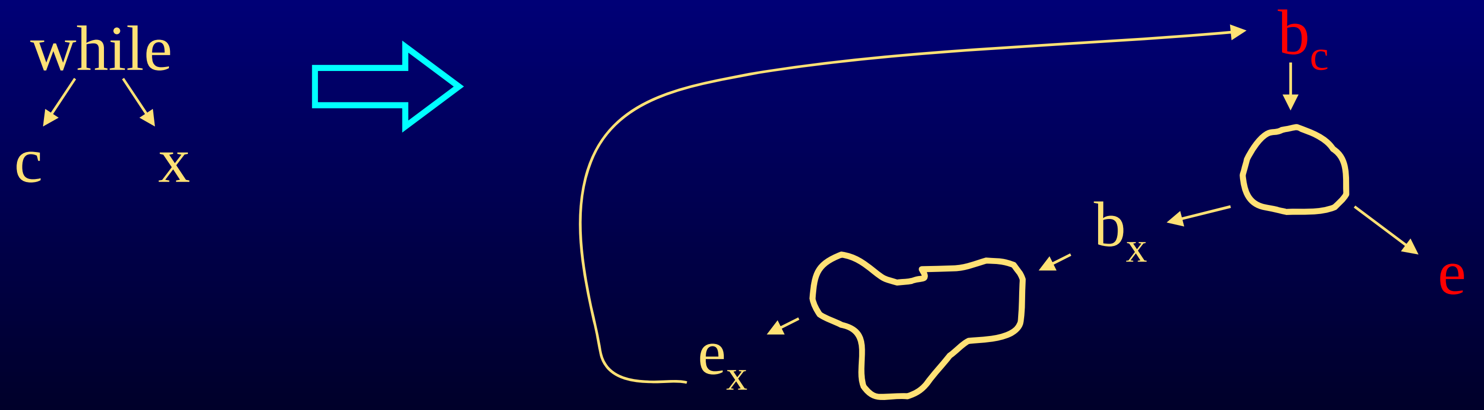
generates lowered form of structured code represented by n

returns (b,e) - b is begin node, e is end node in destructed form

if n is of the form while c x

1: e = new nop; 2: (b_x,e_x) = destruct(x);

3: b_c = shortcircuit(c, b_x, e); 4: next(e_x) = b_c; 5: return (b_c, e);



Shortcircuiting And Conditions

shortcircuit(c, t, f)

generates shortcircuit form of conditional represented by c

returns b - b is begin node of shortcircuit form

if c is of the form c_1 && c_2

c_1 && c_2

Shortcircuiting And Conditions

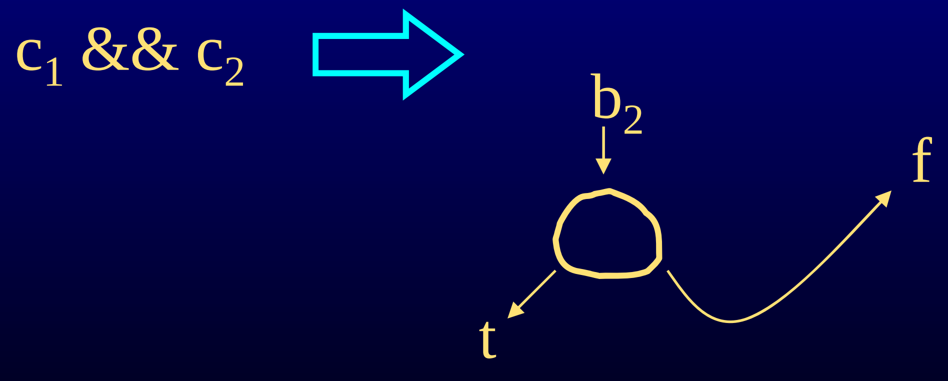
shortcircuit(c, t, f)

generates shortcircuit form of conditional represented by c

returns b - b is begin node of shortcircuit form

if c is of the form c_1 && c_2

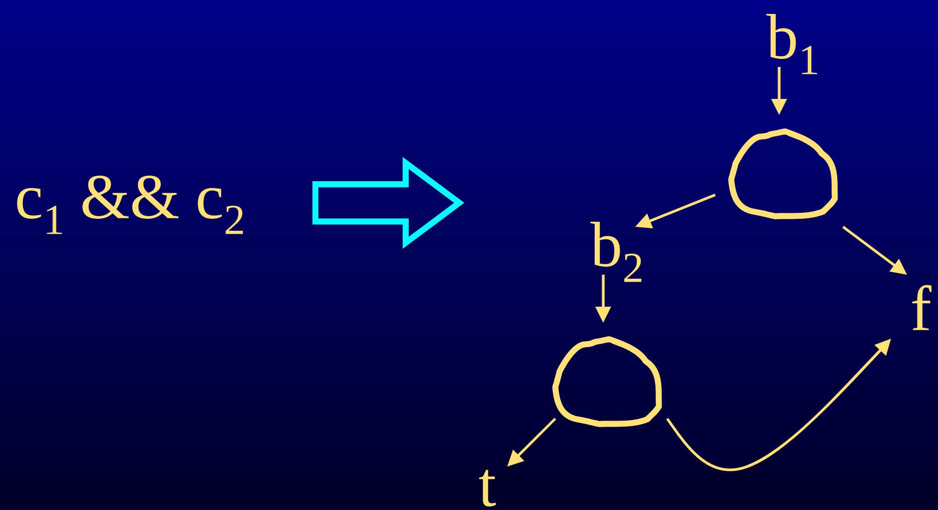
1: b_2 = shortcircuit(c_2, t, f);



Shortcircuiting And Conditions

shortcircuit(c, t, f)

- generates shortcircuit form of conditional represented by c
- returns b - b is begin node of shortcircuit form
- if c is of the form c_1 && c_2
  - 1: b_2 = shortcircuit(c_2, t, f); 2: b_1 = shortcircuit(c_1, b_2, f);



Shortcircuiting And Conditions

shortcircuit(c, t, f)

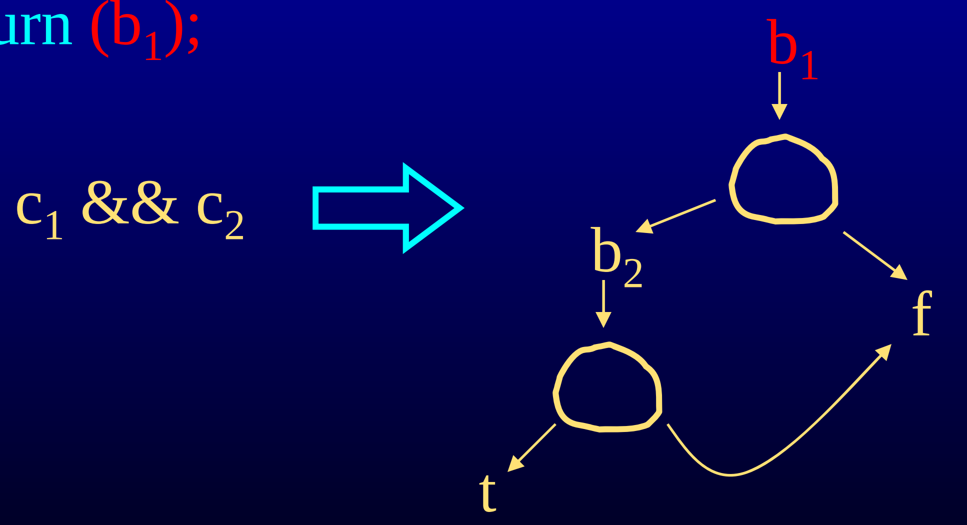
generates shortcircuit form of conditional represented by c

returns b - b is begin node of shortcircuit form

if c is of the form c_1 && c_2

1: b_2 = shortcircuit(c_2, t, f); 2: b_1 = shortcircuit(c_1, b_2, f);

3: return (b_1);



Shortcircuiting Or Conditions

shortcircuit(c, t, f)

- generates shortcircuit form of conditional represented by c
- returns b - b is begin node of shortcircuit form
- if c is of the form $c_1 \| c_2$

$c_1 \| c_2$

Shortcircuiting Or Conditions

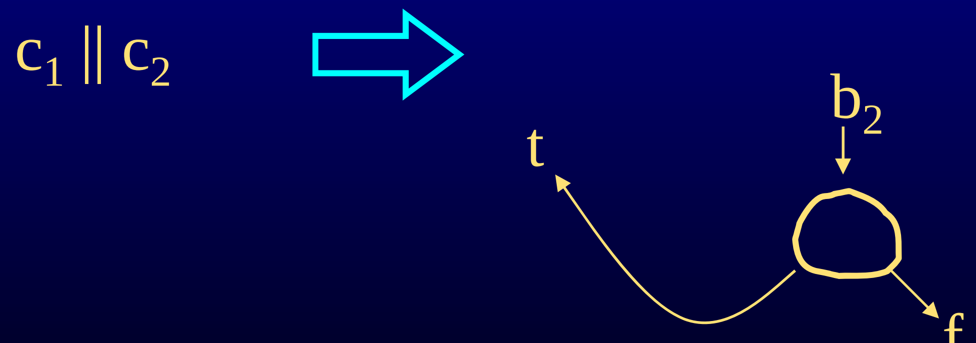
shortcircuit(c, t, f)

generates shortcircuit form of conditional represented by c

returns b - b is begin node of shortcircuit form

if c is of the form $c_1 \| c_2$

1: b_2 = shortcircuit(c_2, t, f);



Shortcircuiting Or Conditions

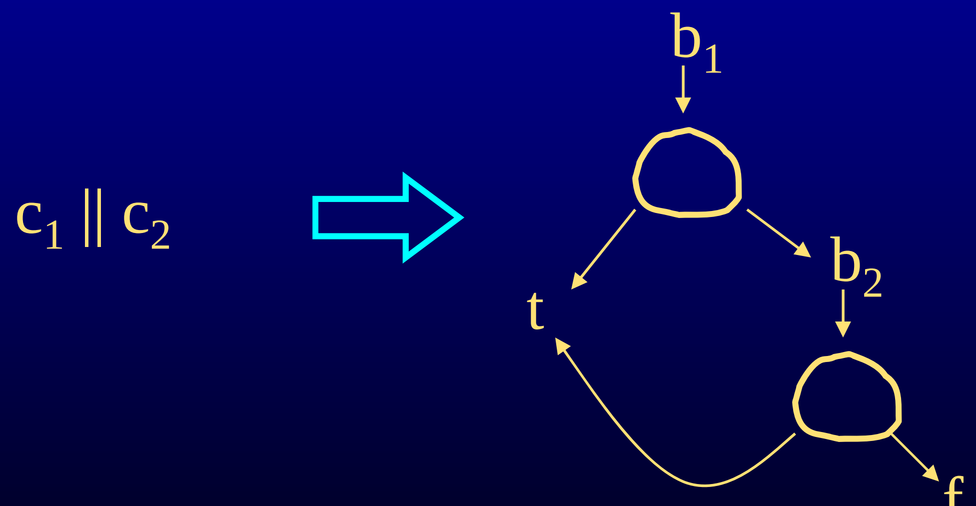
shortcircuit(c, t, f)

generates shortcircuit form of conditional represented by c

returns b - b is begin node of shortcircuit form

if c is of the form $c_1 \| c_2$

1: b_2 = shortcircuit(c_2, t, f); 2: b_1 = shortcircuit(c_1, t, b_2);



Shortcircuiting Or Conditions

shortcircuit(c, t, f)

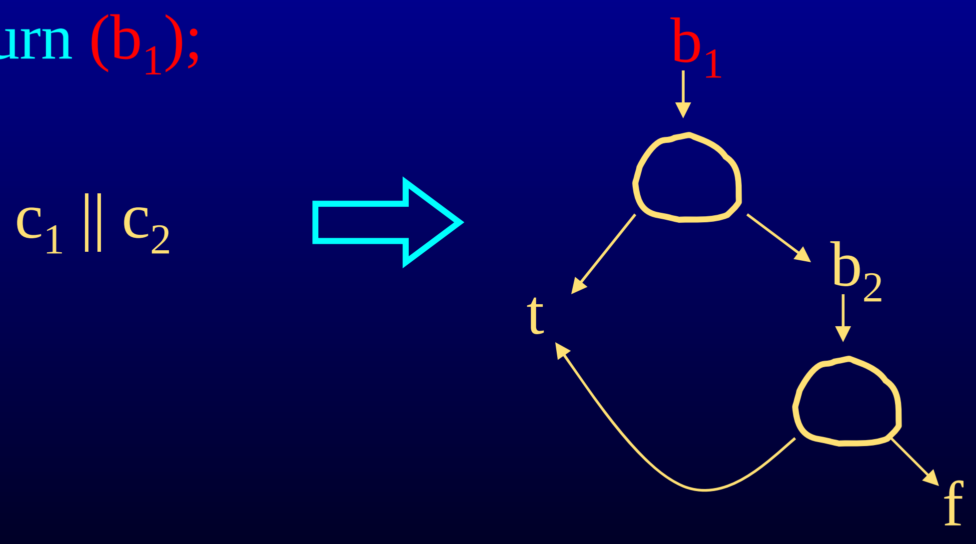
generates shortcircuit form of conditional represented by c

returns b - b is begin node of shortcircuit form

if c is of the form $c_1 \| c_2$

1: b_2 = shortcircuit(c_2, t, f); 2: b_1 = shortcircuit(c_1, t, b_2);

3: return (b_1);



Shortcircuiting Not Conditions

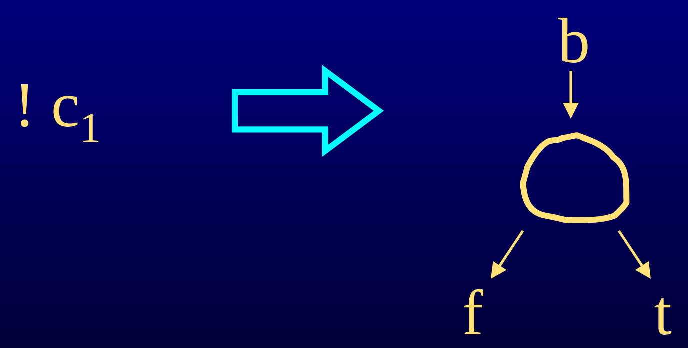
shortcircuit(c, t, f)

generates shortcircuit form of conditional represented by c

returns b - b is begin node of shortcircuit form

if c is of the form ! c_1

1: b = shortcircuit(c_1, f, t); return(b);



Computed Conditions

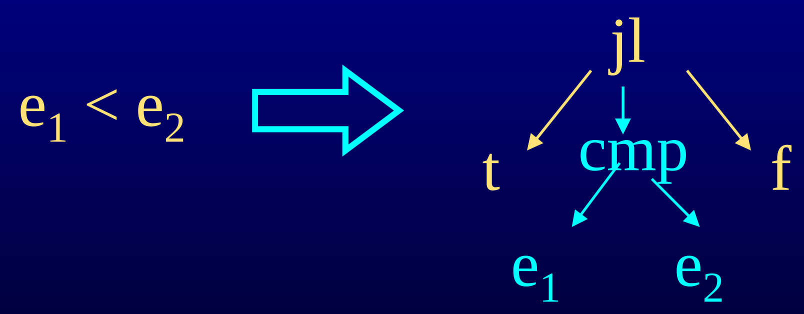
shortcircuit(c, t, f)

generates shortcircuit form of conditional represented by c

returns b - b is begin node of shortcircuit form

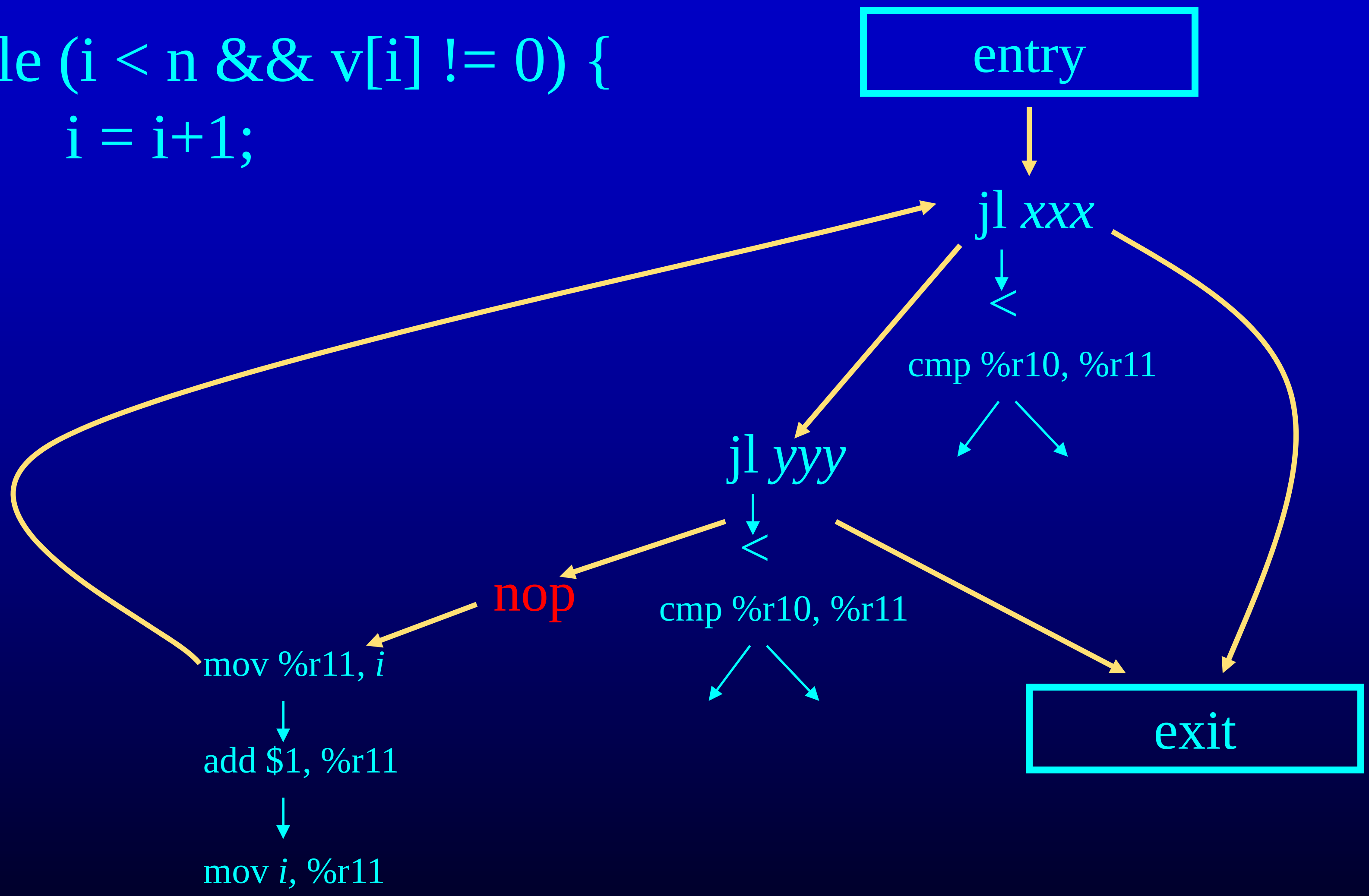
if c is of the form $e_1 < e_2$

1: b = new cbr($e_1 < e_2$, t, f); 2: return (b);

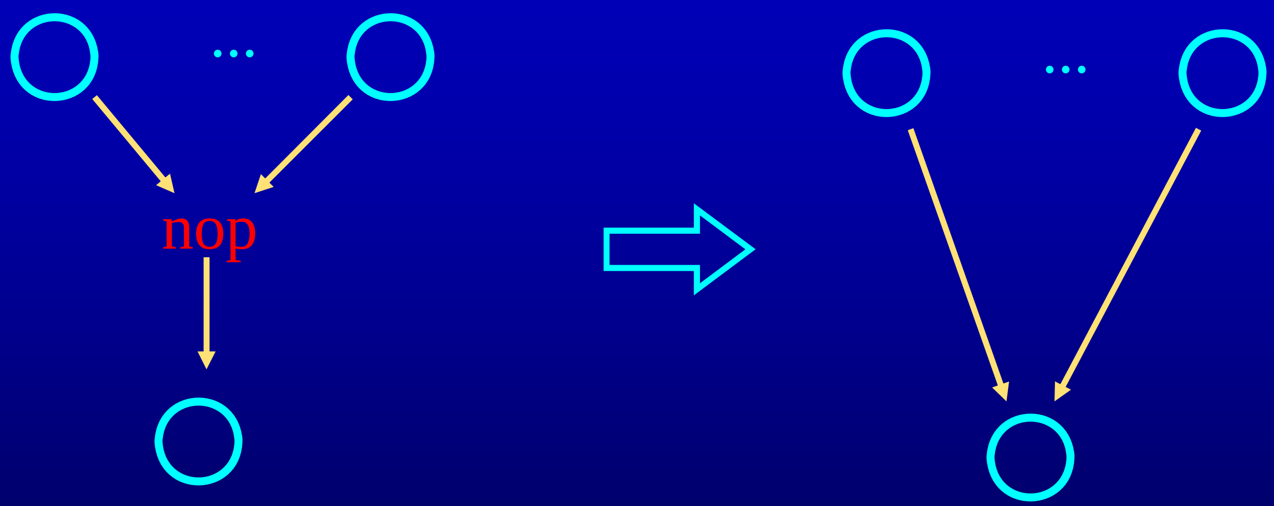


Nops In Destructured Representation

```
while (i < n && v[i] != 0) {
        i = i+1;
}
```



Eliminating Nops Via Peephole Optimization



Question:

- What are the pros and cons of template matching vs. algorithmic approach?

Outline

- Generation of statements
- Generation of control flow
- x86-64 Processor
- Guidelines in writing a code generator

Guidelines for the code generator

- Lower the abstraction level slowly
  - Do many passes, that do few things (or one thing)
    - Easier to break the project down, generate and debug
- Keep the abstraction level consistent
  - IR should have ‘correct’ semantics at all time
    - At least you should know the semantics
  - You may want to run some of the optimizations between the passes.
- Use assertions liberally
  - Use an assertion to check your assumption

Guidelines for the code generator

- Do the simplest but dumb thing
  - it is ok to generate 0 + 1*x + 0*y
  - Code is painful to look at; let optimizations improve it
- Make sure you know want can be done at…
  - Compile time in the compiler
  - Runtime using generated code

Guidelines for the code generator

- Remember that optimizations will come later
  - Let the optimizer do the optimizations
  - Think about what optimizer will need and structure your code accordingly
  - Example: Register allocation, algebraic simplification, constant propagation
- Setup a good testing infrastructure
  - regression tests
    - If a input program creates a bug, use it as a regression test
  - Learn good bug hunting procedures
    - Example: binary search , delta debugging