6.110 Computer Language Engineering

Quiz 1 Review Session

March 13, 2024

Quiz 1: **Friday, March 15**

- Quiz will be in class, worth **10%** of the overall grade
- **Open-book**, any *direct* link from course website is OK except Godbolt, your own notes are OK, no wider internet or ChatGPT
- Covers lecture content up to yesterday's lecture:
  - Regex, context-free grammars
  - Top-down parsing
  - High-level IR and semantics
  - Unoptimized codegen
- Past quizzes are now on course website

Regex, automata ←

Context-free grammars, top-down parsing

High-level IR and semantics

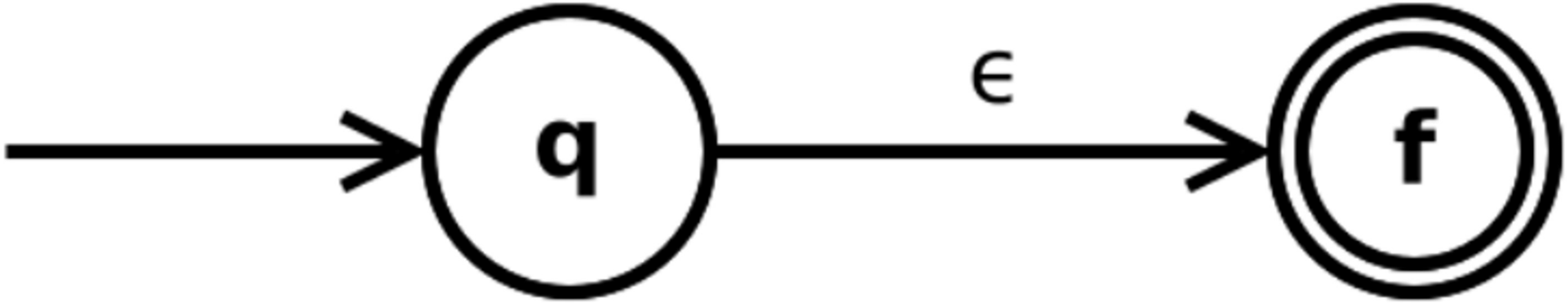
Unoptimized codegen

Regex, NFAs, DFAs

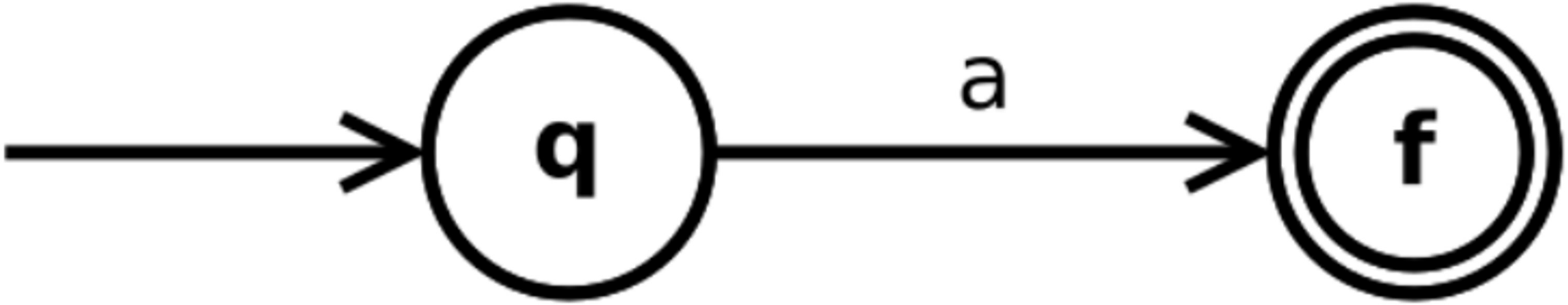
- Regular expressions, NFAs, and DFAs all have the same strength: they describe *regular languages*
- Conversion from regular expressions to NFAs: **Thompson's construction**
- Conversion from NFAs to DFAs: states in DFAs are *sets* of states in NFAs.
  - Blowup: n states in NFA → at most 2^n states in DFA

Thompson's construction

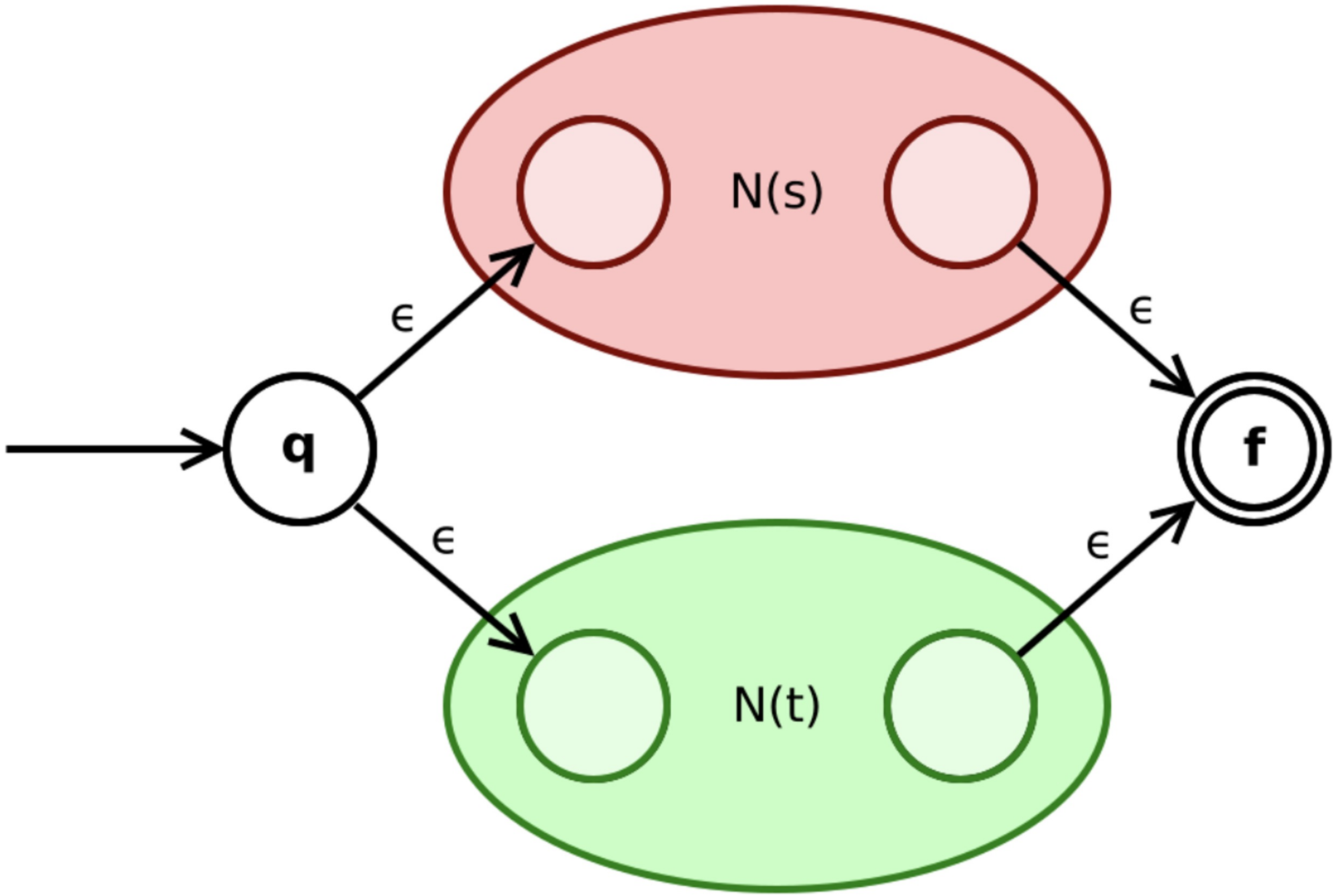
The **empty-expression** ε is converted to



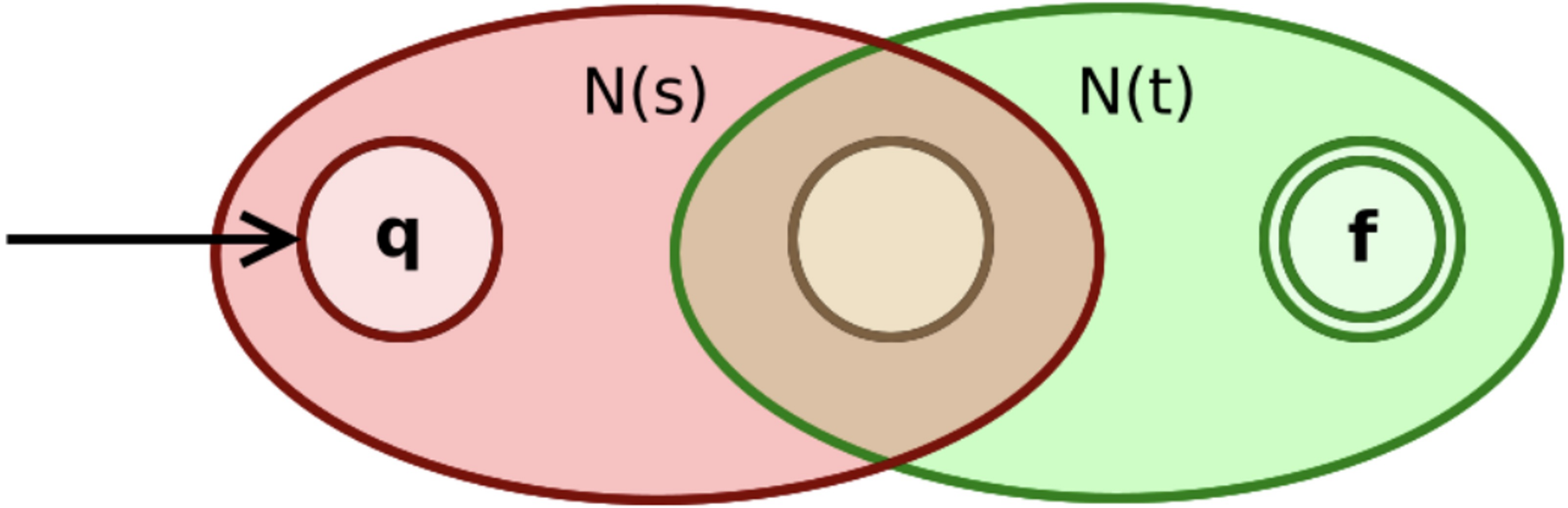
A **symbol** *a* of the input alphabet is converted to



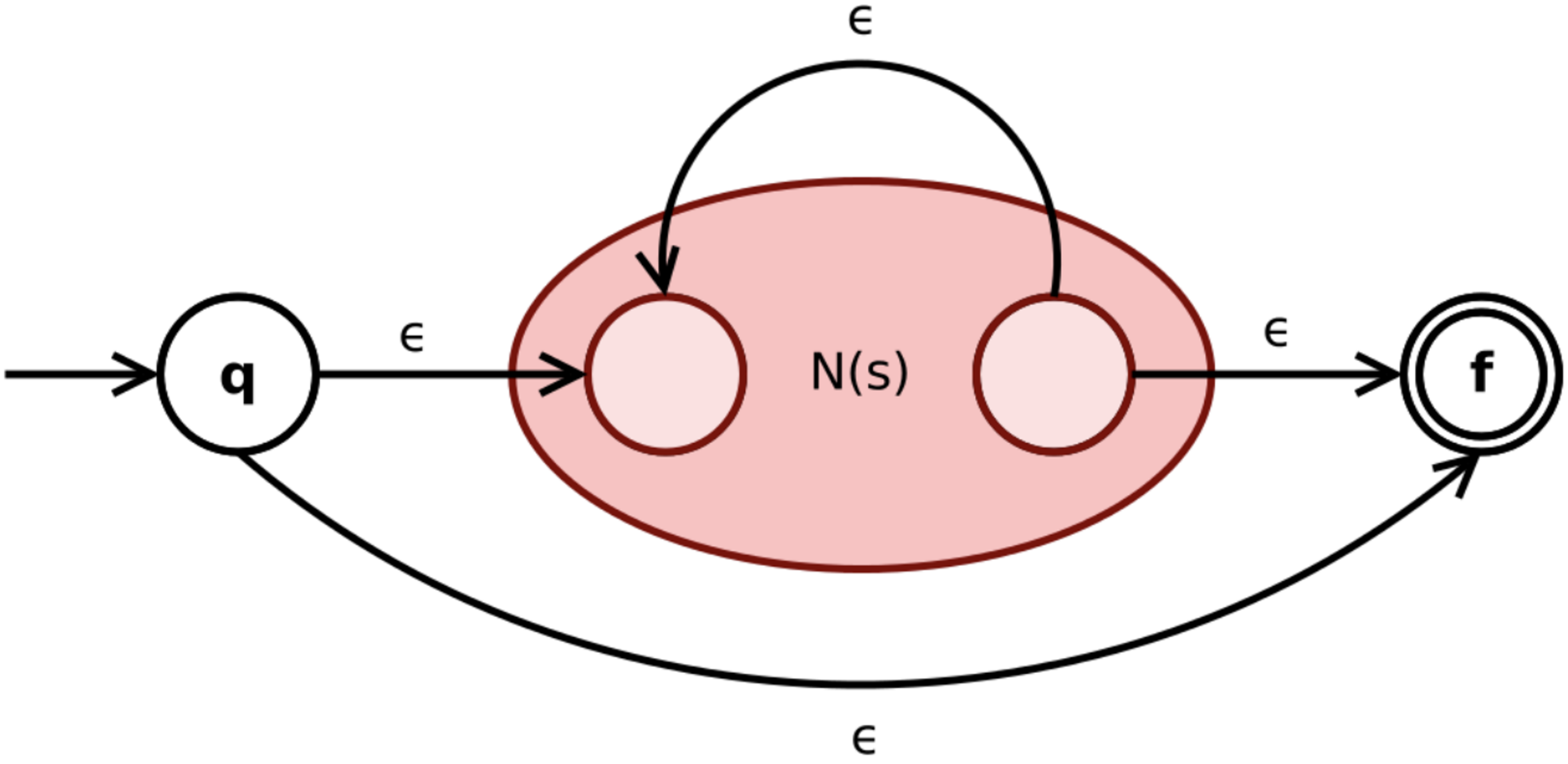
The **union expression** *s*|*t* is converted to



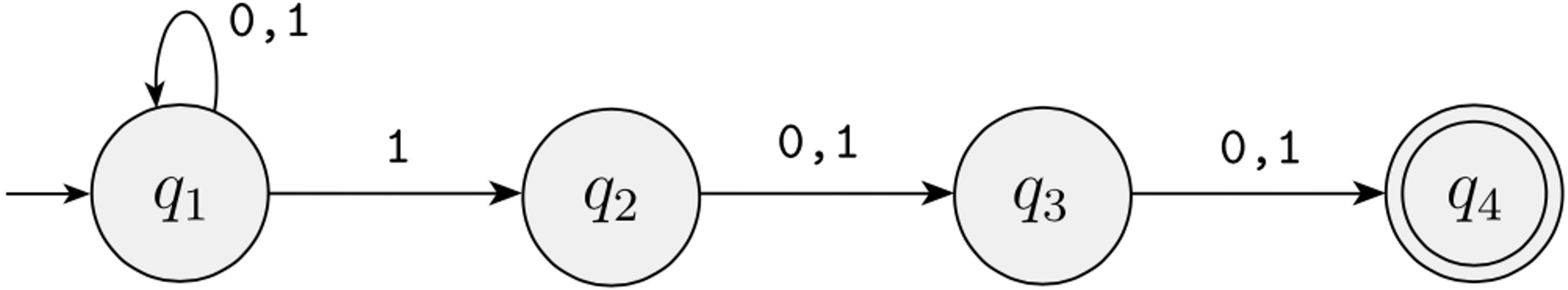
The **concatenation expression** *st* is converted to

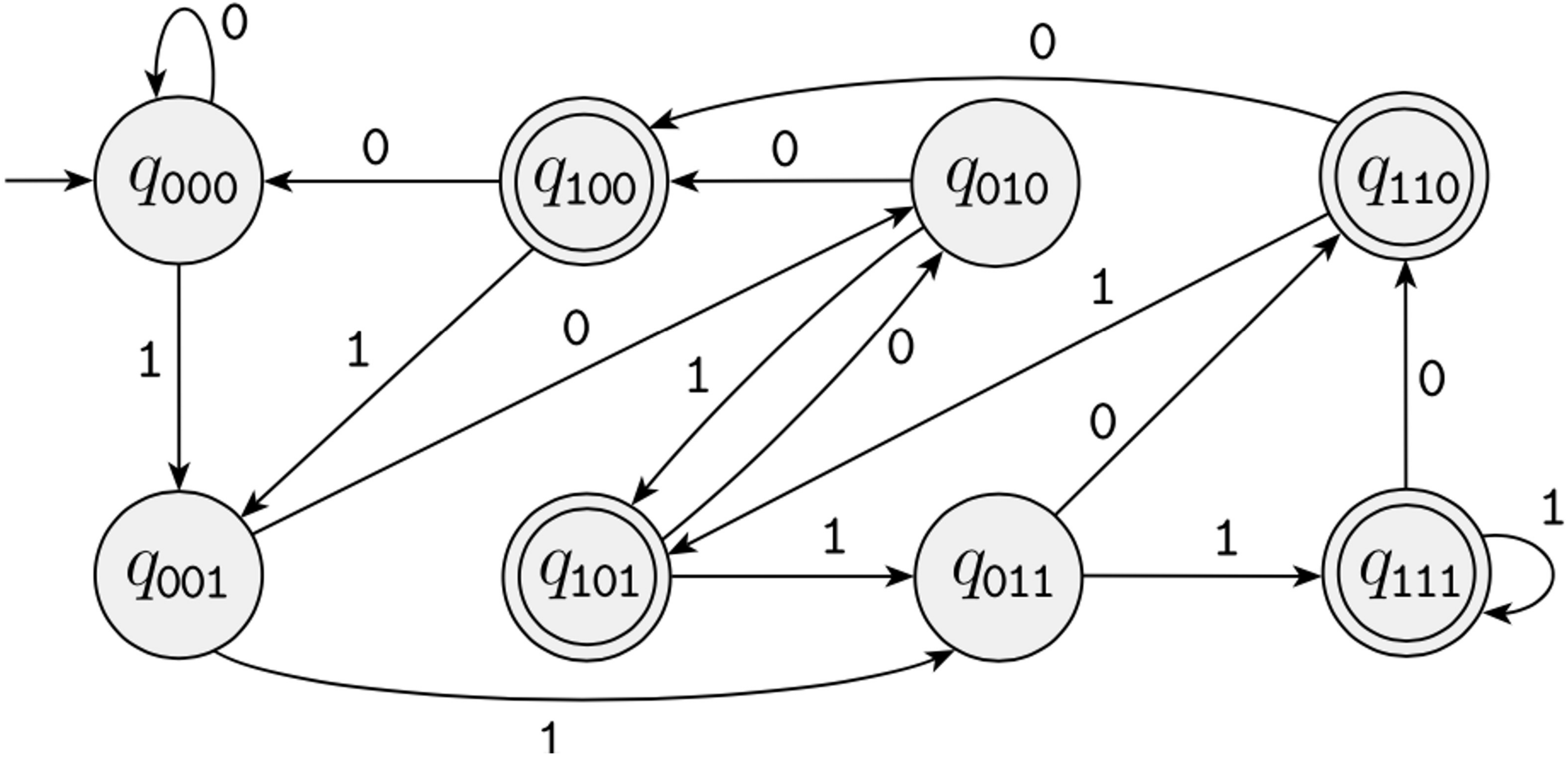


The **Kleene star expression** s^* is converted to



NFA → DFA


0,1
1
0,1
0,1
q_1
q_2
q_3
q_4



q_{000}
q_{100}
q_{010}
q_{110}
q_{001}
q_{101}
q_{011}
q_{111}
0
1

Regex, automata

Context-free grammars, top-down parsing ←

High-level IR and semantics

Unoptimized codegen

Context-free grammars

- Stronger than regexes: language $\{a^n b^n\}$ is recognizable by a context-free grammar but not a regex
- Issues to worry about in parsing
  - Ambiguity
  - Left recursion
  - Operator precedence

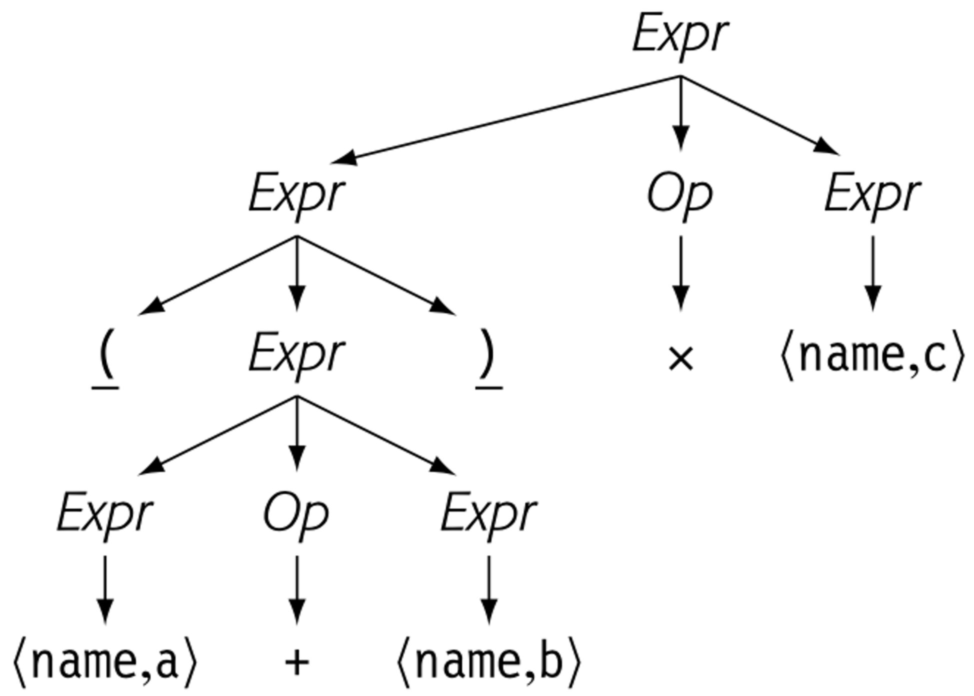
$$
\begin{aligned}
\langle \text{SENTENCE} \rangle &\to \langle \text{NOUN-PHRASE} \rangle \langle \text{VERB-PHRASE} \rangle \\
\langle \text{NOUN-PHRASE} \rangle &\to \langle \text{CMPLX-NOUN} \rangle \mid \langle \text{CMPLX-NOUN} \rangle \langle \text{PREP-PHRASE} \rangle \\
\langle \text{VERB-PHRASE} \rangle &\to \langle \text{CMPLX-VERB} \rangle \mid \langle \text{CMPLX-VERB} \rangle \langle \text{PREP-PHRASE} \rangle \\
\langle \text{PREP-PHRASE} \rangle &\to \langle \text{PREP} \rangle \langle \text{CMPLX-NOUN} \rangle \\
\langle \text{CMPLX-NOUN} \rangle &\to \langle \text{ARTICLE} \rangle \langle \text{NOUN} \rangle \\
\langle \text{CMPLX-VERB} \rangle &\to \langle \text{VERB} \rangle \mid \langle \text{VERB} \rangle \langle \text{NOUN-PHRASE} \rangle \\
\langle \text{ARTICLE} \rangle &\to \texttt{a} \mid \texttt{the} \\
\langle \text{NOUN} \rangle &\to \texttt{boy} \mid \texttt{girl} \mid \texttt{flower} \\
\langle \text{VERB} \rangle &\to \texttt{touches} \mid \texttt{likes} \mid \texttt{sees} \\
\langle \text{PREP} \rangle &\to \texttt{with}
\end{aligned}
$$

1	*Expr*	$\rightarrow$	<u>(</u> *Expr* <u>)</u>		4	*Op*	$\rightarrow$	+
2		|	*Expr* *Op* *Expr*		5		|	-
3		|	`name`		6		|	×
					7		|	÷

1	*Expr*	→	(*Expr*)	4	*Op*	→	+
2		|	*Expr* *Op* *Expr*	5		|	-
3		|	`name`	6		|	×
				7		|	÷

(a + b) x c

1	*Expr*	→	(*Expr*)
2		|	*Expr* *Op* *Expr*
3		|	name
4	*Op*	→	+
5		|	-
6		|	×
7		|	÷



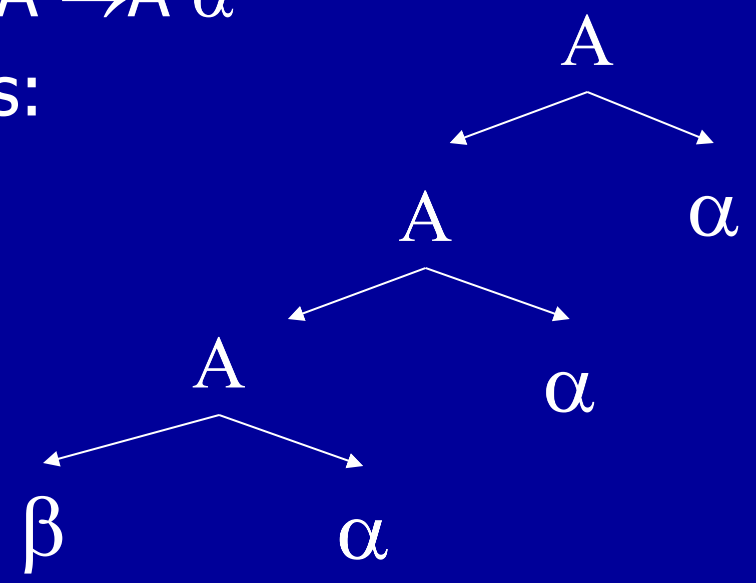
Ambiguity

1	*Stmt*	→	`if` *Expr* `then` *Stmt*
2		|	`if` *Expr* `then` *Stmt* `else` *Stmt*
3		|	*Other*

Left factoring

Eliminating Left Recursion

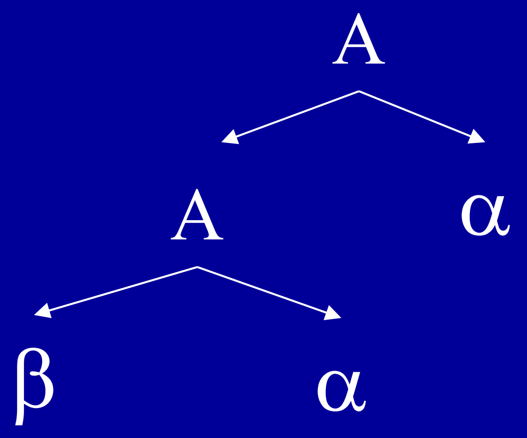
- Start with productions of form
  - $A \rightarrow A\ \alpha$
  - $A \rightarrow \beta$
  - α, β sequences of terminals and nonterminals that do not start with A
- Repeated application of $A \rightarrow A\ \alpha$ builds parse tree like this:



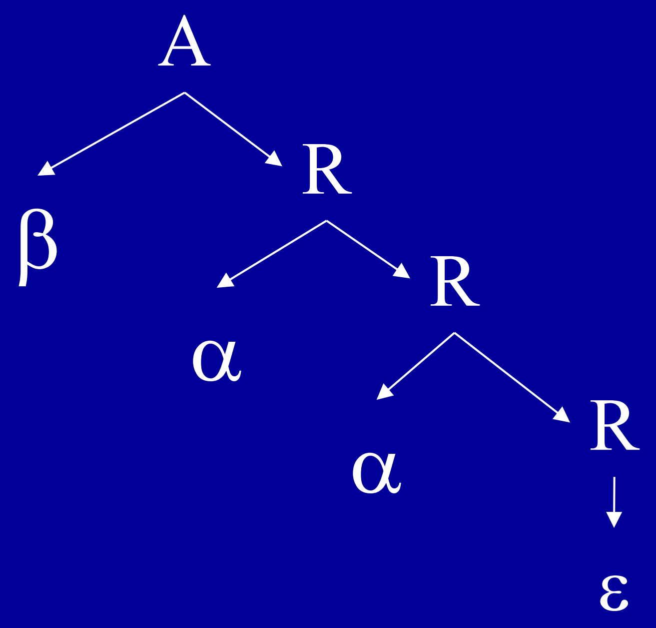
Eliminating Left Recursion

- Replacement productions
  - $A \rightarrow A\ \alpha$ $\quad$ $A \rightarrow \beta\ R$ $\quad$ R is a new nonterminal
  - $A \rightarrow \beta$ $\quad$ $R \rightarrow \alpha\ R$
  - $\quad$ $R \rightarrow \varepsilon$

Old Parse Tree



New Parse Tree



Hacked Grammar

Original Grammar Fragment

$Term \rightarrow Term * \text{Int}$

$Term \rightarrow Term / \text{Int}$

$Term \rightarrow \text{Int}$

New Grammar Fragment

$Term \rightarrow \text{Int}\ Term'$

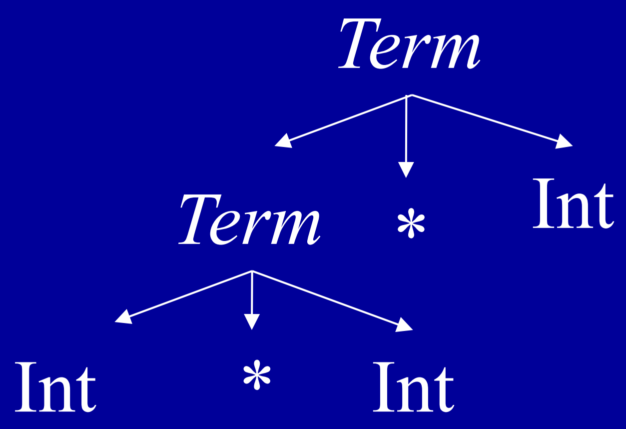
$Term' \rightarrow * \text{Int}\ Term'$

$Term' \rightarrow / \text{Int}\ Term'$

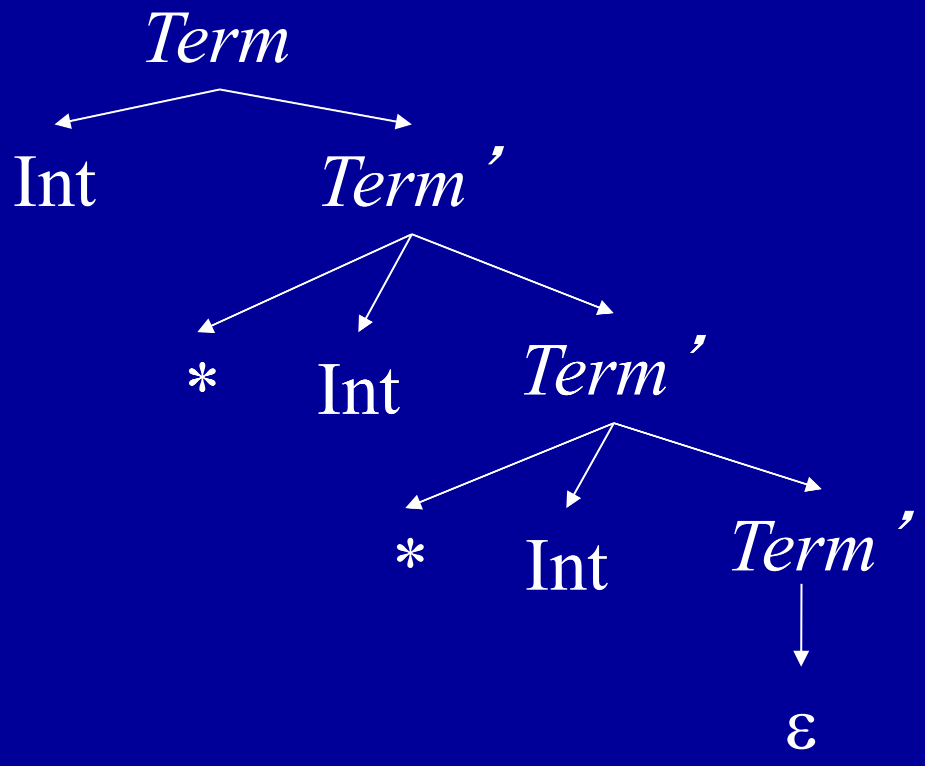
$Term' \rightarrow \varepsilon$

Parse Tree Comparisons

Original Grammar



New Grammar



Precedence climbing

1	*Expr*	→	(*Expr*)	4	*Op*	→	+
2		|	*Expr* *Op* *Expr*	5		|	-
3		|	`name`	6		|	×
				7		|	÷

1	*Expr*	→	(*Expr*)
2		|	*Expr* *Op* *Expr*
3		|	`name`

4	*Op*	→	+
5		|	-
6		|	×
7		|	÷

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* × *Factor*
5		|	*Term* ÷ *Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	`num`
9		|	`name`

Regex, automata

Context-free grammars, top-down parsing

High-level IR and semantics ←

Unoptimized codegen

High-level IR

- Goal: **semantic checking** and **program analysis**
- Augment an AST with **symbol tables**, so that we can look up identifiers

Symbol tables

- Stores relevant information about each identifier

identifier →	*descriptor*
`x`	local variable id 1, type int
`f`	method id 3, type bool → int

Scope

global scope

```
import printf;
int x = 0;

void main() {
    int x = 1, y = 2;
    if (x > 0)
    {
        int x = 3;
        printf(“%d %d”, x + y);
    }
}
```

method scope

block scope

Symbol tables

global symbol table

`printf` → imported method
`x` → global variable, type = int
`main` → method, params = [], return type = void

child of

symbol table

- `x` → local variable, type = int
- `y` → local variable, type = int

child of

symbol table

- `x` → local variable, type = int

Scope

global scope

method scope

block scope

```
import printf;
int x = 0;

void main() {
    int x = 1, y = 2;
    if (x > 0)
    {
        int x = 3;
        printf(“%d %d”, x + y);
    }
}
```

Summary

- One symbol table per scope
  - Each symbol table links to symbol table of parent scope
- First search for identifier in current scope
  - If not found, go to parent symbol table
  - If not found in any table, *semantic error!*

For the quiz, you should know how to:

- Explain what descriptors are and describe what information they contain
- Construct symbol tables for simple programs, including programs with simple classes
- Identify the scope of each identifier

Type compatibility

```
class A {
    int x;
}
class B extends A {
    int y;
}
```

We say

- B is compatible with A
- B is a subtype of A
- B can substitute for A

(The reverse is not true!)

Type compatibility

```
class A {
    int x;
}
class B extends A {
    int y;
}
B f(A a);
```

```
A a;
B b;
a.y = 1;  // invalid
b.x = 0;  // valid
a = b;    // valid
b = a;    // invalid
a = f(b); // valid
```

For the quiz, you should know how to:

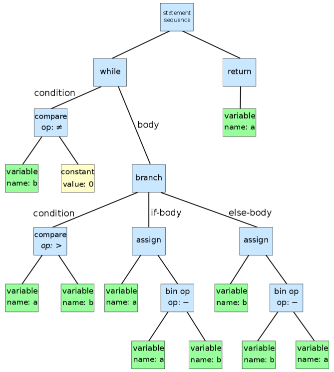
- Determine what semantic checks need to be done for each given statement
- Perform semantic checks on a given program
- Determine compatibility of subclasses/superclasses

Regex, automata

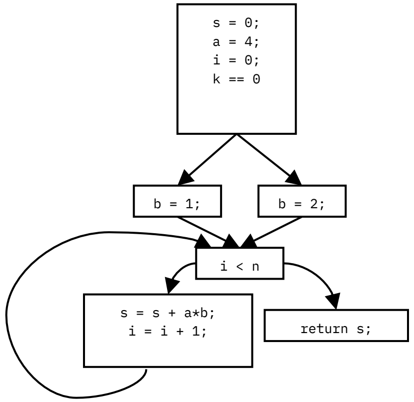
Context-free grammars, top-down parsing

High-level IR and semantics

Unoptimized codegen ←



High-level IR (AST)



Low-level IR (CFG)

Code generation →

```
push %rbp
mov  %rsp, %rbp
…
```

x86-64 assembly

Structured control flow
if/else, loops, break, continue

Destructuring →

Unstructured control flow
edges = jumps

Unstructured control flow
jumps only!

Complex expressions
`x+=y[4*z]/a`

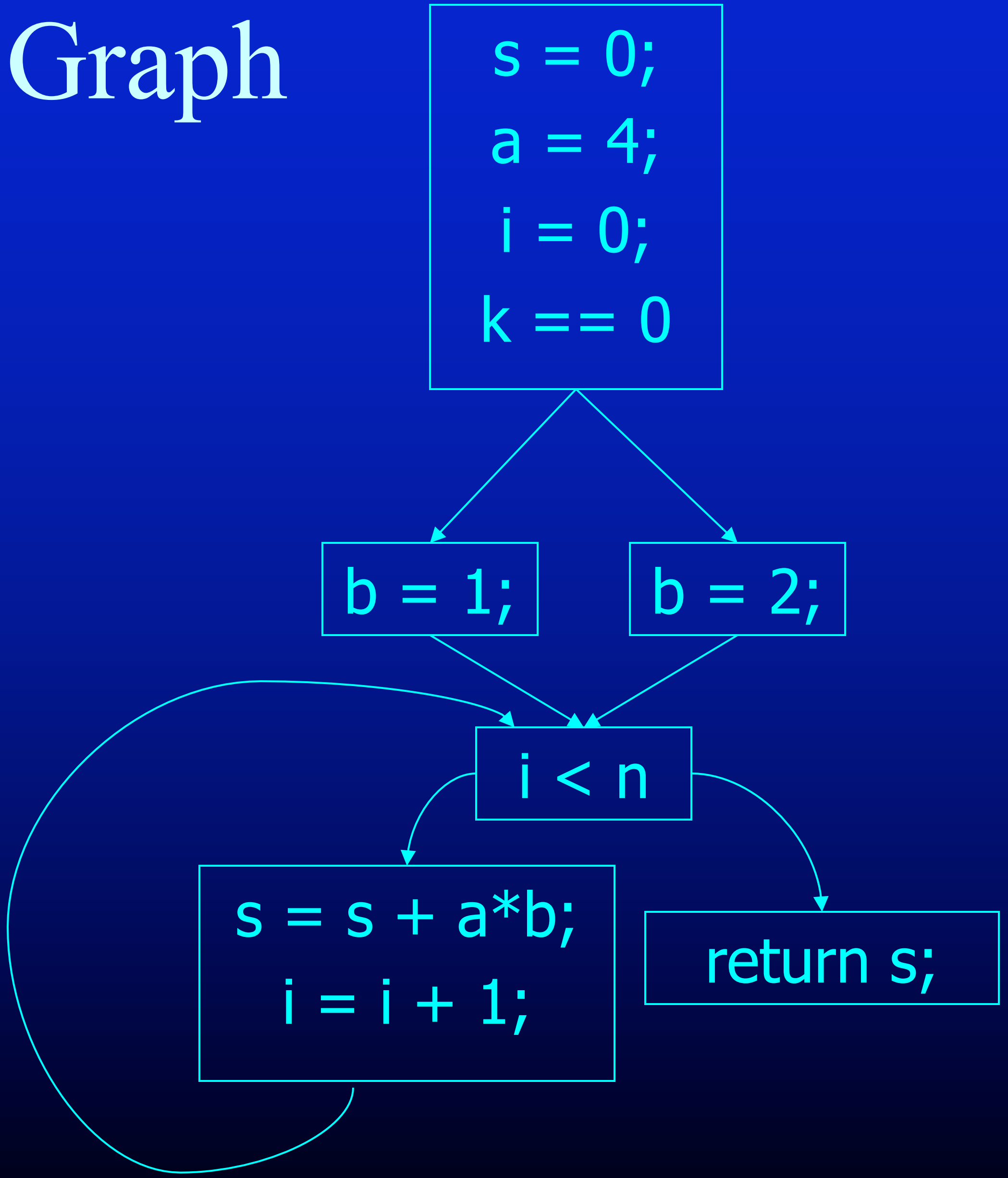
Linearizing →

Three-address code
`t1 ← 4 * z`

Two-address code
`mulq $4, %rcx`

Control Flow Graph

```
into add(n, k) {
    s = 0; a = 4; i = 0;
    if (k == 0)
        b = 1;
    else
        b = 2;
    while (i < n) {
        s = s + a*b;
        i = i + 1;
    }
    return s;
}
```



Control Flow Graph

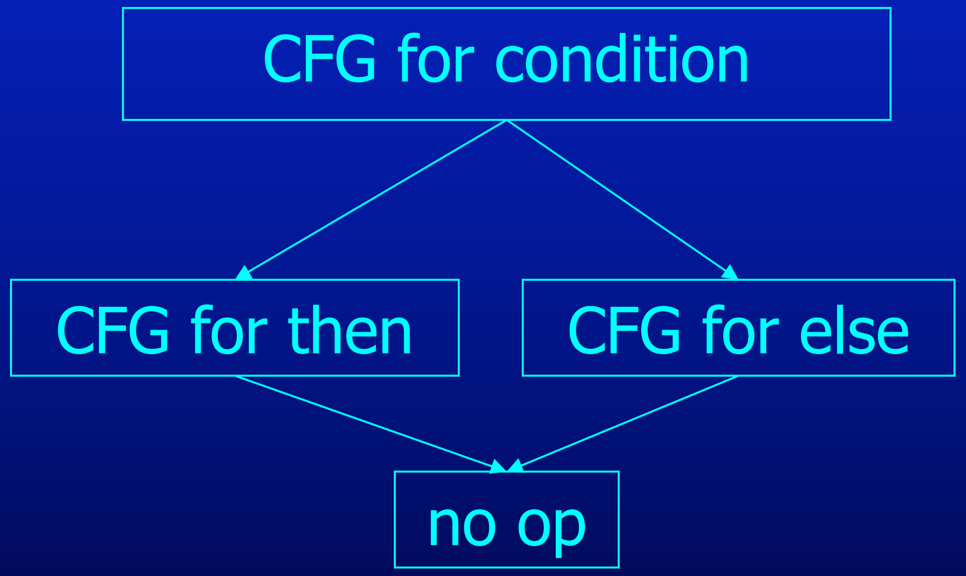
- Nodes Represent Computation
  - Each Node is a Basic Block
  - Basic Block is a Sequence of Instructions with
    - No Branches Out Of Middle of Basic Block
    - No Branches Into Middle of Basic Block
    - Basic Blocks should be maximal
  - Execution of basic block starts with first instruction
  - Includes all instructions in basic block
- Edges Represent Control Flow

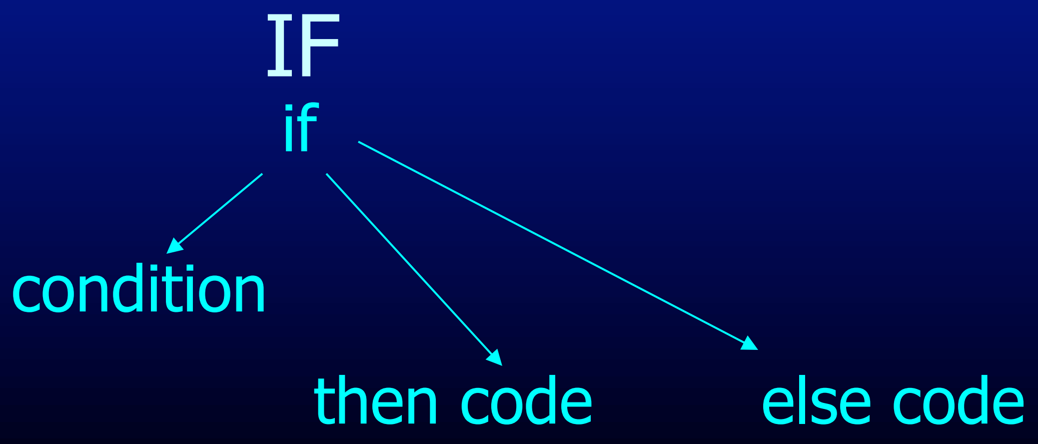
AST to CFG for If Then Else

Source Code

```
if (condition) {
  code for then
} else {
  code for else
}
```

CFG





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

For the quiz, you should know:

- What is a CFG
- What are basic blocks
- What/why of short-circuiting
- How to construct a CFG for simple programs

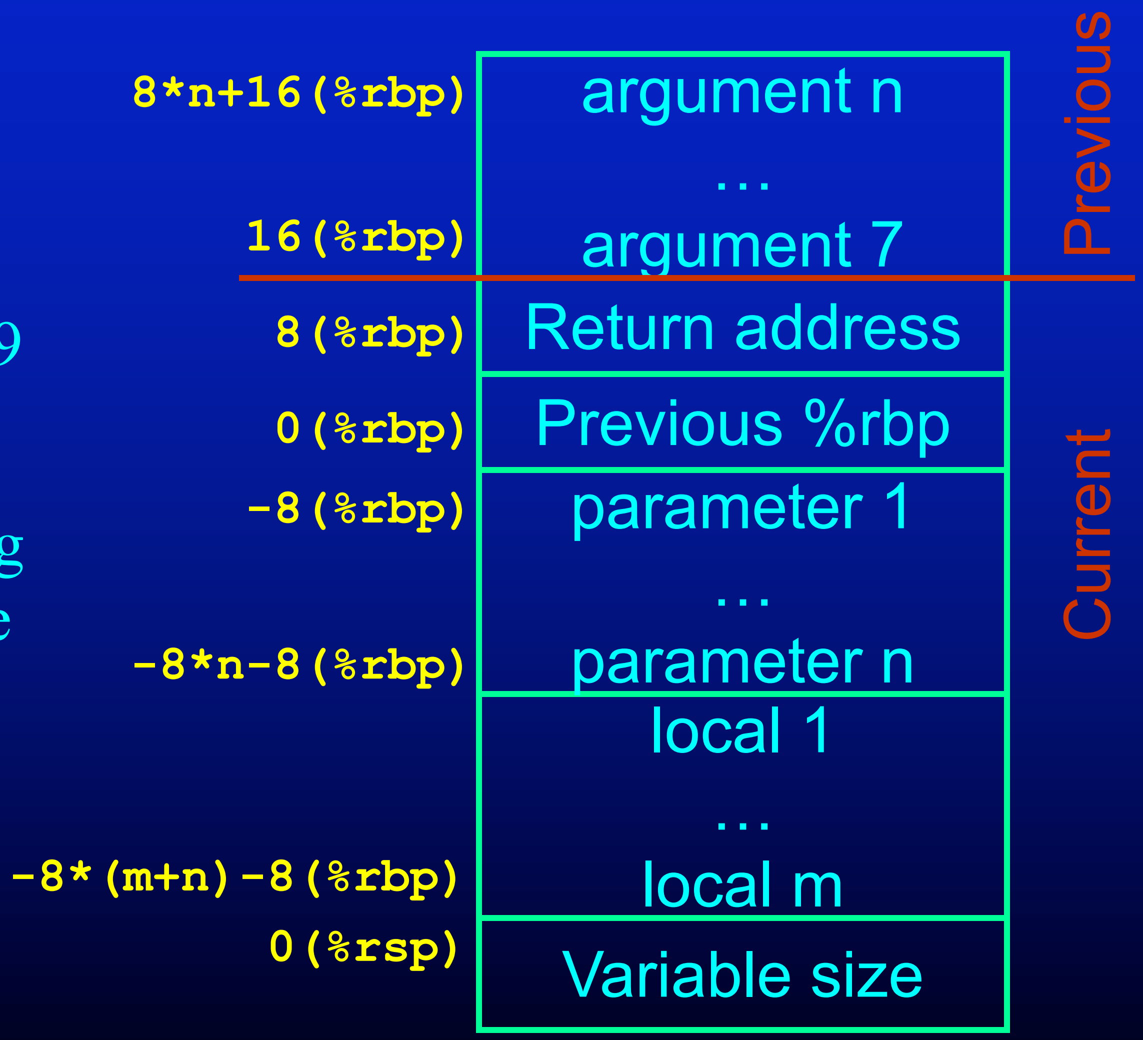
The Call Stack

- Arguments 1 to 6 are in:
  - %rdi, %rsi, %rdx,
  - %rcx, %r8, and %r9

%rbp
  - marks the beginning of the current frame

%rsp
  - marks top of stack

%rax
  - return value



Questions

- Why allocate activation records on a stack?
- Why not statically preallocate activation records?
- Why not dynamically allocate activation records in the heap?

Allocate space for parameters/locals

- Each parameter/local has its own slot on stack
- Each slot accessed via %rbp negative offset
- Iterate over parameter/local descriptors
- Assign a slot to each parameter/local

Generate procedure entry prologue

- Push base pointer (%rbp) onto stack
- Copy stack pointer (%rsp) to base pointer (%rbp)
- Decrease stack pointer by activation record size
- All done by:

  enter <stack frame size in bytes>, <lexical nesting level>

  enter $48, $0

- For now (will optimize later) move parameters to slots in activation record (top of call stack)

```
movq %rdi, -24(%rbp)
```

x86 Register Usage

- 64 bit registers (16 of them)

  %rax, %rbx, %rcx, %rdx, %rdi, %rsi, %rbp, %rsp, %r8-%r15
- Stack pointer %rsp, base pointer %rbp
- Parameters
  - First six integer/pointer parameters in %rdi, %rsi, %rdx, %rcx, %r8, %r9
  - Rest passed on the stack
- Return value
  - 64 bits or less in %rax
  - Longer return values passed on the stack

Questions

- Why have %rbp if also have %rsp?

- Why not pass all parameters in registers?
- Why not pass all parameters on stack?

- Why not pass return value in register(s) regardless of size?
- Why not pass return value on stack regardless of size?

Callee vs caller save registers

- Registers used to compute values in procedure
- Should registers have same value after procedure as before procedure?
  - Callee save registers (must have same value) %rsp, %rbx, %rbp, %r12-%r15
  - Caller save registers (procedure can change value) %rax, %rcx, %rdx, %rsi, %rdi, %r8-%r11
- Why have both kinds of registers?

Generate procedure call epilogue

- Put return value in %rax

```
mov -32(%rbp), %rax
```

- Undo procedure call
  - Move base pointer (%rbp) to stack pointer (%rsp)
  - Pop base pointer from caller off stack into %rbp
  - Return to caller (return address on stack)
  - All done by

```
leave
ret
```

For the quiz, you should know:

- Basics of x86 assembly
- General principles of memory layout (what it is, why heap grows up and stack grows down)
- General principles of calling convention
  - Why calling conventions exist, motivation for their tradeoffs
  - What callee/caller save registers are, why you want both