

IR Trees

Basic Liveness

Basic Lexing

15-411/15-611 Compiler Design

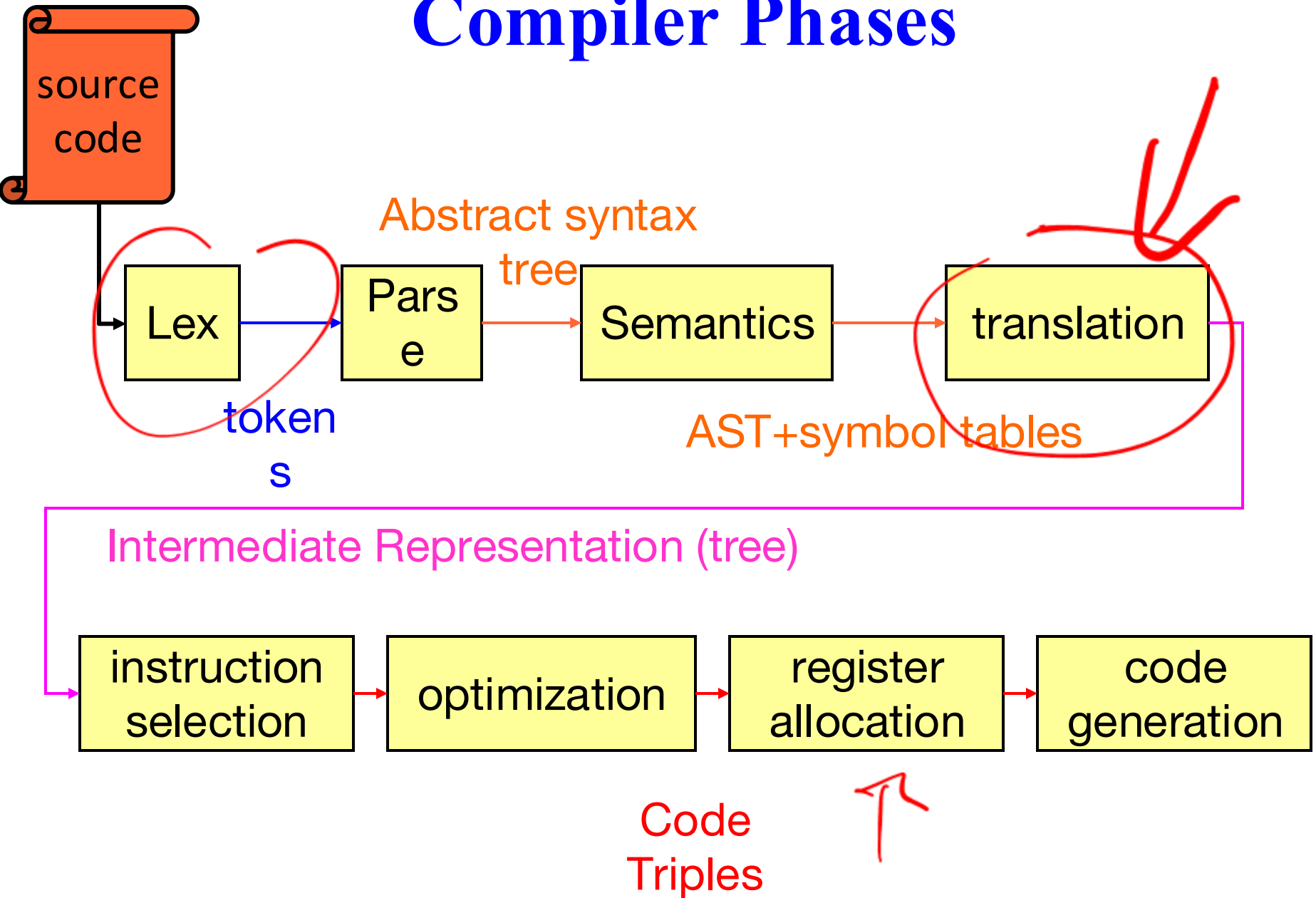
Seth Copen Goldstein

February 3, 2026

Today

- From AST $\Rightarrow$ Tree IR
- Basic Blocks
 - Liveness across Control Flow
 - Simple Lexing

Compiler Phases



Compiler Phases

Characters	⇒	Lex	⇒	Tokens
Tokens	⇒	Parse	⇒	AST
AST	⇒	Elaborate	⇒	AST
AST	⇒	Semantic Analysis	⇒	AST
AST	⇒	Translate	⇒	IR Trees
IR Trees	⇒	Munch	⇒	Abs Asm
Abs Asm	⇒	Optimize	⇒	Abs Asm
Abs Asm	⇒	Select	⇒	ASM
ASM	⇒	Reg Alloc	⇒	ASM

AST

$e ::= n \mid x \mid e_1 \oplus e_2 \mid e_1 \oslash e_2 \mid e_1 \geq e_2$
 $\mid f(e_1, \dots, e_n) \mid !e \mid e_1 \&\& e_2 \mid e_1 ? e_2 : e_3$

$S ::= \text{assign}(x, e)$
 $\mid \text{if}(e, s_1, s_2)$
 $\mid \text{decl}(x, \tau, s)$
 $\mid \text{while}(e, s)$
 $\mid \text{return } e$
 $\mid \text{nop}$
 $\mid \text{seq}(s_1, s_2)$

Translation to IR Trees

- Translate from AST into IR Trees
- Goals:
 - Isolate side-effects
 - Make order of execution explicit
 - Support optimization of pure expressions
 - Make control flow explicit
- Tree IR contains
 - pure expressions p
 - commands c
 - a program r
- Any expression which can have a side-effect must be a top-level expression.

The Tree IR

$p ::= \underline{n \mid x \mid p_1 \oplus p_2}$ Pure Expressions

$c ::= \underline{x \leftarrow p}$ Commands

| $\underline{x \leftarrow p_1 \oslash p_2}$

| $x \leftarrow f(p_1, p_2, \dots, p_n)$

| $\underline{\text{if } (p_1 \geq p_2) \text{ then } l_t \text{ else } l_f}$

| goto l

| $l:$

| return(p)

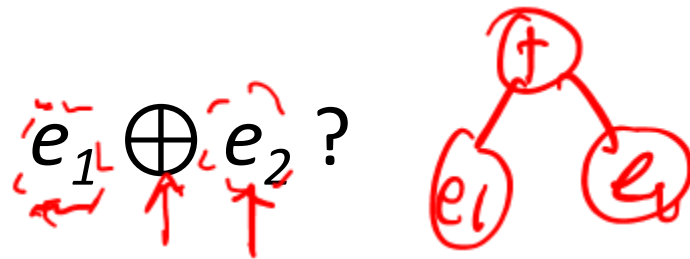
$r ::= \underline{c_1 ; \dots ; c_n}$ Programs



Translating (Integer) Expressions

$$e ::= n \mid x \mid e_1 \oplus e_2 \mid e_1 \oslash e_2 \mid e_1 \geq e_2 \\ \mid f(e_1, \dots, e_n) \mid !e \mid e_1 \&\& e_2 \mid e_1 ? e_2 : e_3$$

$n \mid x \mid e_1 \oplus e_2 ?$



$tr(e) = \langle \text{code}, p \rangle$

$e_1 \oslash e_2 ?$

```

p ::= n | x | p1 ⊕ p2
c ::= x ← p
    | x ← p1 ⊙ p2
    | x ← f(p1, p2, ..., pn)
    | if (p1 ≥ p2) then lt else lf
    | goto l
    | l:
    | return(p)
r ::= c1 ; ... ; cn
    
```


Translating (Integer) Expressions

$$e ::= n \mid x \mid e_1 \oplus e_2 \mid e_1 \oslash e_2 \mid e_1 \geq e_2 \\ \mid f(e_1, \dots, e_n) \mid !e \mid e_1 \&\& e_2 \mid e_1 ? e_2 : e_3$$

- exp becomes
 - seq of commands
 - pure-expression
- $\text{tr}(e) = \langle r, p \rangle$
 result of e is p

```

p ::= n | x | p1 ⊕ p2
c ::= x ← p
    | x ← p1 ⊙ p2
    | x ← f(p1, p2, ..., pn)
    | if (p1 ≥ p2) then lt else lf
    | goto l
    | l:
    | return(p)
r ::= c1 ; ... ; cn
    
```

Translating (Integer) Expressions

*

- $\text{tr}(n) = \langle \cdot, n \rangle$
- $\text{tr}(x) = \langle \cdot, x \rangle$
- $\text{tr}(e_1 \oplus e_2) = \langle \cdot, p_1 \oplus p_2 \rangle$

$\text{tr}(e_1) \rightarrow \langle r_1, p_1 \rangle$
 $\text{tr}(e_2) \rightarrow \langle r_2, p_2 \rangle$
 $\langle r_1, r_2, \langle p_1 \oplus p_2 \rangle \rangle$


p	$::=$	$n \mid x \mid p_1 \oplus p_2$
c	$::=$	$x \leftarrow p$
		$x \leftarrow p_1 \odot p_2$
		$x \leftarrow f(p_1, p_2, \dots, p_n)$
		if $(p_1 \geq p_2)$ then l_t else l_f
		goto l
		$l:$
		return(p)
r	$::=$	$c_1 ; \dots ; c_n$

Translating (Integer) Expressions

- $\text{tr}(n) = \langle \cdot, n \rangle$
- $\text{tr}(x) = \langle \cdot, x \rangle$
- $\text{tr}(e_1 \oplus e_2) :=$
 $\text{tr}(\underline{e_1}) = \langle r_1, p_1 \rangle$
 $\text{tr}(\underline{e_2}) = \langle r_2, p_2 \rangle$
 $\langle r_1; r_2, p_1 \oplus p_2 \rangle$

```
 $p$  ::=  $n \mid x \mid p_1 \oplus p_2$   
 $c$  ::=  $x \leftarrow p$   
       $\mid x \leftarrow p_1 \oslash p_2$   
       $\mid x \leftarrow f(p_1, p_2, \dots, p_n)$   
       $\mid \text{if } (p_1 \geq p_2) \text{ then } l_t \text{ else } l_f$   
       $\mid \text{goto } l$   
       $\mid l:$   
       $\mid \text{return}(p)$   
 $r$  ::=  $c_1; \dots; c_n$ 
```

Example

$$\text{tr}(e_1 \oplus e_2) = \text{tr}(e_1) = \langle r_1, p_1 \rangle$$

$$\text{tr}(e_2) = \langle r_2, p_2 \rangle$$

$$\langle r_1, r_2, t \leftarrow p_1 \oplus p_2, t \rangle$$

(from t)

$$x \leftarrow a / \phi$$

- $\text{tr}(n) = \langle \cdot, n \rangle$
- $\text{tr}(x) = \langle \cdot, x \rangle$
- $\text{tr}(e_1 \oplus e_2) :=$
 $\text{tr}(e_1) = \langle r_1, p_1 \rangle$
 $\text{tr}(e_2) = \langle r_2, p_2 \rangle$
 $\langle r_1, r_2, p_1 \oplus p_2 \rangle$

```

p    ::=  n | x | p1 ⊕ p2
c    ::=  x ← p
      |  x ← p1 ⊙ p2
      |  x ← f(p1, p2, ..., pn)
      |  if (p1 ≥ p2) then lt else lf
      |  goto l
      |  l:
      |  return(p)
r    ::=  c1; ... ; cn
    
```

Side Effects?

- $\text{tr}(n) = \langle \cdot, n \rangle$
- $\text{tr}(x) = \langle \cdot, x \rangle$
- $\text{tr}(e_1 \oplus e_2) :=$
 $\text{tr}(e_1) = \langle r_1, p_1 \rangle$
 $\text{tr}(e_2) = \langle r_2, p_2 \rangle$
 $\langle r_1; r_2, p_1 \oplus p_2 \rangle$
- $\text{tr}(e_1 \oslash e_2) :=$

p	$::=$	$n \mid x \mid p_1 \oplus p_2$
c	$::=$	$x \leftarrow p$
	$ $	$x \leftarrow p_1 \oslash p_2$
	$ $	$x \leftarrow f(p_1, p_2, \dots, p_n)$
	$ $	if $(p_1 \geq p_2)$ then l_t else l_f
	$ $	goto l
	$ $	$l:$
	$ $	return(p)
r	$::=$	$c_1 ; \dots ; c_n$

Side Effects?

- $\text{tr}(n) = \langle \cdot, n \rangle$
- $\text{tr}(x) = \langle \cdot, x \rangle$
- $\text{tr}(e_1 \oplus e_2) :=$

$$\text{tr}(e_1) = \langle r_1, p_1 \rangle$$

$$\text{tr}(e_2) = \langle r_2, p_2 \rangle$$

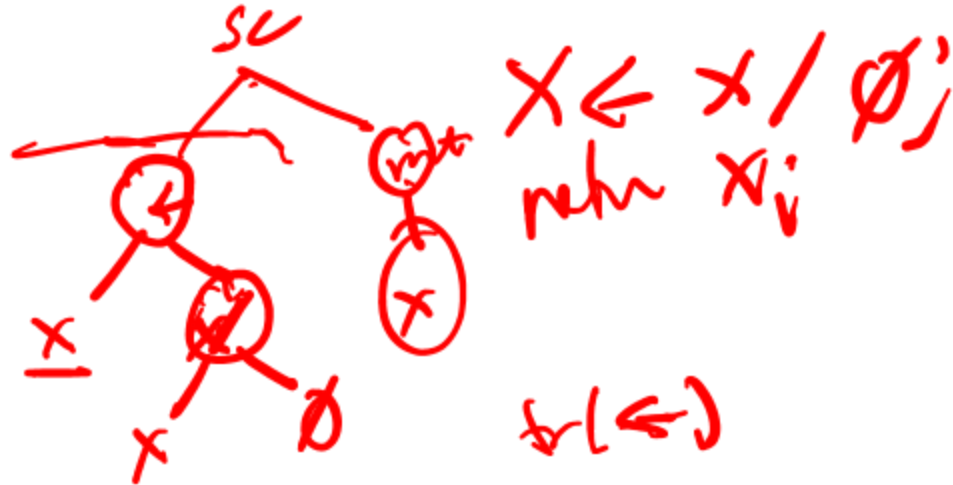
$$\langle r_1; r_2, p_1 \oplus p_2 \rangle$$

- $\text{tr}(e_1 \oslash e_2) :=$

$$\text{tr}(e_1) = \langle r_1, p_1 \rangle$$

$$\text{tr}(e_2) = \langle r_2, p_2 \rangle$$

$$\langle r_1; r_2; t \leftarrow p_1 \oslash p_2, t \rangle \quad (t \text{ fresh})$$



Translating (Integer) Expressions

- $\text{tr}(n) = \langle \cdot, n \rangle$
- $\text{tr}(x) = \langle \cdot, x \rangle$
- $\text{tr}(e_1 \oplus e_2) :=$
 $\text{tr}(e_1) = \langle r_1, p_1 \rangle$
 $\text{tr}(e_2) = \langle r_2, p_2 \rangle$
 $\langle r_1; r_2, p_1 \oplus p_2 \rangle$
- $\text{tr}(e_1 \odot e_2) :=$
 $\text{tr}(e_1) = \langle r_1, p_1 \rangle$
 $\text{tr}(e_2) = \langle r_2, p_2 \rangle$ (t fresh)
 $\langle r_1; r_2; t \leftarrow p_1 \odot p_2, t \rangle$

p	$::=$	$n \mid x \mid p_1 \oplus p_2$
c	$::=$	$x \leftarrow p$
		$\mid x \leftarrow p_1 \odot p_2$
		$\mid \underline{x \leftarrow f(p_1, p_2, \dots, p_n)}$
		$\mid \text{if } (p_1 \geq p_2) \text{ then } l_t \text{ else } l_f$
		$\mid \text{goto } l$
		$\mid l:$
		$\mid \text{return}(p)$
r	$::=$	$c_1; \dots; c_n$

- $\text{tr}(\underbrace{f(e_1, \dots, e_n)}_{f(e_1, e_2)}) =$

$$\begin{aligned} &\text{tr}(e_1) = \langle r_1, p_1 \rangle \\ &\text{tr}(e_2) = \langle r_2, p_2 \rangle \\ &\langle r_1; r_2; t \leftarrow f(p_1, p_2), t \rangle \end{aligned} \quad \text{fresh } t$$

Why call it Tree IR?

- $\text{tr}(n) = \langle \cdot, n \rangle$
- $\text{tr}(x) = \langle \cdot, x \rangle$
- $\text{tr}(e_1 \oplus e_2) :=$
 $\text{tr}(e_1) = \langle r_1, p_1 \rangle$
 $\text{tr}(e_2) = \langle r_2, p_2 \rangle$
 $\langle r_1; r_2, p_1 \oplus p_2 \rangle$
- $\text{tr}(e_1 \oslash e_2) :=$
 $\text{tr}(e_1) = \langle r_1, p_1 \rangle$
 $\text{tr}(e_2) = \langle r_2, p_2 \rangle$
 $\langle r_1; r_2; t \leftarrow p_1 \oslash p_2, t \rangle \quad (t \text{ fresh})$
- $\text{tr}(f(e_1, \dots, e_n)) =$
 $\text{tr}(e_1) = \langle r_1, p_1 \rangle$
 $\text{tr}(e_n) = \langle r_n, p_n \rangle$
 $\langle r_1; r_2; t \leftarrow f(p_1, \dots, p_n), t \rangle \quad (t \text{ fresh})$

What next?

$e ::= n \mid x \mid e_1 \oplus e_2 \mid e_1 \oslash e_2 \mid e_1 \geq e_2$
 $\mid f(e_1, \dots, e_n) \mid !e \mid e_1 \&\& e_2 \mid e_1 ? e_2 : e_3$

$s ::=$ assign(x, e)
 $\mid$ if(e, s_1, s_2)
 $\mid$ decl(x, τ, s)
 $\mid$ while (e, s)
 $\mid$ return e
 $\mid$ nop
 $\mid$ seq(s_1, s_2)



$fr(x) = \langle a, x \rangle$
 $fr(e) = \langle r, P \rangle$
 $\langle r, x \leftarrow P \rangle$

$fr(s_1) = \langle r_1, \tau \rangle$
 $fr(s_2) = \langle r_2, \tau \rangle$
 $\langle r_1, \tau_1 \rangle$

Statements

$\text{tr}(\text{assign}(x, e)) :=$

$\text{tr}(\text{if}(e, s_1, s_2))$

$\text{tr}(\text{decl}(x, \tau, s))$

$\text{tr}(\text{while } (e, s))$

$\text{tr}(\text{return } e) :=$

$\text{tr}(\text{nop}) :=$

$\text{tr}(\text{seq}(s_1, s_2)) :=$

p	$::=$	$n \mid x \mid p_1 \oplus p_2$
c	$::=$	$x \leftarrow p$
	$ $	$x \leftarrow p_1 \odot p_2$
	$ $	$x \leftarrow f(p_1, p_2, \dots, p_n)$
	$ $	$\text{if } (p_1 \geq p_2) \text{ then } l_t \text{ else } l_f$
	$ $	$\text{goto } l$
	$ $	$l:$
	$ $	$\text{return}(p)$
r	$::=$	$c_1 ; \dots ; c_n$

Statements

$$\begin{aligned} \text{tr}(\text{assign}(x,e)) &:= \text{tr}(e_1) = \langle r_1, p_1 \rangle \\ &\quad \langle r_1; x \leftarrow p_1 \rangle \end{aligned}$$

$$\text{tr}(\text{if}(e,s_1,s_2))$$

$$\text{tr}(\text{decl}(x, \tau, s))$$

$$\text{tr}(\text{while } (e,s))$$

$$\begin{aligned} \text{tr}(\text{return } e) &:= \text{tr}(e_1) = \langle r_1, p_1 \rangle \\ &\quad \langle r_1; \text{return}(p_1) \rangle \end{aligned}$$

$$\text{tr}(\text{nop}) := .$$

$$\text{tr}(\text{seq}(s_1,s_2)) := \langle \text{tr}(s_1); \text{tr}(s_2) \rangle$$

take 1: $\text{tr}(\text{if}(e, s_1, s_2))$

$\text{tr}(\text{if}(e, s_1, s_2)) :=$

$\text{tr}(e) = \langle r, p \rangle$

if $(p \neq 0)$ then l_t else l_f

fresh($l_t, l_f, l_{\text{done}}$)

l_t

$\text{tr}(s_1); \text{goto } l_{\text{done}}$

l_f

$\text{tr}(s_2); \text{goto } l_{\text{done}}$

l_{done}

```

p ::= n | x | p1 ⊕ p2
x ::= x ← p
    | x ← p1 ⊙ p2
    | x ← f(p1, p2, ..., pn)
    | if (p1 ≥ p2) then lt else lf
    | goto l
    | l:
    | return(p)
r ::= c1 ; ... ; cn
    
```

take 1: tr(if(e, s_1, s_2))

$\text{tr}(\text{if}(e, s_1, s_2)) :=$

$\text{tr}(e) = \langle r, p \rangle$

$\langle r ;$

if ($p \neq 0$) then l_t else $l_f ;$

$l_t : s_1 ; \text{goto } l_3 ;$

$l_f : s_2 ; \text{goto } l_3 ;$

$l_3 : >$

$(l_t, l_f, l_3 \text{ fresh})$

take 1: $\text{tr}(\text{if}(e, s_1, s_2))$

$\text{tr}(\text{if}(e, s_1, s_2)) :=$

$\text{tr}(e) = \langle r, p \rangle$

$\langle r ;$

if $(p \neq 0)$ then l_t else $l_f ;$

$l_t : s_1 ; \text{goto } l_3 ;$

$l_f : s_2 ; \text{goto } l_3 ;$

$l_3 : \rangle$

$(l_t, l_f, l_3 \text{ fresh})$

tree-IR is in basic block form

take 1: $\text{tr}(\text{if}(e, s_1, s_2))$

$\text{tr}(\text{if}(e, s_1, s_2)) :=$

$\text{tr}(e) = \langle r, p \rangle$

$\langle r ;$

if $(p \neq 0)$ then l_t else $l_f ;$

$l_t : s_1 ; \text{goto } l_3 ;$

$l_f : s_2 ; \text{goto } l_3 ;$

$l_3 : >$

And yet,
something feels
wrong here.

$(l_t, l_f, l_3 \text{ fresh})$

$$\text{cp}(b, l_t, l_f) = \langle r \rangle$$

$$\text{cp}(b, l_t, l_f) = \langle r \rangle$$

where the last command in r is
a goto to either l_t or l_f based on b

$$\text{cp}(e_1 \geq e_2, l_t, l_f)$$

$$\text{cp}(! e, l_t, l_f)$$

$$\text{cp}(e_1 \&\& e_2, l_t, l_f)$$

$$\text{cp}(0, l_t, l_f)$$

$$\text{cp}(1, l_t, l_f)$$

$$\text{cp}(e, l_t, l_f)$$

$$\begin{aligned} \text{tr}(e_1) &= \langle r_1 \rangle \\ \text{tr}(e_2) &= \langle r_2 \rangle \end{aligned}$$

Translating Boolean Expressions

$\text{cp}(\underline{e_1} \geq e_2, l_t, l_f) := \text{tr}(\underline{e_1}) = \langle r_1, p_1 \rangle$
 $\text{tr}(\underline{e_2}) = \langle r_2, p_2 \rangle$
 $\langle r_1 ; r_2 ; \text{if } (p_1 \geq p_2) \text{ then } l_t \text{ else } l_f ;$

$\text{cp}(! e, l_t, l_f) := \text{cp}(e, l_f, l_t)$

$\text{cp}(\underline{e_1} \&\& \underline{e_2}, l_t, l_f) := \text{cp}(\underline{e_1}, l_{\text{fresh}}, l_f) ;$

$l_{\text{fresh}} : \text{cp}(\underline{e_2}, l_t, l_f) \quad \text{--- } l_{\text{fresh}} \text{ ---}$

$\text{cp}(\underline{0}, l_t, l_f) := \text{goto } l_f$

$\text{cp}(\underline{1}, l_t, l_f) := \text{goto } l_t$

$\text{cp}(\underline{e}, l_t, l_f) := \text{tr}(\underline{e}) = \langle r, p \rangle$

$\langle r ; \text{if } (p \neq 0) \text{ then } l_t \text{ else } l_f \rangle$

Translating Boolean Expressions

$$\begin{aligned}
 \text{cp}(e_1 \geq e_2, l_v, l_f) &:= \text{tr}(e_1) = \langle r_1, p_1 \rangle \\
 &\quad \text{tr}(e_2) = \langle r_2, p_2 \rangle \\
 &\quad \langle r_1 ; r_2 ; \text{if } (p_1 \geq p_2) \text{ then } l_t \text{ else } l_f ; \\
 \text{cp}(! e, l_v, l_f) &:= \text{cp}(e, l_f, l_t) \\
 \text{cp}(e_1 \&\& e_2, l_v, l_f) &:= \text{cp}(e_1, l_{\text{fresh}}, l_f) ; \\
 &\quad l_{\text{fresh}} : \text{cp}(e_2, l_v, l_f) \\
 \text{cp}(0, l_v, l_f) &:= \text{goto } l_f \\
 \text{cp}(1, l_v, l_f) &:= \text{goto } l_t \\
 \text{cp}(e, l_v, l_f) &:= \text{tr}(e) = \langle r, p \rangle \\
 &\quad \langle r ; \text{if } (p \neq 0) \text{ then } l_t \text{ else } l_f \rangle
 \end{aligned}$$

$\text{tr}(e) =$

Translating Boolean Expressions

?

$\text{tr}(e) = \langle \text{cp}(e, l_t, l_f) ;$

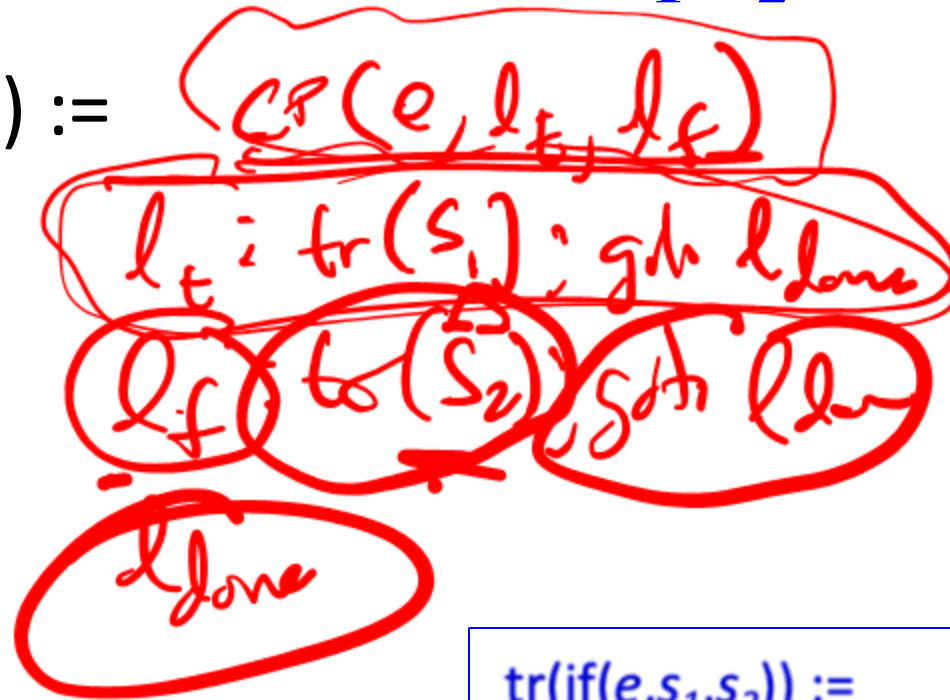
$l_t: t \leftarrow 1 ; \text{goto } l_{\text{done}} ;$

$l_f: t \leftarrow 0 ; \text{goto } l_{\text{done}} ;$ $(l_t, l_f, l_{\text{done}} \text{ fresh})$

$l_{\text{done}}: \rangle$

take 2: $\text{tr}(\text{if}(e, s_1, s_2))$

$\text{tr}(\text{if}(e, s_1, s_2)) :=$



fresh l_t, l_f

$\text{tr}(\text{if}(e, s_1, s_2)) :=$

$\text{tr}(e) = \langle r, p \rangle$

$\langle r;$

if $(p \neq 0)$ then l_t else l_f ;

$l_t: s_1; \text{goto } l_3;$

$l_f: s_2; \text{goto } l_3;$

$l_3: >$

$(l_t, l_f, l_3 \text{ fresh})$

take 2: $\text{tr}(\text{if}(e, s_1, s_2))$

$\text{tr}(\text{if}(e, s_1, s_2)) :=$

$< \text{cp}(e, l_t, l_f) ;$

$l_t : s_1 ; \text{goto } l_3 ;$

$l_f : s_2 ; \text{goto } l_3 ;$

$l_3 : >$

$(l_t, l_f, l_3 \text{ fresh})$

Liveness Analysis

- An example of a dataflow analysis
- There are many different dataflow analysis.
- Liveness is an example of a **backward, may** analysis
- We will see many others later on.
- Today we just extend to handle control flow

Computing liveness

v	←	1	{ }
w	←	v + 3	{ }
x	←	w + v	{ }
u	←	v	{ }
s	←	u + x	{ }
	←	w	{ }
	←	s	{ }
	←	u	{ }

If **t** is used at
point **p**, then it is
live immediately
before **p**

Computing liveness

v	←	1	{ }
w	←	v + 3	{ v }
x	←	w + v	{ w, v }
u	←	v	{ v }
s	←	u + x	{ u, x }
	←	w	{ w }
	←	s	{ s }
	←	u	{ u }

If **t** is used at point **p**, then it is live **immediately** before **p**

Computing liveness

$v \leftarrow 1$	$\{ \}$
$w \leftarrow v + 3$	$\{ v \}$
$x \leftarrow w + v$	$\{ w, v \}$
$u \leftarrow v$	$\{ v \}$
$s \leftarrow u + x$	$\{ u, x \}$
$\leftarrow w$	$\{ w \}$
$\leftarrow s$	$\{ s \}$
$\leftarrow u$	$\{ u \}$

If t is used at point p , then it is live **immediately** before p

What about before p ?

Computing liveness

$v \leftarrow 1$	$\{ \}$
$w \leftarrow v + 3$	$\{ v \}$
$x \leftarrow w + v$	$\{ w, v \}$
$u \leftarrow v$	$\{ v \}$
$s \leftarrow u + x$	$\{ u, x \}$
$\leftarrow w$	$\{ w \}$
$\leftarrow s$	$\{ s \}$
$\leftarrow u$	$\{ u \}$

If t is used at point p , then it is live **immediately** before p

t is live at p if:

- it is live after p & it is NOT defined at p
- it is used in p

Computing liveness

v ← 1	{ }
w ← v + 3	{ v }
x ← w + v	{ w , v }
u ← v	{ v }
s ← u + x	{ u , x }
← w	{ w }
← s	{ s , u }
← u	{ u }

If **t** is used at point **p**, then it is live **immediately** before **p**

t is live at **p** if:

- it is live after **p** & it is NOT defined at **p**
- it is used in **p**

Computing liveness

$v \leftarrow 1$	$\{ \}$
$w \leftarrow v + 3$	$\{ v \}$
$x \leftarrow w + v$	$\{ w, v \}$
$u \leftarrow v$	$\{ v \}$
$s \leftarrow u + x$	$\{ u, x \}$
$\leftarrow w$	$\{ w, s, u \}$
$\leftarrow s$	$\{ s, u \}$
$\leftarrow u$	$\{ u \}$

If t is used at point p , then it is live **immediately** before p

t is live at p if:

- it is live after p & it is NOT defined at p
- it is used in p

Computing liveness

$v \leftarrow 1$	$\{ \}$
$w \leftarrow v + 3$	$\{ v \}$
$x \leftarrow w + v$	$\{ w, v \}$
$u \leftarrow v$	$\{ v \}$
$s \leftarrow u + x$	$\{ u, x \}$
$\leftarrow w$	$\{ w, s, u \}$
$\leftarrow s$	$\{ s, u \}$
$\leftarrow u$	$\{ u \}$

If t is used at point p , then it is live **immediately** before p

t is live at p if:

- it is live after p & it is NOT **defined at** p
- it is used in p

Computing liveness

$v \leftarrow 1$	$\{ \}$
$w \leftarrow v + 3$	$\{ v \}$
$x \leftarrow w + v$	$\{ w, v \}$
$u \leftarrow v$	$\{ v \}$
$s \leftarrow u + x$	$\{ u, x, w \}$
$\leftarrow w$	$\{ w, s, u \}$
$\leftarrow s$	$\{ s, u \}$
$\leftarrow u$	$\{ u \}$

If t is used at point p , then it is live **immediately** before p

t is live at p if:

- it is live after p & it is NOT defined at p
- it is used in p

Computing liveness

$v \leftarrow 1$	$\{ \}$
$w \leftarrow v + 3$	$\{ v \}$
$x \leftarrow w + v$	$\{ w, v \}$
$u \leftarrow v$	$\{ v, x, w \}$
$s \leftarrow u + x$	$\{ u, x, w \}$
$\leftarrow w$	$\{ w, s, u \}$
$\leftarrow s$	$\{ s, u \}$
$\leftarrow u$	$\{ u \}$

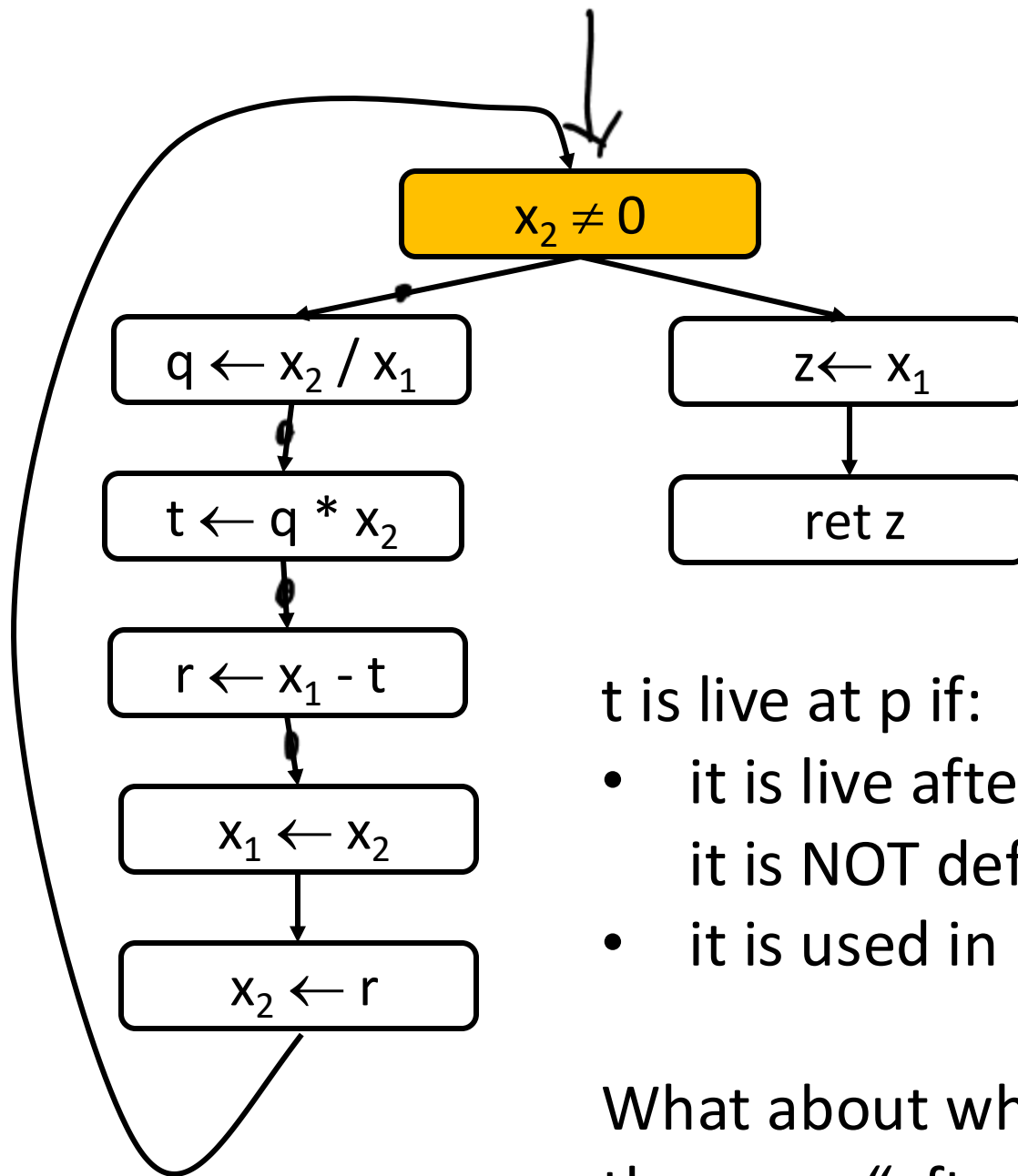
If t is used at point p , then it is live **immediately** before p

t is live at p if:

- it is live after p & it is NOT defined at p
- it is used in p

What about control flow?

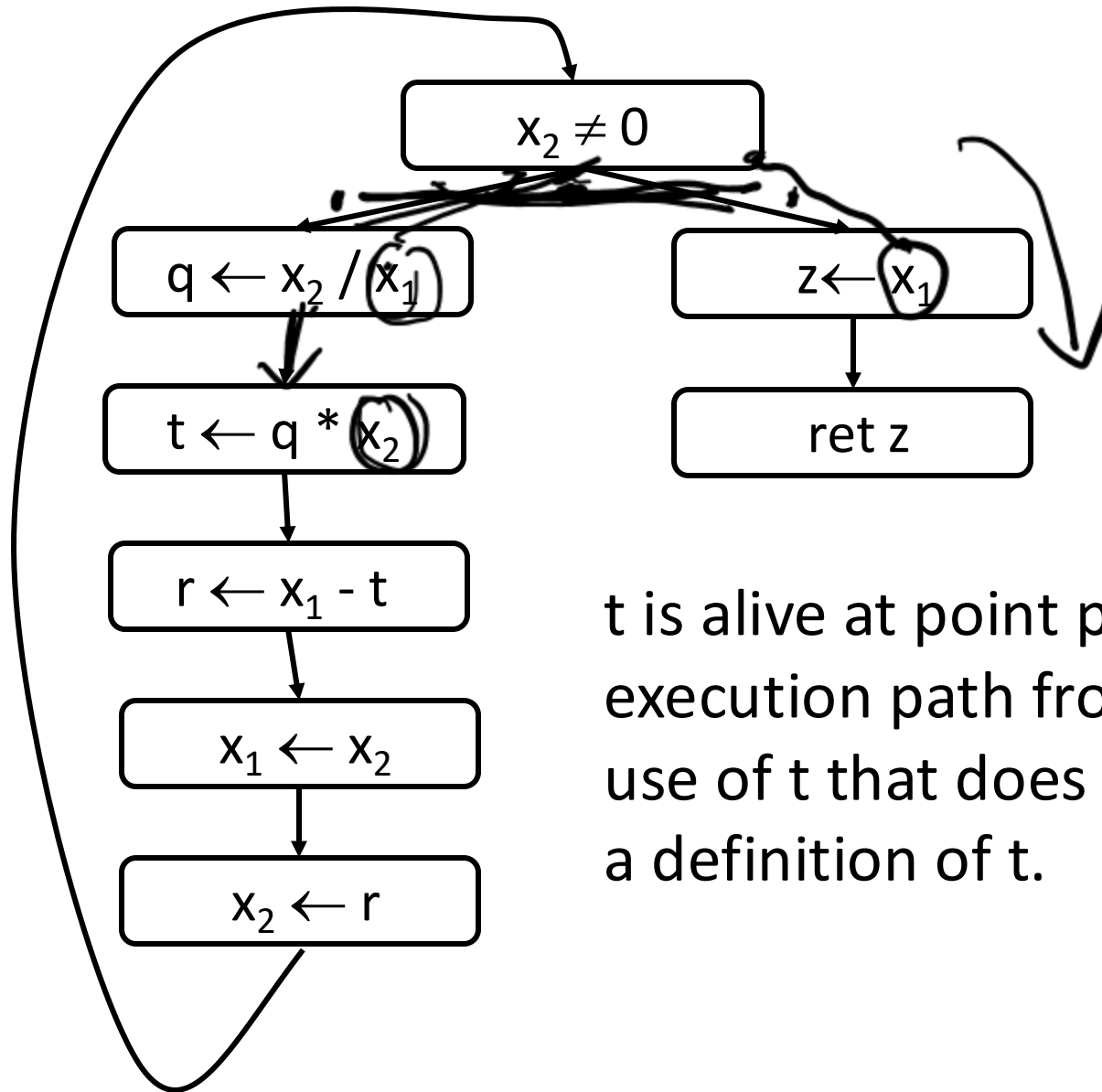
- We need to extend ideas to programs with control flow.
- Use Control Flow Graph (CFG) to represent the program
 - Nodes: program points, entry, exit
 - Edges: $(u,v) \in G$ if control can potentially go from u to v



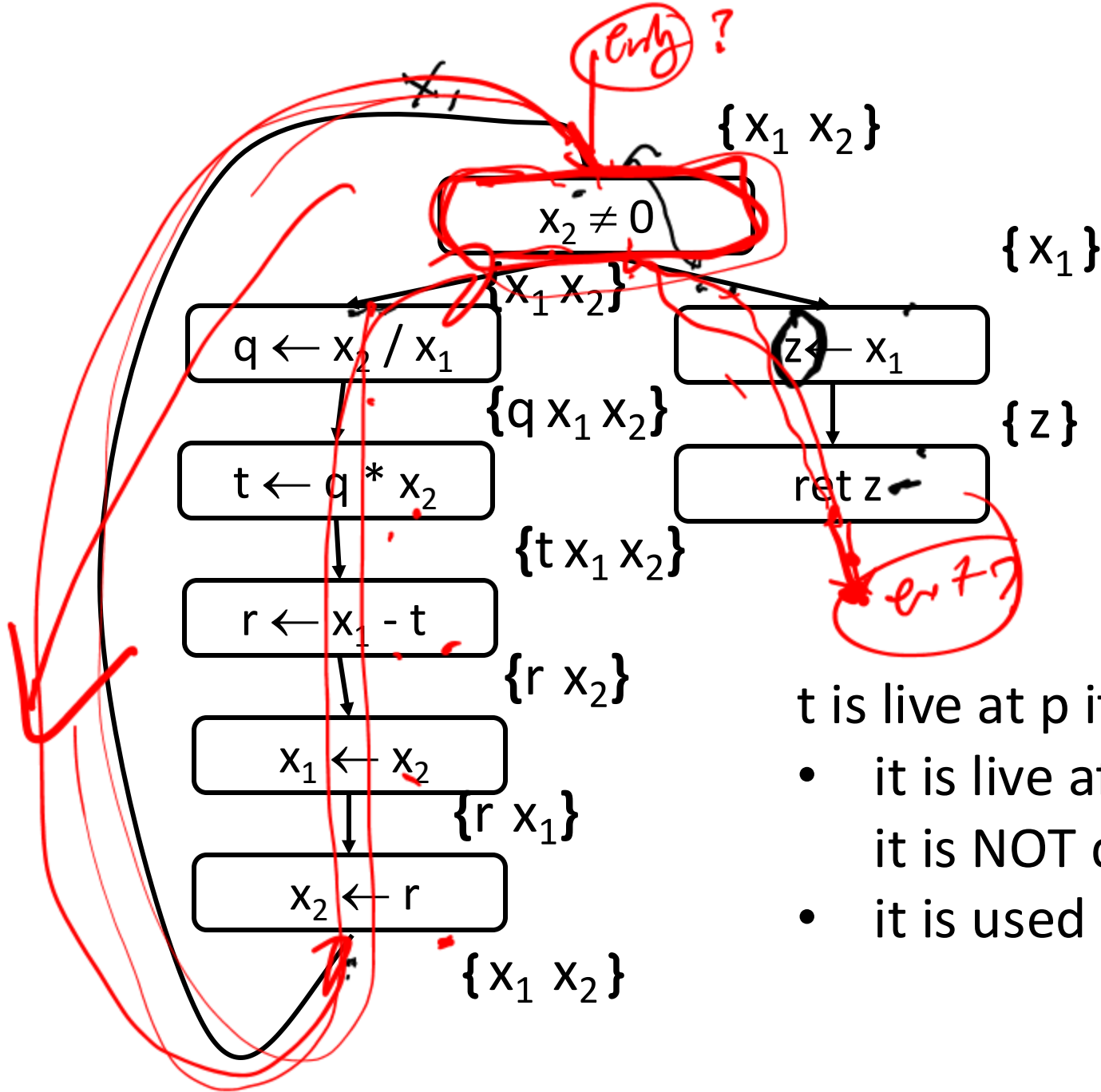
t is live at p if:

- it is live after p & it is NOT defined at p
- it is used in p

What about when there is more than one “after p ”?



t is alive at point p if there is an execution path from p to some use of t that does not go through a definition of t .



t is live at p if:

- it is live after p & it is NOT defined at p
- it is used in p

Liveness Analysis

- Each point in program has a liveIn set and a liveOut set.
- Each point either adds to the set (any uses)
- Or removes from the set (any defs)

$$\left\{ \begin{array}{l} \underline{\text{In}(p)} = \underline{\text{Uses}(p)} \cup (\underline{\text{Out}(p)} - \underline{\text{Defs}(p)}) \\ \text{Out}(p) = \bigcup_{s \in \underline{\text{Succ}(p)}} \text{In}(s) \end{array} \right.$$

Handwritten diagram illustrating the liveness analysis equations. A node p is shown with a box representing its state. Red arrows and labels indicate the flow of information: $\text{Uses}(p)$ points to the box, $\text{Defs}(p)$ points away from the box, and $\text{Succ}(p)$ points to the box. The box contains a red 'x' and a red 'y'.

Algorithm

forall nodes, $n \in \text{CFG}$

$$\text{In}(n) = \text{Out}(n) = \{\}$$

Until no changes in In or Out sets:

forall nodes, $p \in \text{CFG}$

$$\text{In}(p) = \text{Uses}(p) \cup (\text{Out}(p) - \text{Defs}(p))$$

$$\text{Out}(p) = \bigcup_{s \in \text{Succ}(p)} \text{In}(s)$$

- Does this terminate?
- Practically, what is best order to visit nodes?

Alternative: Worklist Approach

forall nodes, $n \in \text{CFG}$

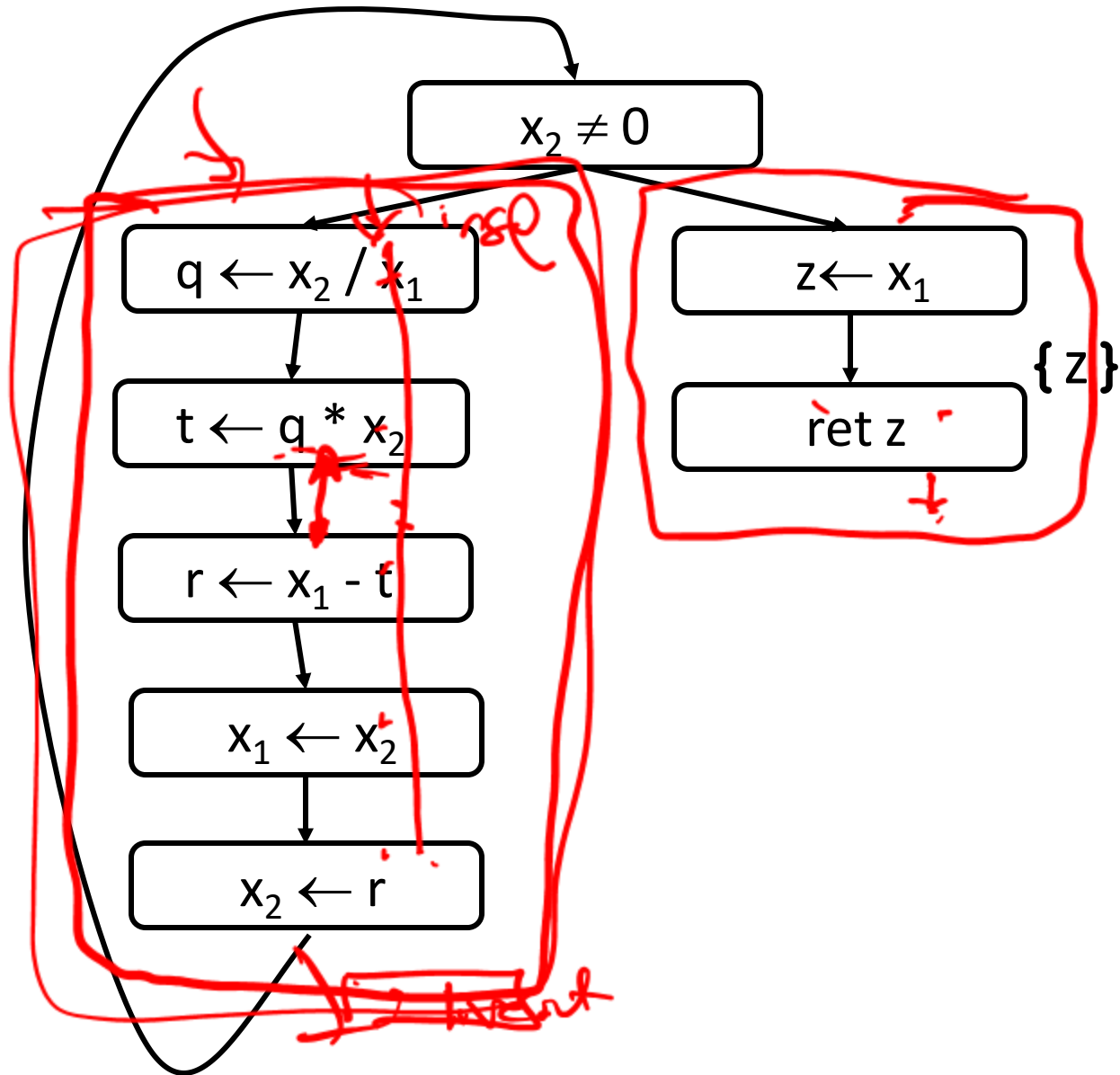
$\text{In}(n) = \{\}$

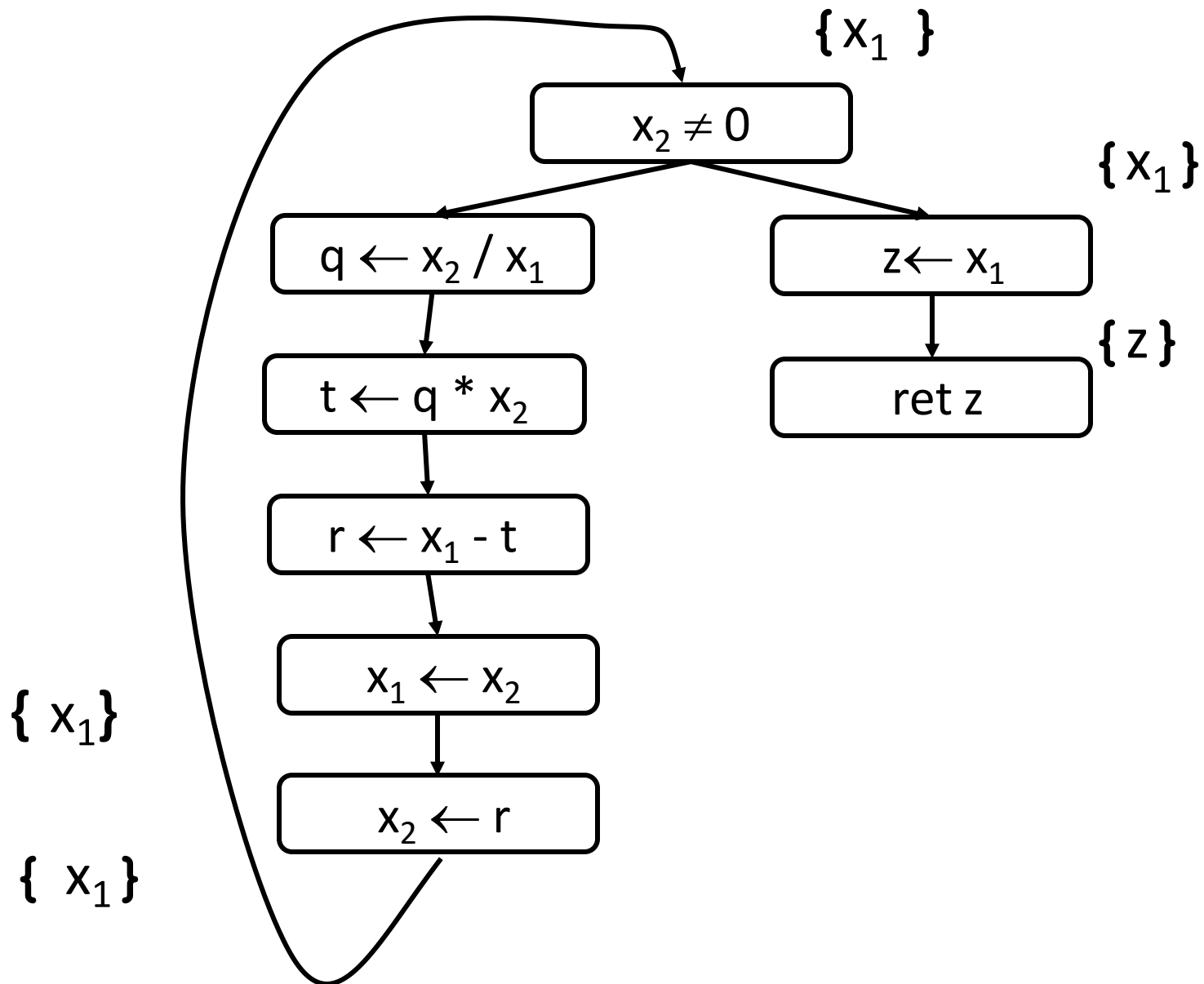
forall nodes, $p \in \text{CFG}$:

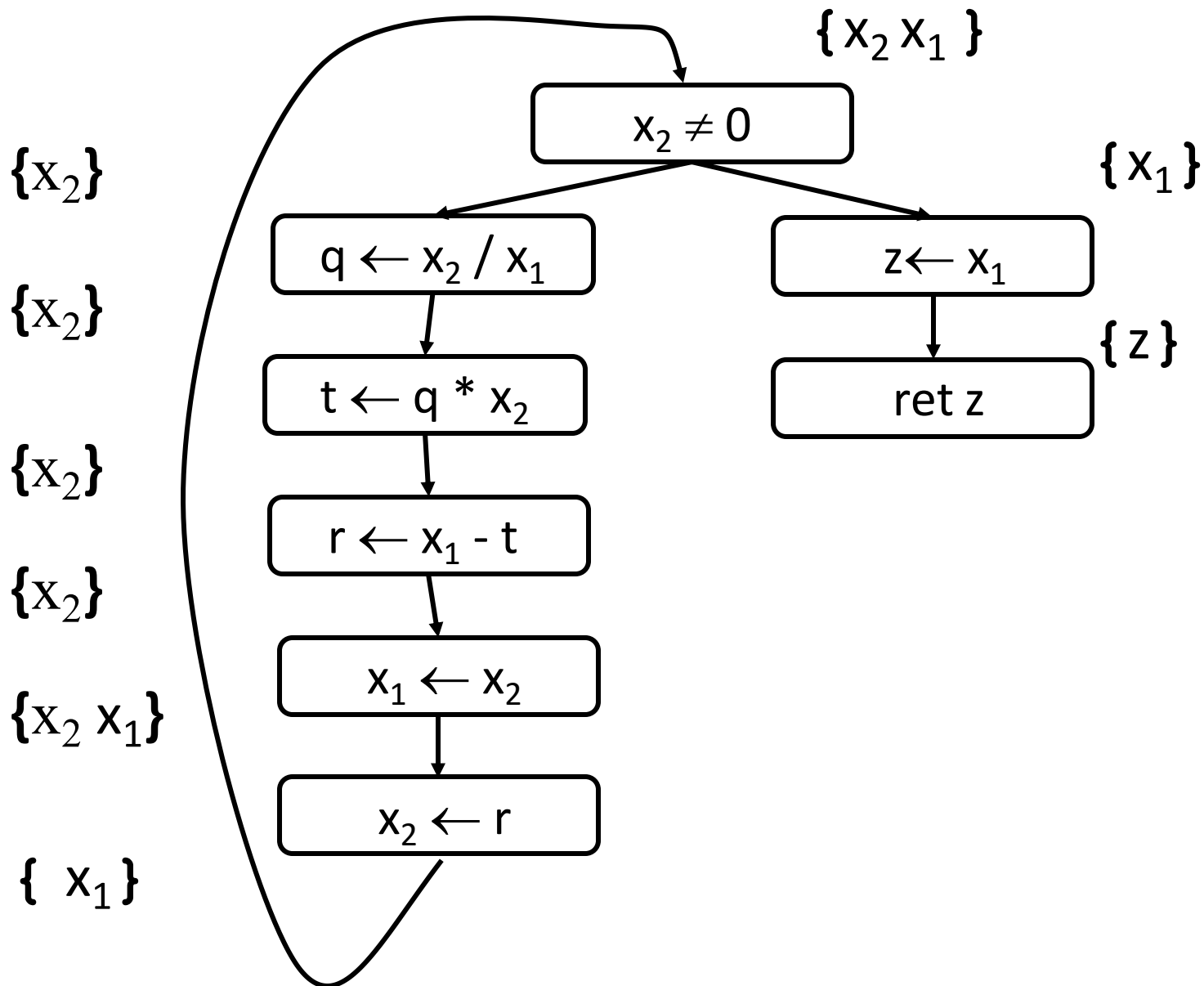
if $v \in \text{Uses}(p) - \text{Defs}(p)$ not in $\text{In}(p)$

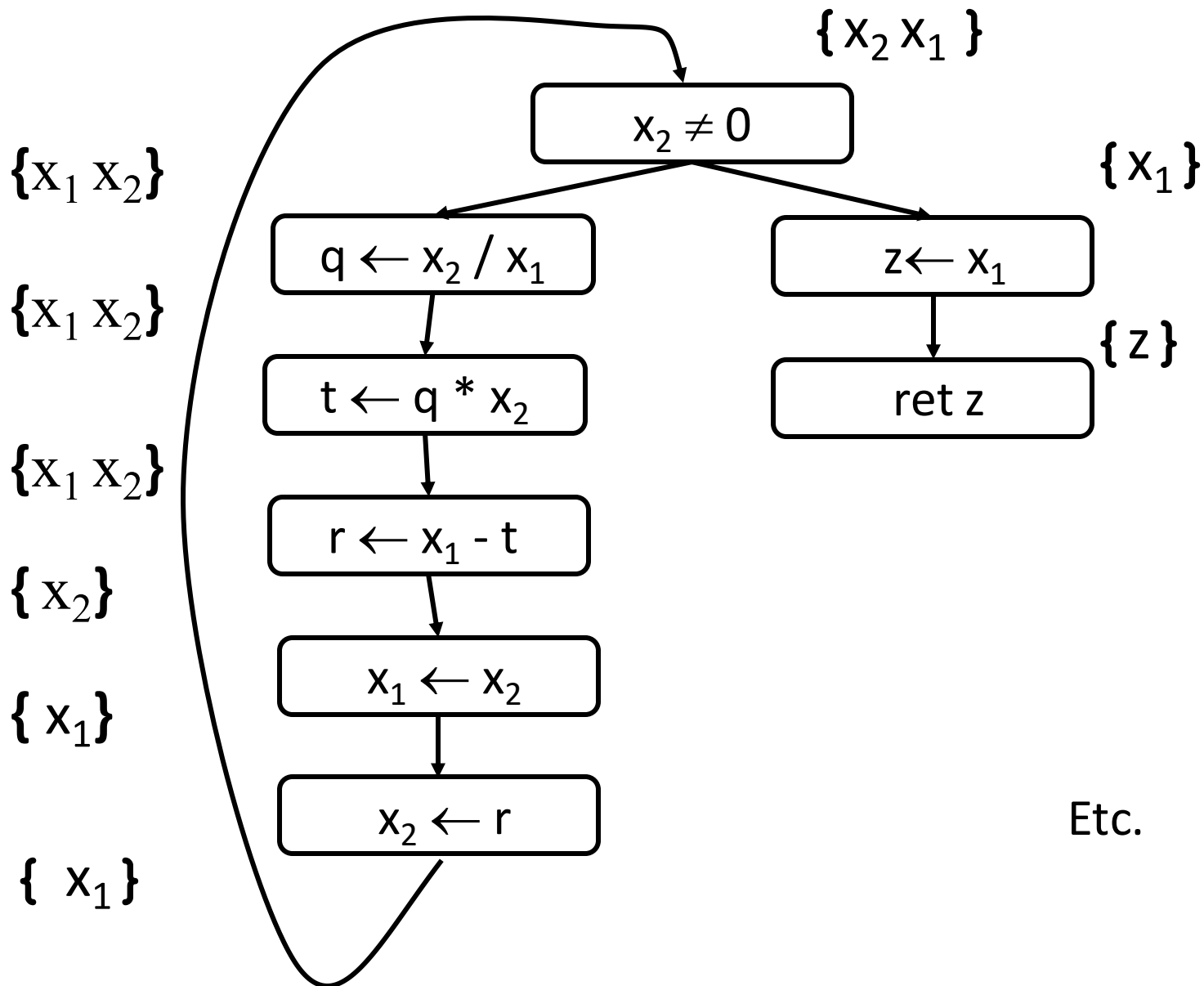
$\text{In}(p) \cup \{v\}$

propagate to all pred(p) until v is
defined or v is marked livein.

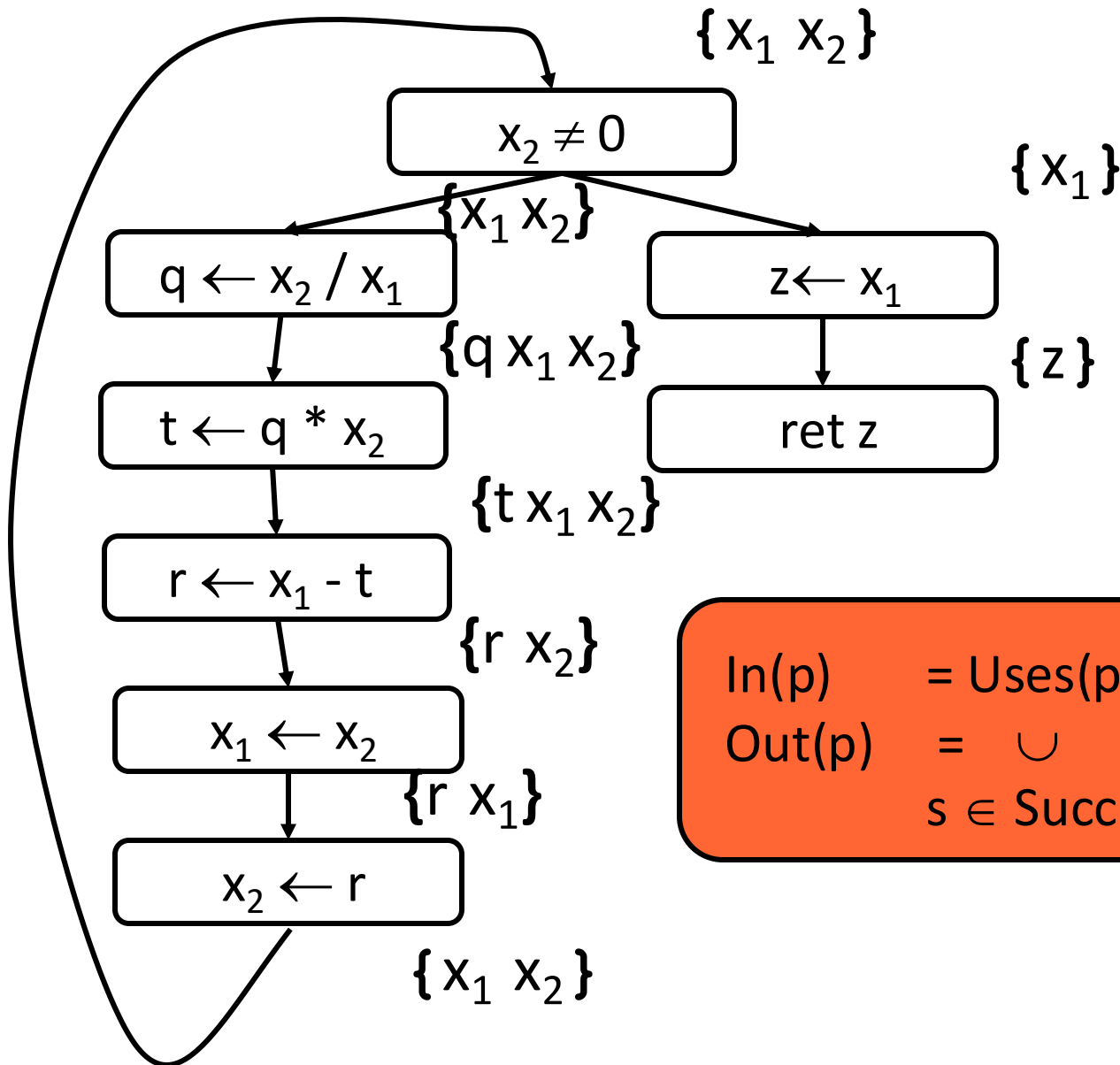






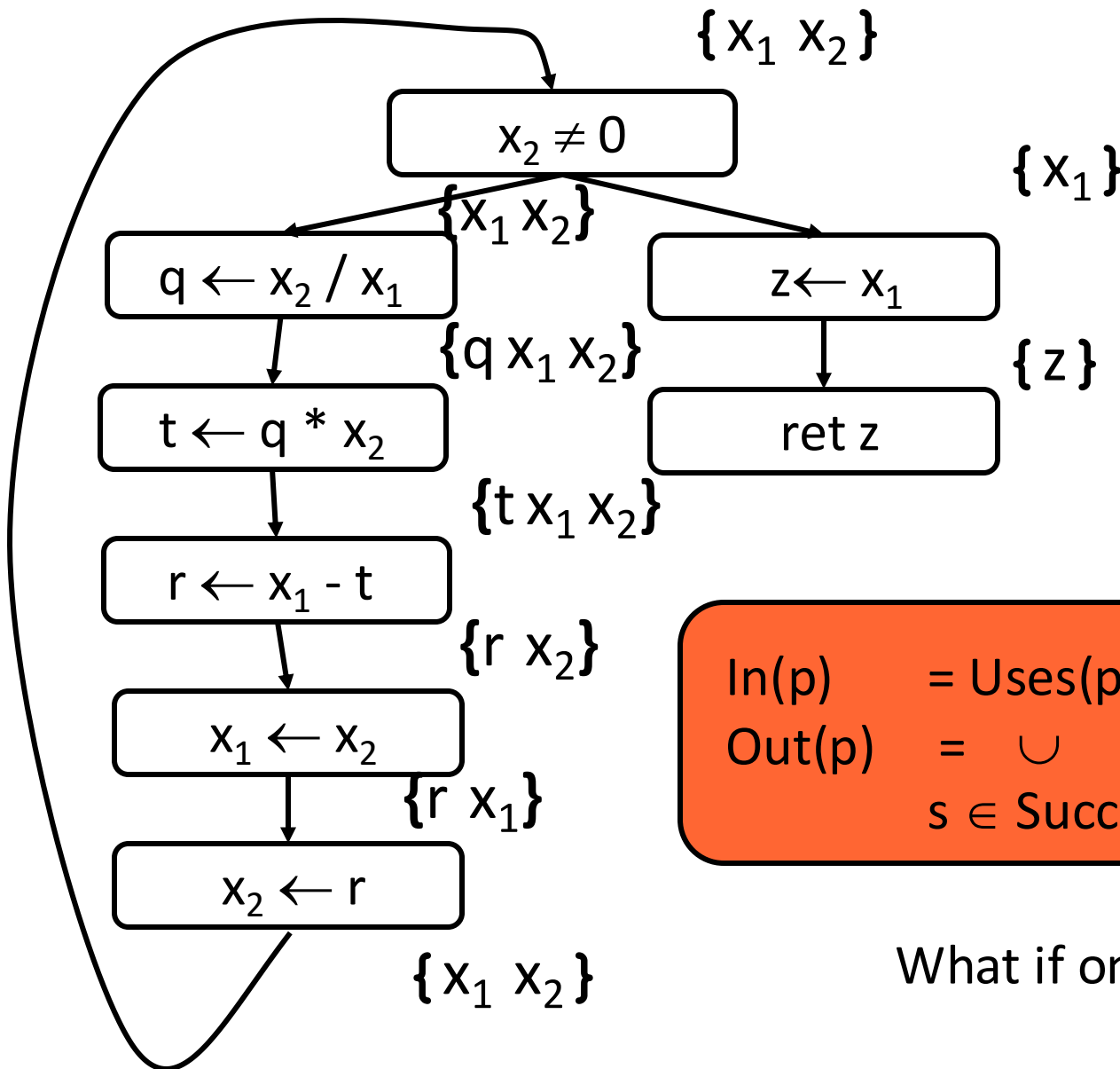


Another Alternative



$\text{In}(p) = \text{Uses}(p) \cup (\text{Out}(p) - \text{Defs}(p))$
 $\text{Out}(p) = \bigcup_{s \in \text{Succ}(p)} \text{In}(s)$

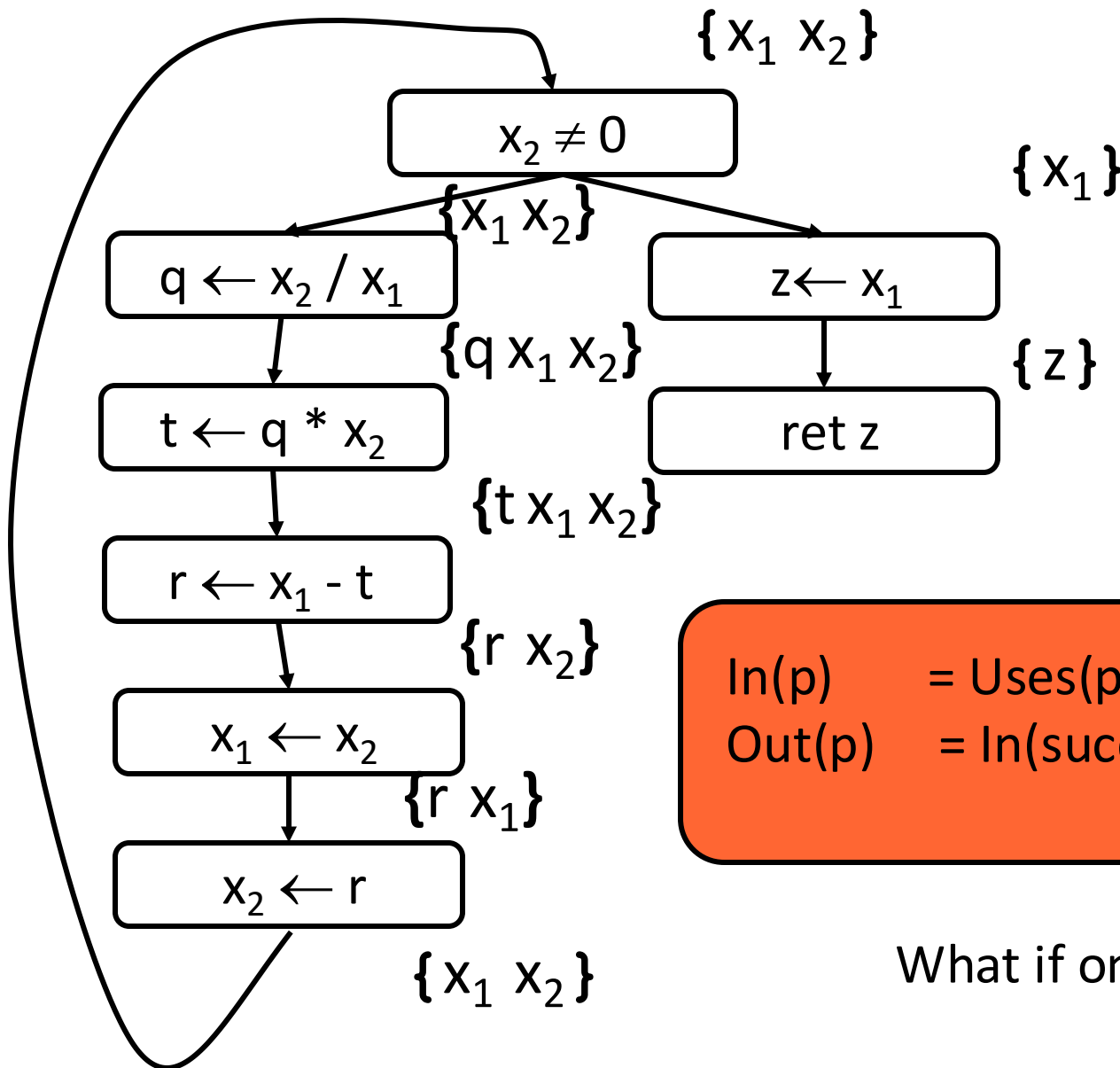
Improving Algorithm



$\text{In}(p) = \text{Uses}(p) \cup (\text{Out}(p) - \text{Defs}(p))$
 $\text{Out}(p) = \bigcup_{s \in \text{Succ}(p)} \text{In}(s)$

What if only 1 successor?

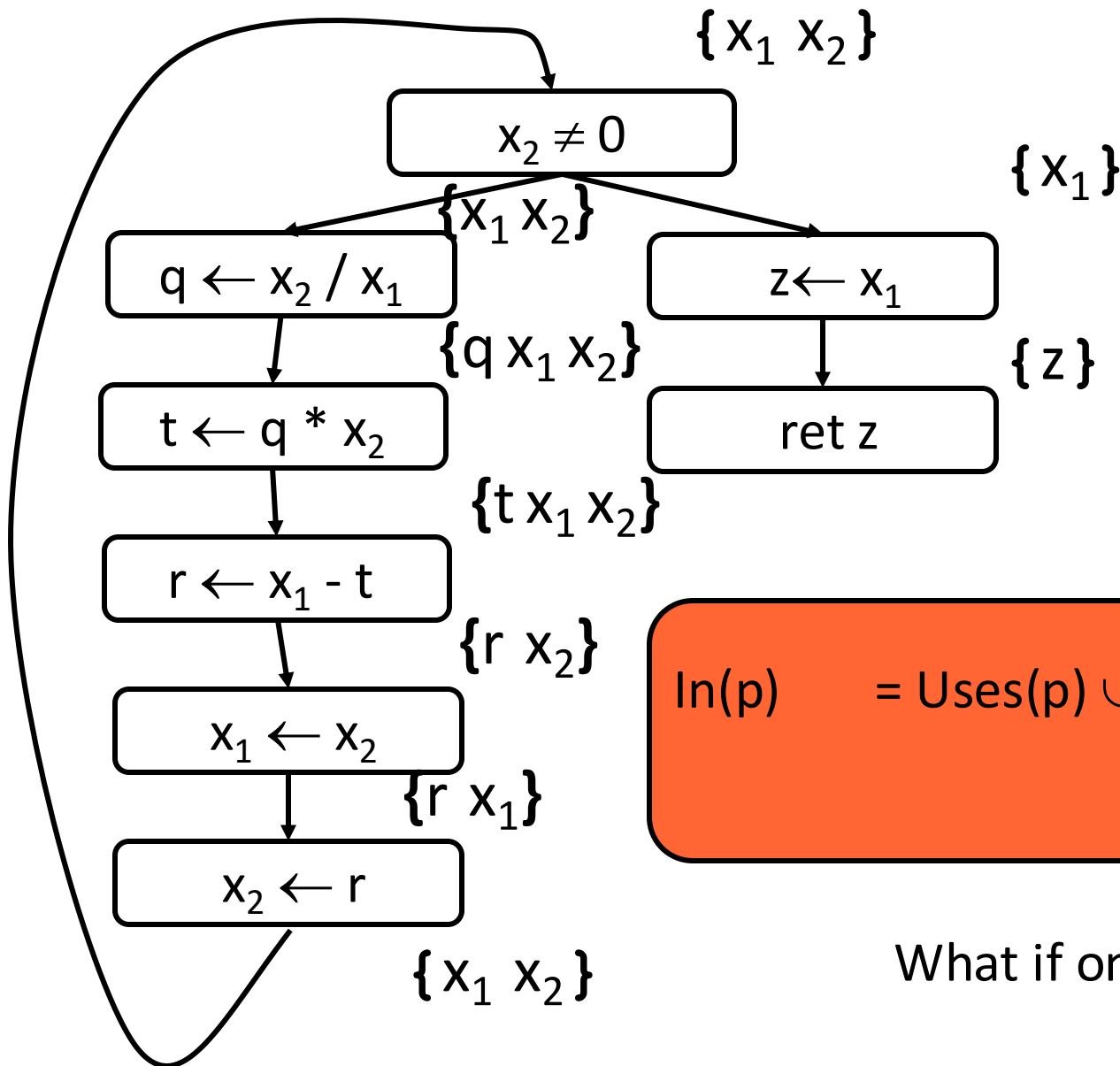
Improving Algorithm



$\text{In}(p) = \text{Uses}(p) \cup (\text{Out}(p) - \text{Defs}(p))$
 $\text{Out}(p) = \text{In}(\text{succ}(p))$

What if only 1 successor?

Improving Algorithm



$$\text{In}(p) = \text{Uses}(p) \cup (\text{In}(\text{succ}(p)) - \text{Defs}(p))$$

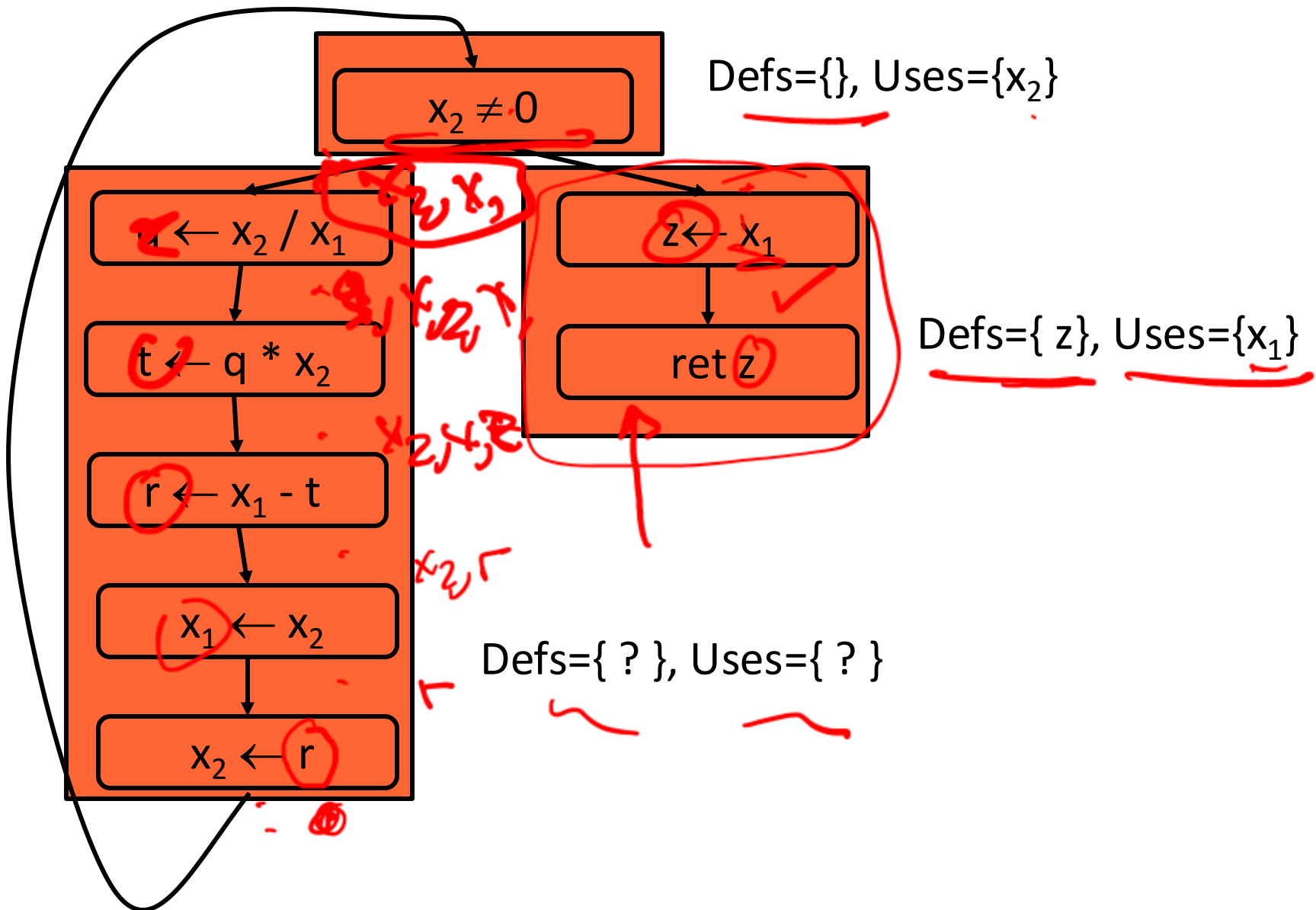
What if only 1 successor?

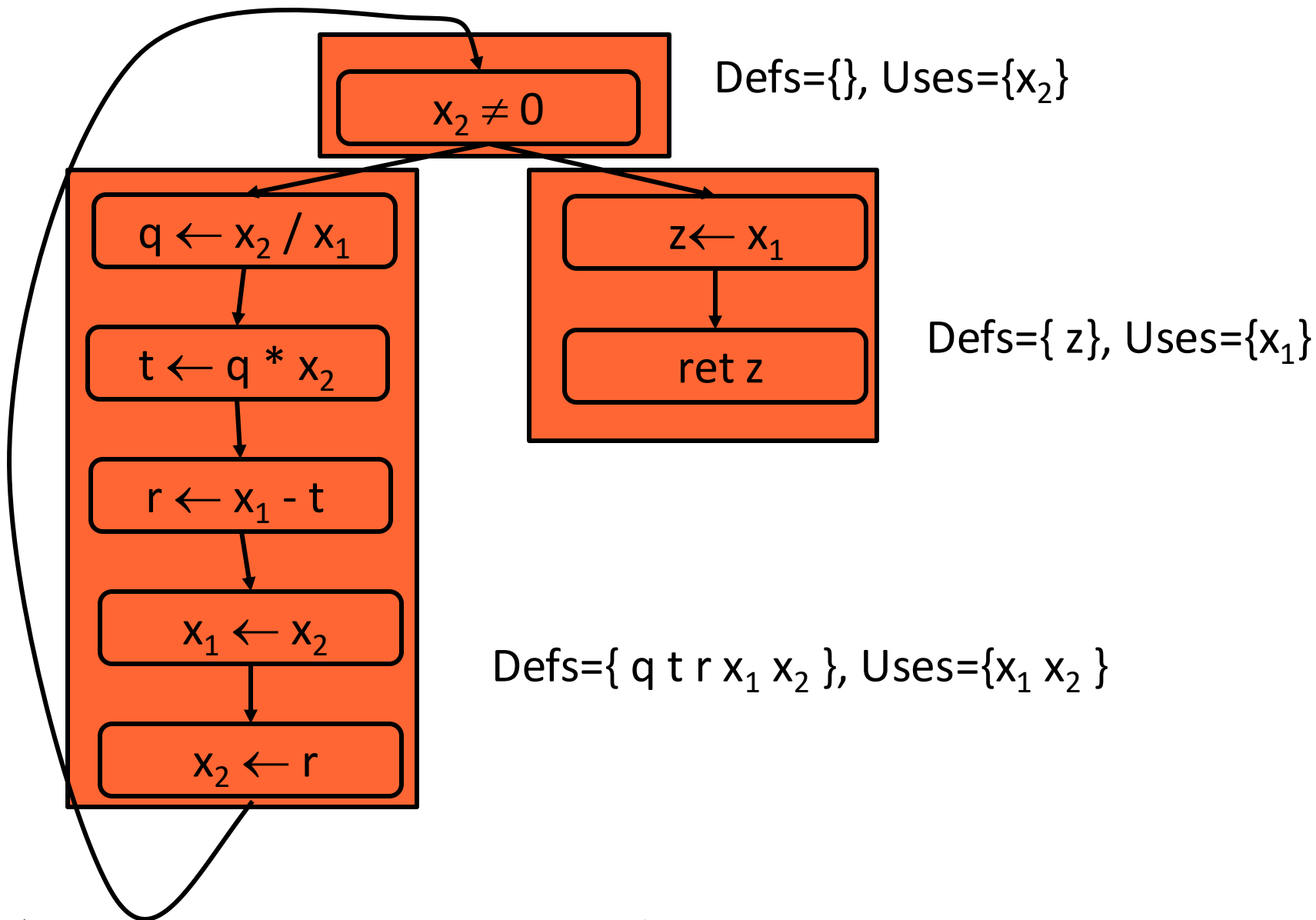
Using BasicBlocks

- A basic block has a single entry point and a single exit.
- Once you start a basic block you execute all of it
- Can create a single dataflow equation for the block by expanding:
$$\text{In}(p) = \text{Uses}(p) \cup (\text{In}(\text{succ}(p)) - \text{Defs}(p))$$
- And, then instead of in and out sets for each program point, you get ones for each basic block

Basic Blocks

- Each basic block starts with a “leader”
 - function entry
 - label
- Ends with **return** or **jmp**
- Only 1 entry, only 1 exit
- If last statement is conditional jump, two possible successors in control flow graph

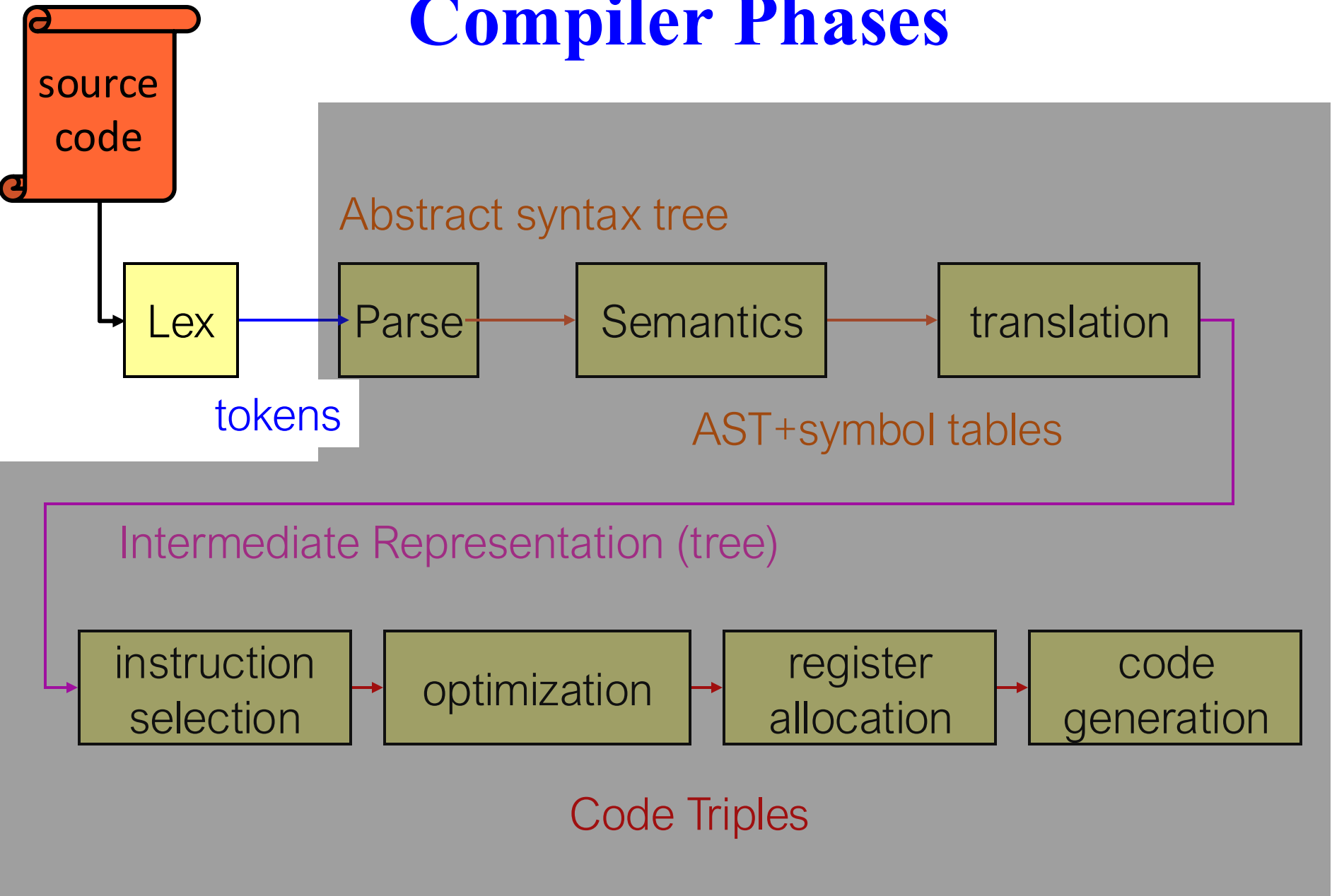




Liveness Analysis

- An example of a dataflow analysis
- There are many different dataflow analysis.
- Liveness is an example of a **backward, may** analysis
- We will see many others later on.

Compiler Phases



The Lexer

- Turn stream of characters into a stream of tokens

```
// create a user friendly descriptor for this arg.  
// if key is absent, then use it.  Otherwise use longkey  
  
char*  
ArgDesc::helpkey(WhichKey keytype, bool includebraks)  
{  
    static char buffer[128];  /* format buffer */  
    char* p = buffer;  
    ...  
}
```

```
CHAR STAR ID DOUBLE_COLON ID LPARIN ID ID COMMA BOOL ID  
RPARIN LBRACE STATIC CHAR ID LBRAK INTCONST RBRAK SEMI  
CHAR STAR ID EQ ID SEMI ...
```

The Lexer

- Turn stream of characters into a stream of tokens
 - Strips out “unnecessary characters”
 - comments
 - whitespace
 - Classify tokens by type
 - keywords
 - numbers
 - punctuation
 - identifiers
 - Track location
 - Associate with syntactic information

The Lexer

- Turn stream of characters into a stream of tokens

```
// create a user friendly descriptor for this arg.  
// if key is absent, then use it.  Otherwise use longkey  
  
char*  
ArgDesc::helpkey(WhichKey keytype, bool includebraks)  
{  
    static char buffer[128]; /* format buffer */  
    char* p = buffer;  
    ...  
}
```

```
CHAR STAR ID DOUBLE COLON ID LPARIN ID ID COMMA BOOL ID  
RPARIN LBRACE STATIC CHAR ID LBRAK INTCONST RBRAK SEMI  
CHAR STAR ID EQ ID SEMI ...
```

The Lexer

- Turn stream of characters into a stream of tokens

```
// create a user friendly descriptor for this arg.  
// if key is absent, then use it.  Otherwise use longkey
```

```
char*  
ArgDesc::helpkey(WhichKey keytype, bool includebraks)  
{  
    static char buffer[128]; /* format buffer */  
    char* p = buffer;
```

Position: 4,0

Position: 5,40
text: "includebraks"

```
CHAR STAR ID DOUBLE COLON ID LPARIN ID ID COMMA BOOL ID  
RPARIN LBRACE STATIC CHAR ID LBRAK INTCONST RBRAK SEMI  
CHAR STAR ID EQ ID SEMI ...
```

Position: 6,23
value: 123

The Lexer

- Turn stream of characters into a stream of tokens
 - More concise
 - Easier to parse

Position: 4,0

Position: 5,40
text: "includebraks"

CHAR STAR ID DOUBLE COLON ID LPARIN ID ID COMMA BOOL ID
RPARIN LBRACE STATIC CHAR ID LBRAK INTCONST RBRAK SEMI
CHAR STAR ID EQ ID SEMI ...

Position: 6,23
value: 123

Lexical Analyzers

- Input: stream of characters
- Output: stream of tokens (with information)
- How to build?
 - By hand is tedious
 - Use Lexical Analyzer Generator, e.g., flex
- Define tokens with regular expressions
- Flex turns REs into Deterministic Finite Automata (DFA) which recognizes and returns tokens.

2. Flex Program Format

- A flex program has three sections:

Definitions

% %

RE rules & actions

% %

User code

wc As a Flex Program

```
%{
    int charCount=0, wordCount=0, lineCount=0;
}%
word    [^ \t\n]+
%%
{word}  {wordCount++; charCount += yyleng; }
[\\n]    {charCount++; lineCount++;}
.        {charCount++;}
%%
int main(void) {
    yylex();
    printf("Chars %d, Words: %d, Lines: %d\\n",
        charCount, wordCount, lineCount);
    return 0;
}
```

Section 1: RE Definitions

- Format:

name	RE
------	----

- Examples:

<code>digit</code>	<code>[0-9]</code>
--------------------	--------------------

<code>letter</code>	<code>[A-Za-z]</code>
---------------------	-----------------------

<code>id</code>	<code>{letter} ({letter} {digit})*</code>
-----------------	---

<code>word</code>	<code>[^ \t\n]+</code>
-------------------	------------------------

Regular Expressions in Flex

x	match the char x
\.	match the char .
"string"	match contents of string of chars
.	match any char except \n
^	match beginning of a line
\$	match the end of a line
[xyz]	match one char x , y , or z
[^xyz]	match any char except x , y , and z
[a-z]	match one of a to z

Some number REs

`[0-9]` A single digit.

`[0-9]+` An integer.

`[0-9]+ (\.[0-9]+)?` An integer or fp number.

`[+-]? [0-9]+ (\.[0-9]+)? ([eE][+-]?[0-9]+)?`
Integer, fp, or scientific notation.

Section 2: RE/Action Rule

- A rule has the form:

```
name      { action }  
re       { action }
```

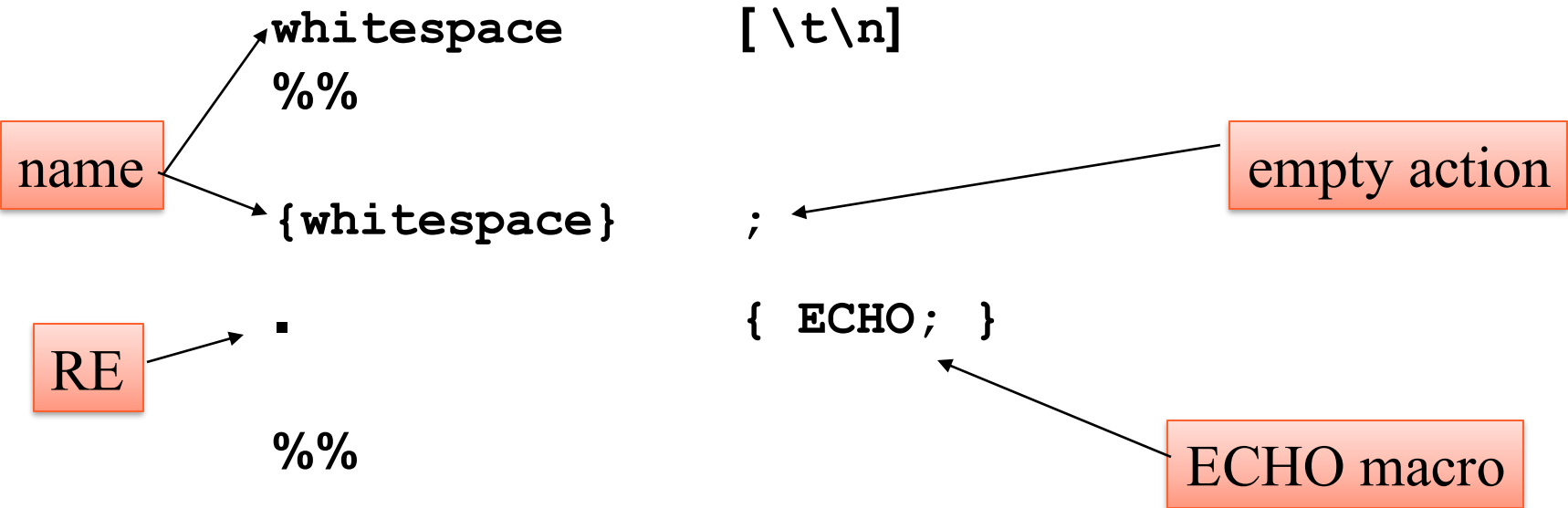
- the name must be defined in section 1
 - the action is any C code
-
- If the named RE matches^{*} an input character sequence, then the C code is executed.

^{*} Some caveats here

Section 3: C Functions

- Added to end of the lexical analyzer

Removing Whitespace



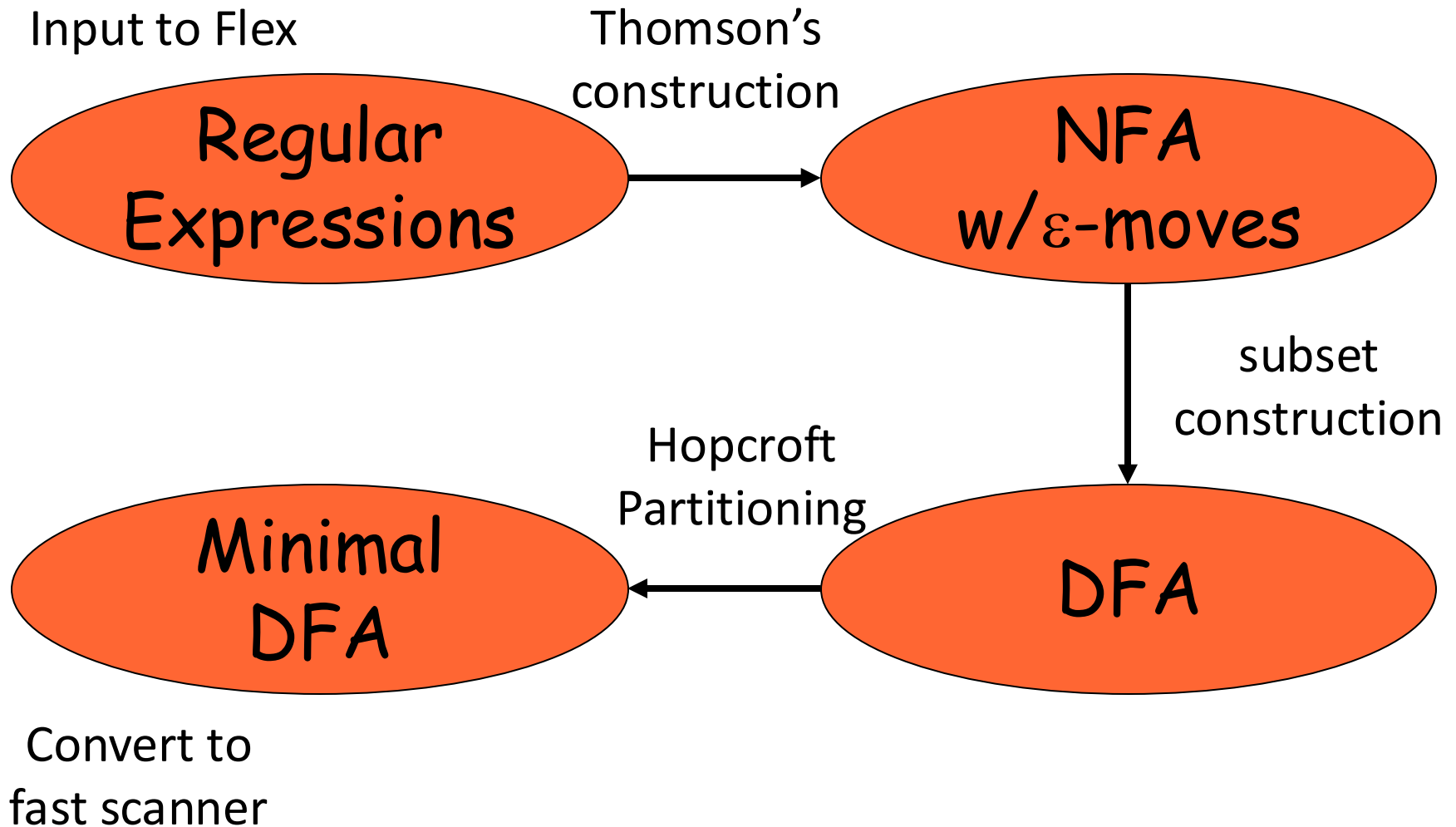
```
int main(void)
{
    yylex();
    return 0;
}
```

Today – part 1

- Lexing
- Flex & other scanner generators
- **Regular Expressions**
- Finite Automata
- RE $\rightarrow$ NFA
- NFA $\rightarrow$ DFA
- DFA $\rightarrow$ Minimized DFA
- Limits of Regular Languages

Under The Covers

- How to go from REs to a working scanner?



Regular Languages

- Finite Alphabet, Σ , of symbols.
- word (or string), a finite sequence of symbols from Σ .
- Language over Σ is a set of words from Σ .
- Regular Expressions describe Regular Languages.
 - easy to write down, but hard to use directly
- The languages accepted by Finite Automata are also Regular.

Regular Expressions defined

- Base Cases:
 - A single character a
 - The empty string ε
- Recursive Rules:

If R_1 and R_2 are regular expressions

 - Concatenation R_1R_2
 - Union $R_1 \mid R_2$
 - Closure R_1^*
 - Grouping (R_1)
- REs describe Regular Languages.

RE Examples

- even a's
- odd b's
- even a's or odd b's
- even a's followed by odd b's

RE Examples

- even a's

$$R^A = b^* (a b^* a b^*)^*$$

- odd b's

$$R^B = a^* b a^* (b a^* b a^*)^*$$

- even a's or odd b's

$$R^A \mid R^B$$

- even a's followed by odd b's

$$R^A R^B$$