IR Trees
Basic Liveness
Basic Lexing

15-411/15-611 Compiler Design

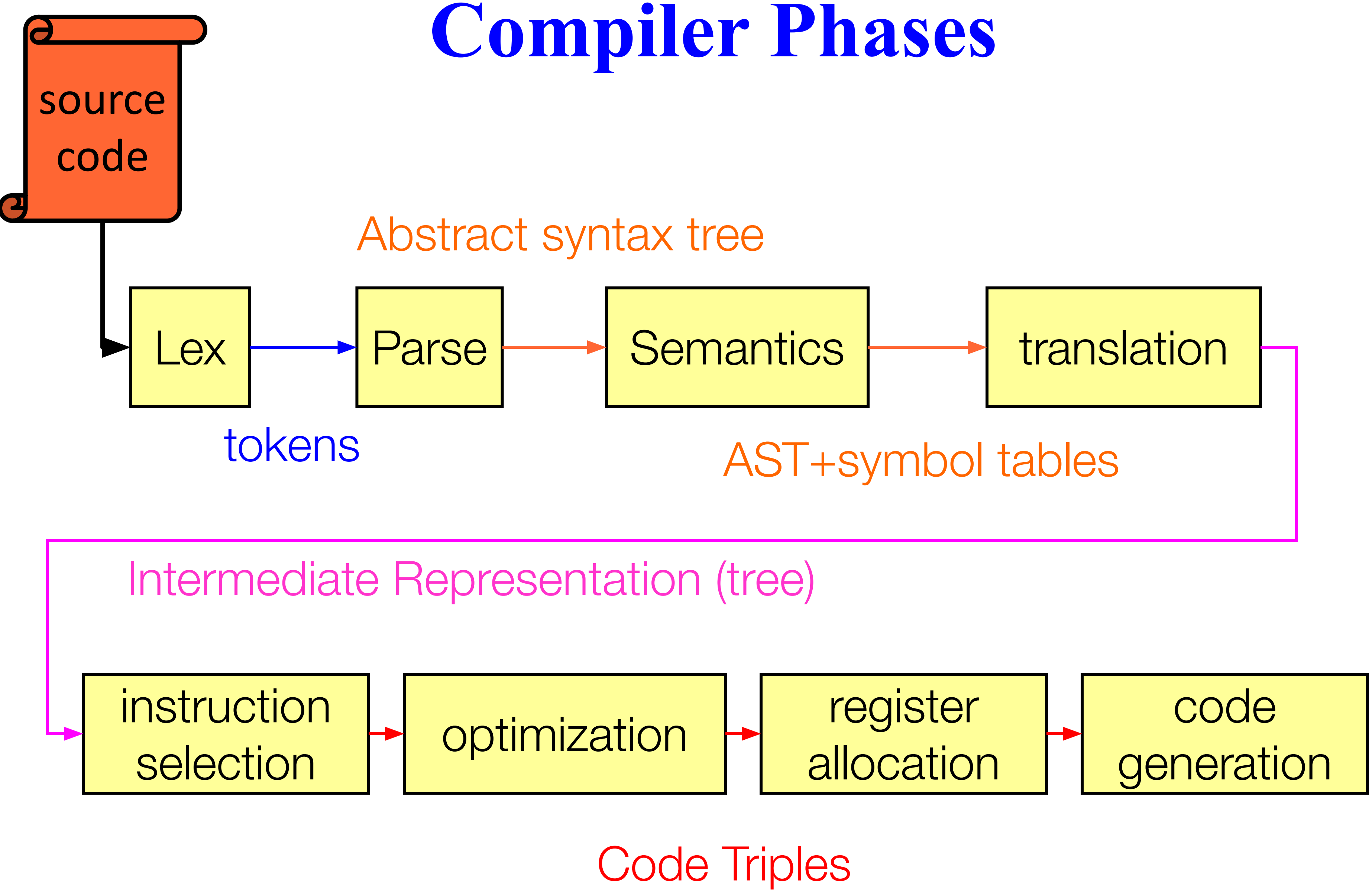
Seth Copen Goldstein

February 6, 2024

Today

- From AST ⇒ Tree IR
- Liveness across Control Flow
  - Basic Blocks
- 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

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

s ::= assign(x,e)
| if(e,s_1,s_2)
| decl(x,τ,s)
| while (e,s)
| return e
| nop
| 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

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

$c ::= x \leftarrow p$ Commands

$\mid x \leftarrow p_1 \oslash p_2$

$\mid x \leftarrow f(p_1, p_2, \ldots, p_n)$

$\mid$ if $(p_1 \gtrless p_2)$ then l_t else l_f

$\mid$ goto l

$\mid l$:

$\mid$ return(p)

$r ::= c_1 ; \ldots ; c_n$ Programs

Translating (Integer) Expressions

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

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

$e_1 \oslash e_2$?

$p \quad ::= \ n \mid x \mid p_1 \oplus p_2$

$c \quad ::= \ x \leftarrow p$

$\quad \mid \quad x \leftarrow p_1 \oslash p_2$

$\quad \mid \quad x \leftarrow f(p_1, p_2, \ldots, p_n)$

$\quad \mid$ if ($p_1 \gtrless p_2$) then l_t else l_f

$\quad \mid$ goto l

$\quad \mid$ l:

$\quad \mid$ return(p)

$r \quad ::= \ c_1 \ ; \ \ldots \ ; \ c_n$

Translating (Integer) Expressions

$$e \quad ::= n \mid x \mid e_1 \oplus e_2 \mid e_1 \oslash e_2 \mid e_1 \gtrless e_2$$
$$\mid f(e_1,\ldots,e_n) \mid\ !e \mid e_1 \mathbin{\&\&} e_2 \mid e_1\ ?\ e_2 : e_3$$

$$
\begin{array}{lll}
p & ::= & n \mid x \mid p_1 \oplus p_2 \\
c & ::= & x \leftarrow p \\
 & \mid & x \leftarrow p_1 \oslash p_2 \\
 & \mid & x \leftarrow f(p_1, p_2, \ldots, p_n) \\
 & \mid & \text{if } (p_1 \gtrless p_2) \text{ then } l_t \text{ else } l_f \\
 & \mid & \text{goto } l \\
 & \mid & l: \\
 & \mid & \text{return}(p) \\
r & ::= & c_1 ; \ldots ; c_n
\end{array}
$$

- exp becomes
  - seq of commands
  - pure-expression
- tr(e) = < r , p >

result of e is p

Translating (Integer) Expressions

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

$$
\begin{array}{lll}
p & ::= & n \mid x \mid p_1 \oplus p_2 \\
c & ::= & x \leftarrow p \\
 & \mid & x \leftarrow p_1 \oslash p_2 \\
 & \mid & x \leftarrow f(p_1, p_2, \ldots, p_n) \\
 & \mid & \text{if } (p_1 \gtrless p_2) \text{ then } l_t \text{ else } l_f \\
 & \mid & \text{goto } l \\
 & \mid & l: \\
 & \mid & \text{return}(p) \\
r & ::= & c_1 \,;\, \ldots \,;\, c_n
\end{array}
$$

$$
\begin{array}{lll}
e & ::= & n \mid x \mid e_1 \oplus e_2 \mid e_1 \oslash e_2 \mid e_1 \gtrless e_2 \\
 & & \mid f(e_1,\ldots,e_n) \mid\ !e \mid e_1 \,\&\&\, e_2 \mid e_1\ ?\ e_2 : e_3
\end{array}
$$

Translating (Integer) Expressions

- $\mathrm{tr}(n) = < \cdot\ , n >$
- $\mathrm{tr}(x) = < \cdot\ , x >$
- $\mathrm{tr}(e_1 \oplus e_2) :=$

  $\mathrm{tr}(e_1) = < r_1, p_1 >$

  $\mathrm{tr}(e_2) = < r_2, p_2 >$

  $< r_1; r_2\ , p_1 \oplus p_2 >$

$$
\begin{array}{lll}
p & ::= & n \mid x \mid p_1 \oplus p_2 \\
c & ::= & x \leftarrow p \\
 & \mid & x \leftarrow p_1 \oslash p_2 \\
 & \mid & x \leftarrow f(p_1, p_2, \ldots, p_n) \\
 & \mid & \text{if } (p_1 \gtrless p_2) \text{ then } l_t \text{ else } l_f \\
 & \mid & \text{goto } l \\
 & \mid & l: \\
 & \mid & \text{return}(p) \\
r & ::= & c_1 ; \ldots ; c_n
\end{array}
$$

$$
\begin{array}{lll}
e & ::= & n \mid x \mid e_1 \oplus e_2 \mid e_1 \oslash e_2 \mid e_1 \gtrless e_2 \\
 & & \mid f(e_1,\ldots,e_n) \mid\ !e \mid e_1 \,\&\&\, e_2 \mid e_1\ ?\ e_2 : e_3
\end{array}
$$

Translating (Integer) Expressions

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

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

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

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

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

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

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

$$
\begin{array}{lll}
p & ::= & n \mid x \mid p_1 \oplus p_2 \\
c & ::= & x \leftarrow p \\
 & \mid & x \leftarrow p_1 \oslash p_2 \\
 & \mid & x \leftarrow f(p_1, p_2, \ldots, p_n) \\
 & \mid & \text{if } (p_1 \gtrless p_2) \text{ then } l_t \text{ else } l_f \\
 & \mid & \text{goto } l \\
 & \mid & l: \\
 & \mid & \text{return}(p) \\
r & ::= & c_1 ; \ldots ; c_n
\end{array}
$$

Translating (Integer) Expressions

- $\mathrm{tr}(n) = < \cdot\,, n >$
- $\mathrm{tr}(x) = < \cdot\,, x >$
- $\mathrm{tr}(e_1 \oplus e_2) :=$
  $\mathrm{tr}(e_1) = < r_1, p_1 >$
  $\mathrm{tr}(e_2) = < r_2, p_2 >$
  $< r_1; r_2\,, p_1 \oplus p_2 >$
- $\mathrm{tr}(e_1 \oslash e_2) :=$
  $\mathrm{tr}(e_1) = < r_1, p_1 >$
  $\mathrm{tr}(e_2) = < r_2, p_2 >$
  $< r_1; r_2\,; t \leftarrow p_1 \oslash p_2\,, t>$ (t fresh)
- $\mathrm{tr}(f(e_1,\ldots,e_n)) =$

$$
\begin{array}{lll}
p & ::= & n \mid x \mid p_1 \oplus p_2 \\
c & ::= & x \leftarrow p \\
 & \mid & x \leftarrow p_1 \oslash p_2 \\
 & \mid & x \leftarrow f(p_1, p_2, \ldots, p_n) \\
 & \mid & \text{if } (p_1 \gtrless p_2) \text{ then } l_t \text{ else } l_f \\
 & \mid & \text{goto } l \\
 & \mid & l\text{:} \\
 & \mid & \text{return}(p) \\
r & ::= & c_1 \,; \ldots ; c_n
\end{array}
$$

Why call it Tree IR?

- $\mathrm{tr}(n) = < \cdot\ , n >$
- $\mathrm{tr}(x) = < \cdot\ , x >$
- $\mathrm{tr}(e_1 \oplus e_2) :=$

  $\mathrm{tr}(e_1) = < r_1, p_1 >$

  $\mathrm{tr}(e_2) = < r_2, p_2 >$

  $< r_1; r_2\ , p_1 \oplus p_2 >$
- $\mathrm{tr}(e_1 \oslash e_2) :=$

  $\mathrm{tr}(e_1) = < r_1, p_1 >$

  $\mathrm{tr}(e_2) = < r_2, p_2 >$

  $< r_1; r_2\ ; t \leftarrow p_1 \oslash p_2\ , t>$ (t fresh)
- $\mathrm{tr}(f(e_1,\ldots,e_n)) =$

  $\mathrm{tr}(e_1) = < r_1, p_1 >$

  $\mathrm{tr}(e_n) = < r_n, p_2 >$

  $< r_1; r_2\ ; t \leftarrow f(p_1,\ldots,p_n),$ (t fresh)

What next?

$$e \quad ::= n \mid x \mid e_1 \oplus e_2 \mid e_1 \oslash e_2 \mid e_1 \gtrless e_2$$
$$\mid f(e_1,\ldots,e_n) \mid\ !e \mid e_1 \mathbin{\&\&} e_2 \mid e_1\ ?\ e_2 : e_3$$

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

Statements

tr(assign(x,e))) :=

tr(if(e,s_1,s_2))

tr(decl(x,τ,s))

tr(while (e,s))

tr(return e) :=

tr(nop) :=

tr(seq(s_1,s_2)) :=

$$
\begin{array}{lll}
p & ::= & n \mid x \mid p_1 \oplus p_2 \\
c & ::= & x \longleftarrow p \\
 & \mid & x \longleftarrow p_1 \oslash p_2 \\
 & \mid & x \longleftarrow f(p_1, p_2, \ldots, p_n) \\
 & \mid & \text{if } (p_1 \gtrless p_2) \text{ then } l_t \text{ else } l_f \\
 & \mid & \text{goto } l \\
 & \mid & l: \\
 & \mid & \text{return}(p) \\
r & ::= & c_1 \,;\, \ldots \,;\, c_n
\end{array}
$$

Statements

$\text{tr}(\text{assign}(x,e))) := \text{tr}(e_1) = < r_1, p_1 >$

$< r_1; x \leftarrow p_1 >$

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

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

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

$\text{tr}(\text{return } e) := \text{tr}(e_1) = < r_1, p_1 >$

$< r_1; \text{return}(p_1) >$

$\text{tr}(\text{nop}) :=$ ·

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

$$
\begin{array}{lll}
p & ::= & n \mid x \mid p_1 \oplus p_2 \\
c & ::= & x \leftarrow p \\
 & \mid & x \leftarrow p_1 \oslash p_2 \\
 & \mid & x \leftarrow f(p_1, p_2, \ldots, p_n \\
 & \mid & \text{if } (p_1 \gtrless p_2) \text{ then } l_t \\
 & \mid & \text{goto } l \\
 & \mid & l: \\
 & \mid & \text{return}(p) \\
r & ::= & c_1 ; \ldots ; c_n
\end{array}
$$

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

tr(if(e,s_1,s_2)) :=

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

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

$tr(if(e,s_1,s_2)) :=$

$tr(e) = < r, p >$

$< r ;$

if (e != 0) then l_t else l_f ;

$l_t : s_1$; goto l_3 ;

$l_f : s_2$; goto l_3 ;

$l_3 : >$

(l_t, l_f, l_3 fresh)

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

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

tr(if(e,s_1,s_2)) :=

tr(e) = $< r, p >$

$<$ r ;

if (e != 0) then l_t else l_f ;

l_t : s_1 ; goto l_3 ;

l_f : s_2 ; goto l_3 ;

l_3: $>$

(l_t, l_f, l_3 fresh)

tree-IR is in basic block form

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

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

$\mathrm{tr}(e) = < r, p >$

< r ;

if (e != 0) then l_t else l_f ;

$l_t : s_1$; goto l_3 ;

$l_f : s_2$; goto l_3 ;

l_3: >

(l_t, l_f, l_3 fresh)

And yet, something feels wrong here.

$\mathrm{cp}(b, l_t, l_f) = < r >$

$\mathrm{cp}(b, l_t, l_f) = < r >$

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

$\mathrm{cp}(e_1 \gtrless e_2, l_t, l_f)$

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

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

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

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

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

Translating Boolean Expressions

$\mathrm{cp}(e_1 \gtrless e_2, l_t, l_f) :=$ $\mathrm{tr}(e_1) = < r_1, p_1 >$

$\mathrm{tr}(e_2) = < r_2, p_2 >$

$< r_1; r_2;\ \mathrm{if}\ (p_1 \gtrless p_2)\ \mathrm{then}\ l_t\ \mathrm{else}\ l_f >$

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

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

$l_{fresh}: \mathrm{cp}(e_2, l_t, l_f)$

$\mathrm{cp}(0, l_t, l_f) = \mathrm{goto}\ l_f$

$\mathrm{cp}(1, l_t, l_f) = \mathrm{goto}\ l_t$

$\mathrm{cp}(e, l_t, l_f) := \mathrm{tr}(e) = < r, p >$

$< r\ ;\ \mathrm{if}\ (p\ != 0)\ \mathrm{then}\ l_t\ \mathrm{else}\ l_f >$

Translating Boolean Expressions

cp($e_1 \gtrless e_2, l_t, l_f$)	:=	tr(e_1) = < r_1, p_1 >
		tr(e_2) = < r_2, p_2 >
		< r_1, r_2, if ($p_1 \gtrless p_2$) then l_t else l_f >
cp(! e, l_t, l_f)	=	cp(e, l_f, l_t)
cp(e_1 && e_2, l_t, l_f)	=	cp(e_1, l_{fresh}, l_f) ;
		l_{fresh}: cp(e_2, l_t, l_f)
cp(0, l_t, l_f)	=	goto l_f
cp(1, l_t, l_f)	=	goto l_t
cp(e, l_t, l_f)	:=	tr(e) = < r, p >
		< r ; if (p != 0) then l_t else l_f >

tr(e) =

Translating Boolean Expressions

$cp(e_1 \gtrless e_2, l_t, l_f)$	:=	$tr(e_1) = < r_1, p_1 >$
		$tr(e_2) = < r_2, p_2 >$
		$< r_1, r_2, \text{if } (p_1 \gtrless p_2) \text{ then } l_t \text{ else } l_f >$
$cp(!\, e, l_t, l_f)$	=	$cp(e, l_f, l_t)$
$cp(e_1 \,\&\&\, e_2, l_t, l_f)$	=	$cp(e_1, l_{fresh}, l_f)\ ;$
		$l_{fresh}\text{: } cp(e_2, l_t, l_f)$
$cp(0, l_t, l_f)$	=	$\text{goto } l_f$
$cp(1, l_t, l_f)$	=	$\text{goto } l_t$
$cp(e, l_t, l_f)$	:=	$tr(e) = < r, p >$
		$< r\ ;\ \text{if } (p\ != 0) \text{ then } l_t \text{ else } l_f >$

$tr(e) = < cp(e, l_t, l_f)\ ;$

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

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

$l_{done} >$

take 2: tr(if(e,s_1,s_2))

tr(if(e,s_1,s_2)) :=

tr(e) = < r, p >

< r ;

if (e != 0) then l_t else l_f ;

l_t : s_1 ; goto l_3 ;

l_f : s_2 ; goto l_3 ;

l_3: >

(l_t, l_f, l_3 fresh)

take 2: tr(if(e,s_1,s_2))

tr(if(e,s_1,s_2)) :=

$<$ cp(e, l_t, l_f) ;

l_t : s_1 ; goto l_3 ;

l_f : s_2 ; goto l_3 ;

l_3: $>$

(l_t, l_f, l_3 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 ← 1
        { v }
w ← v + 3
        { w, v}
x ← w + v
        { v }
u ← v
        { u, x}
s ← u + x
        { w }
  ← w
        { s }
  ← s
        { u }
  ← u
```

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

What about before p?

Computing liveness

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

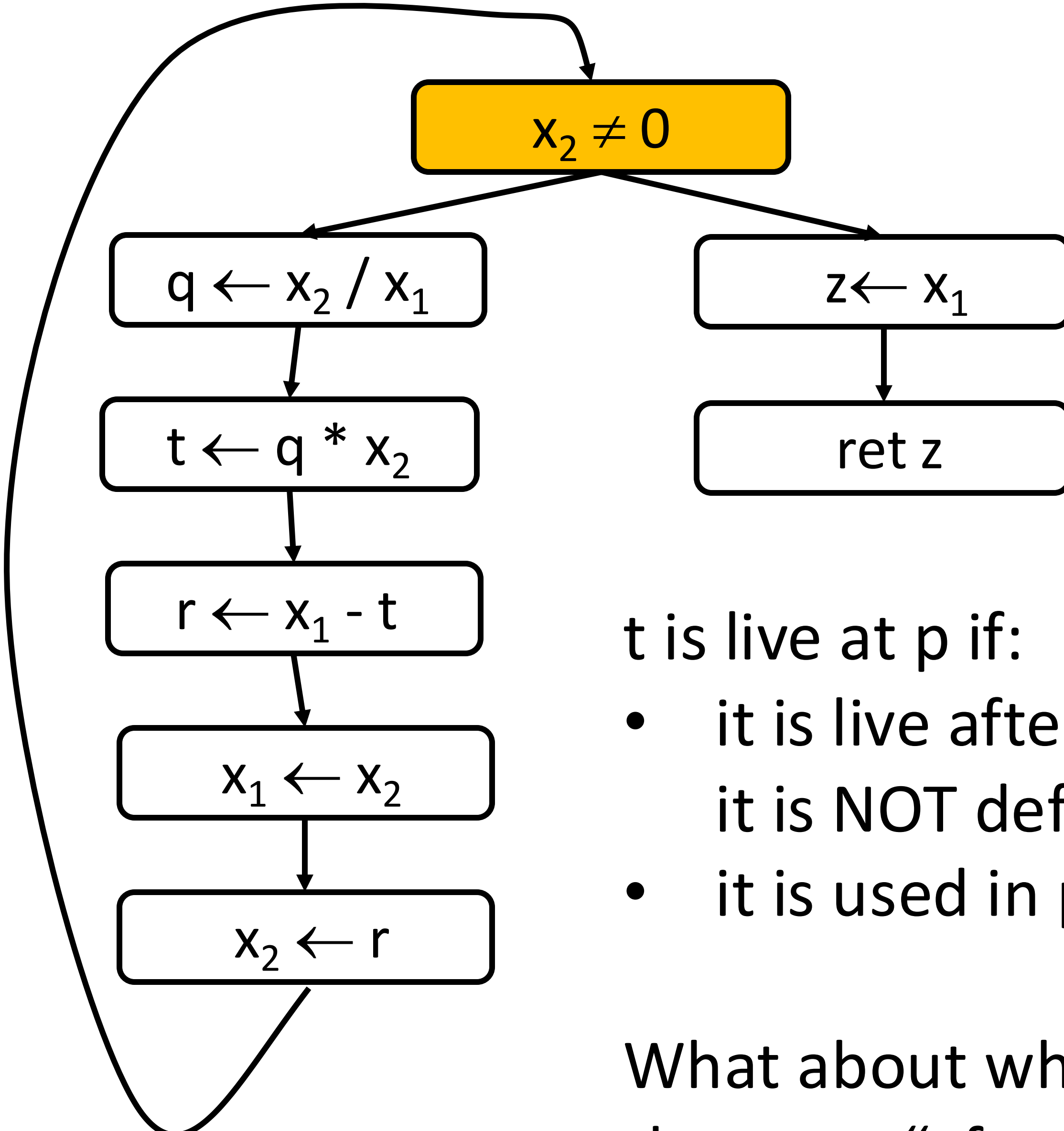
If t is used at point p, then it is live **immediately** before p

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?

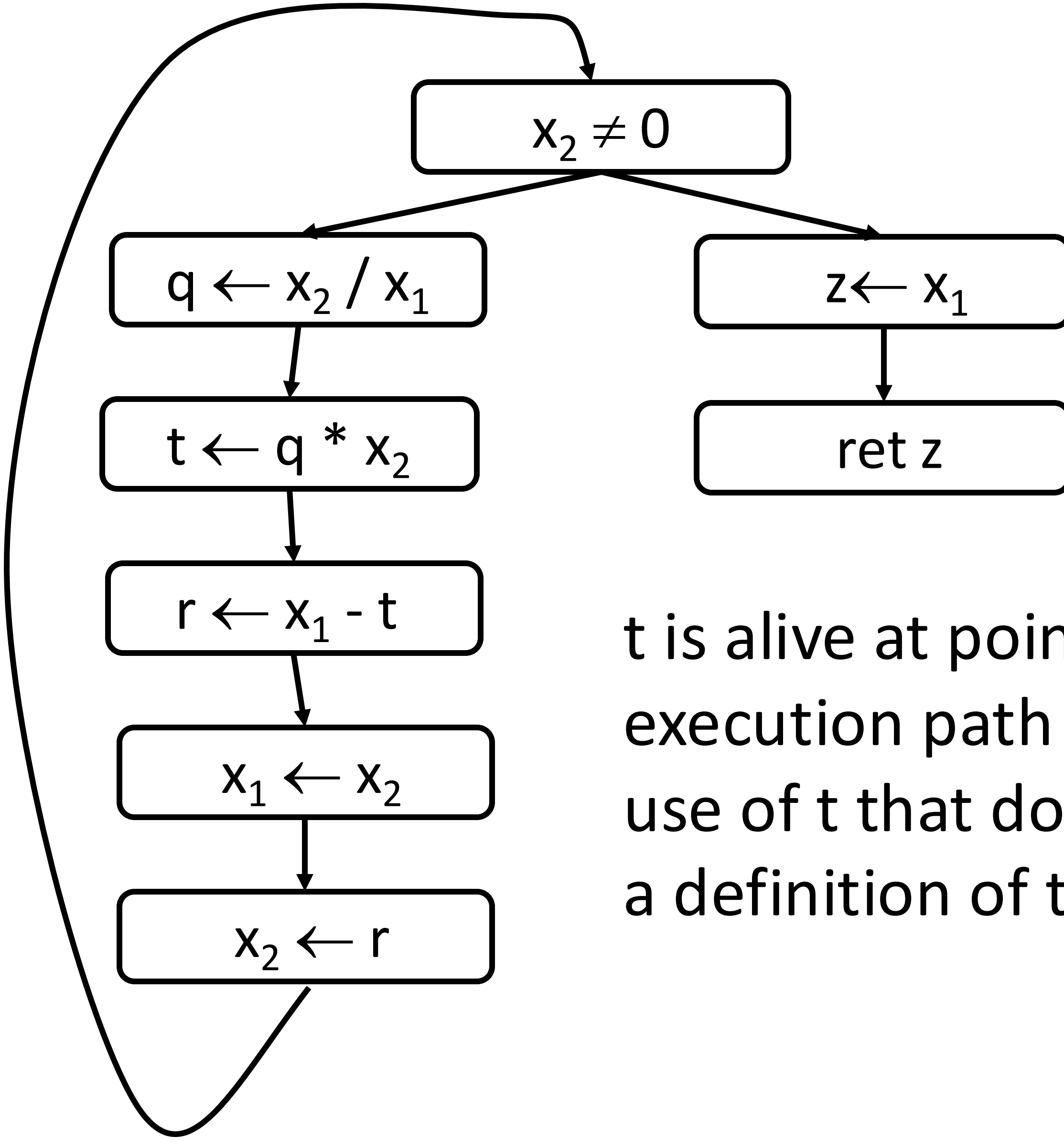
- We need to extend ideas to programs with control flow.
- Use Control Flow Graph (CFG) to represent the program
  - Nodes: program points, entry, exit
  - Edges: $(u,v) \in G$ if control can potentially go from u to v



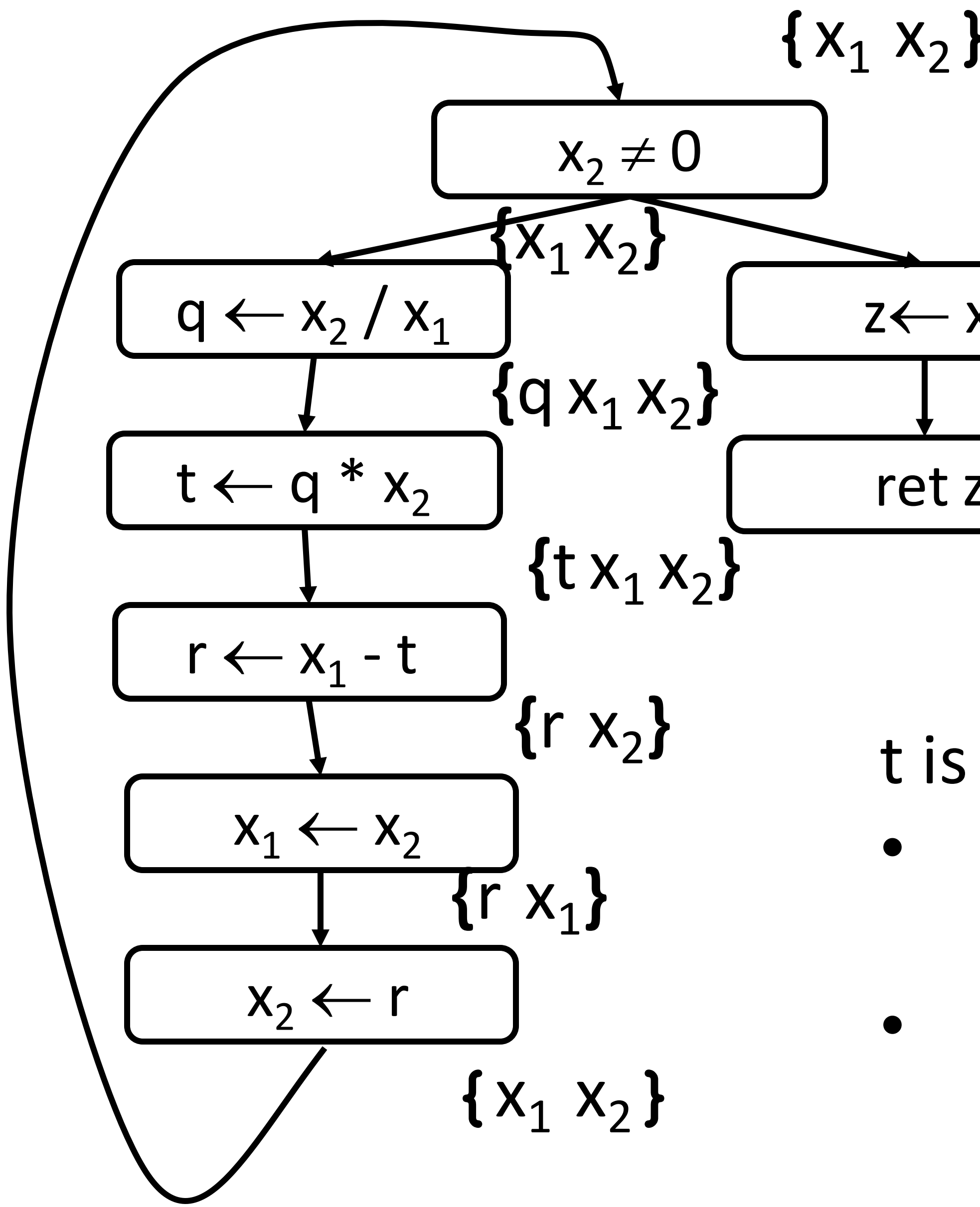
t is live at p if:

- 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)

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

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

Algorithm

forall nodes, $n \in$ CFG

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

Until no changes in In or Out sets:

forall nodes, $p \in$ CFG

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

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

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

Alternative: Worklist Approach

forall nodes, n $\in$ CFG

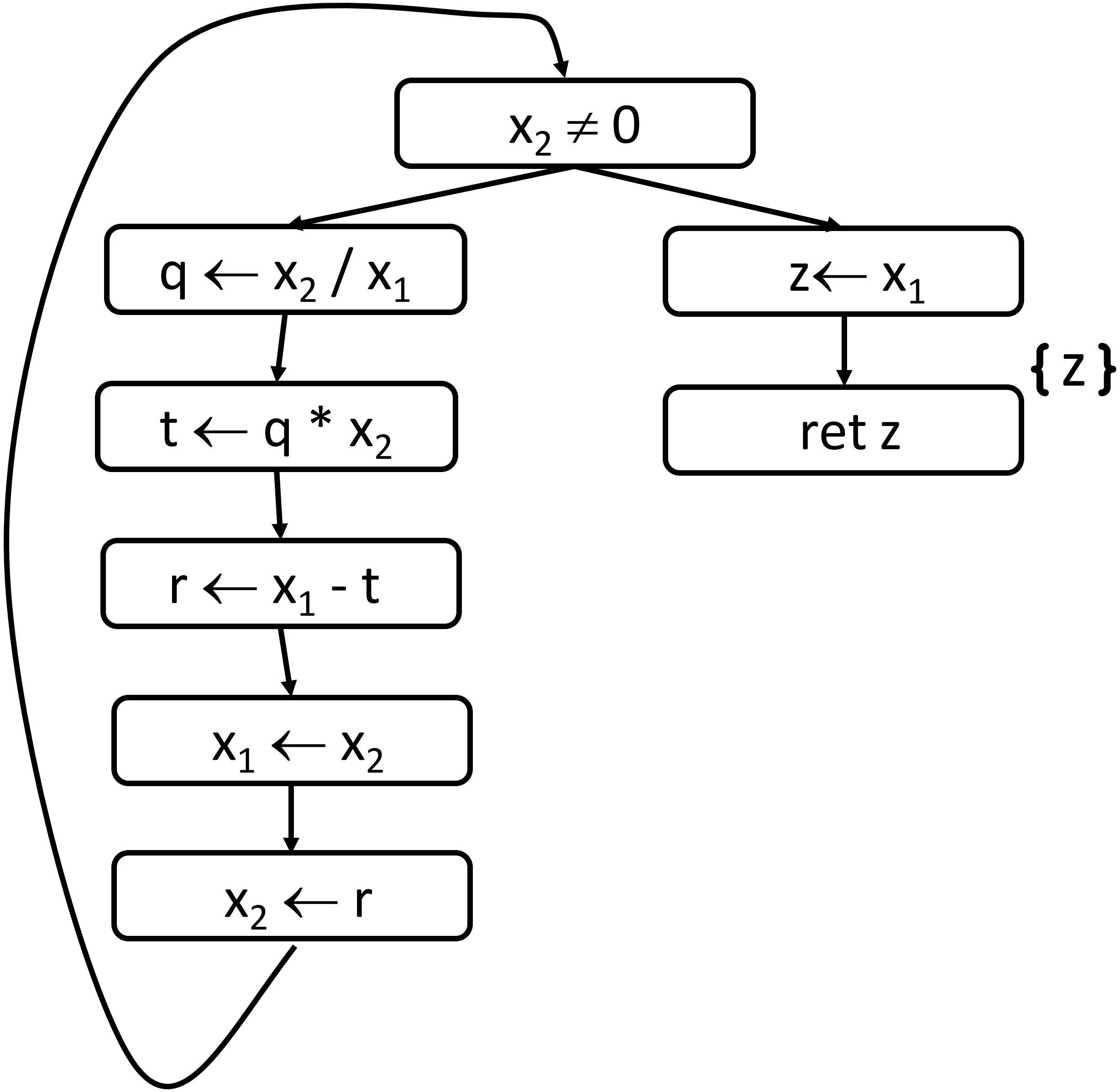
  In(n) = {}

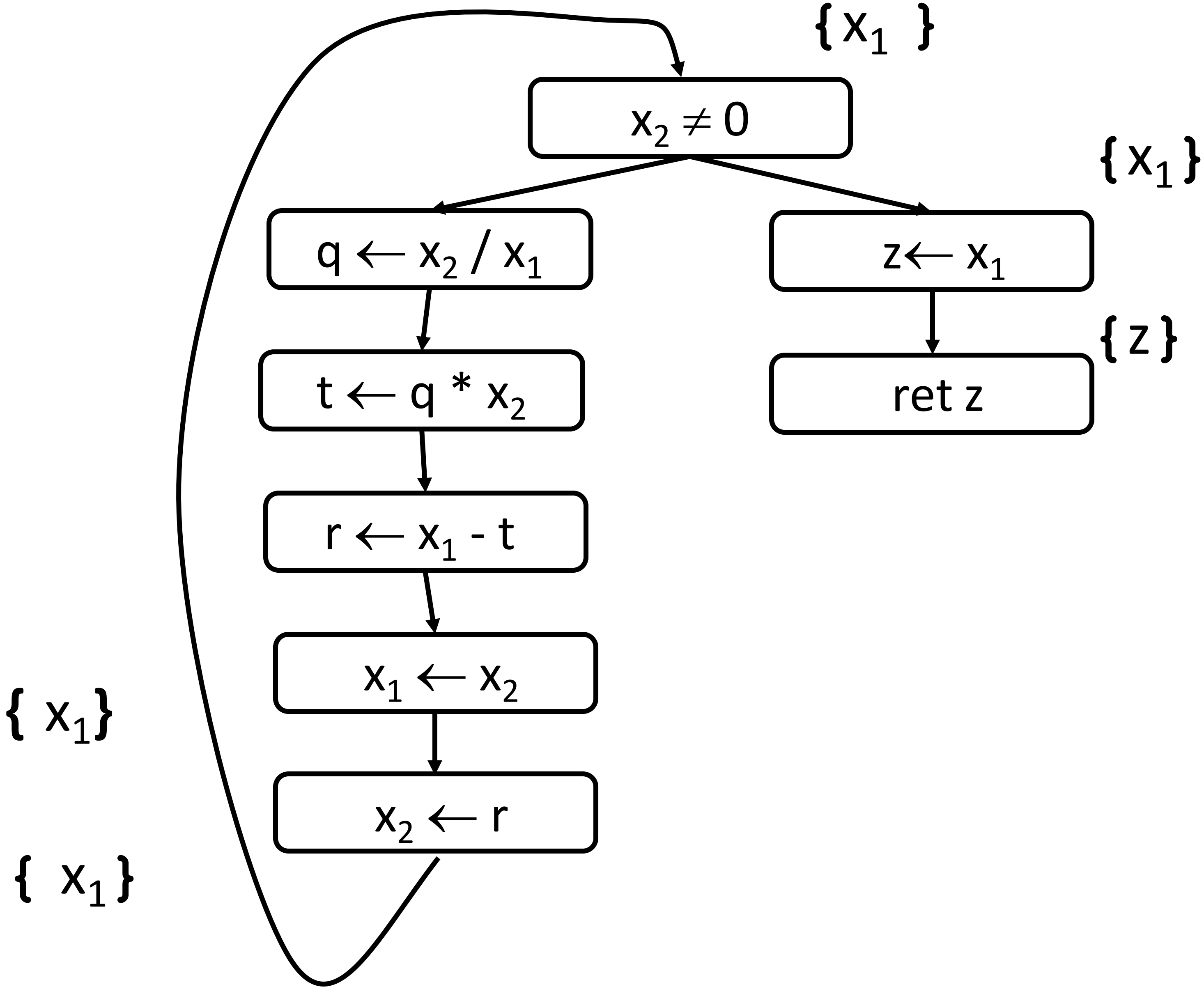
forall nodes, p $\in$ CFG:

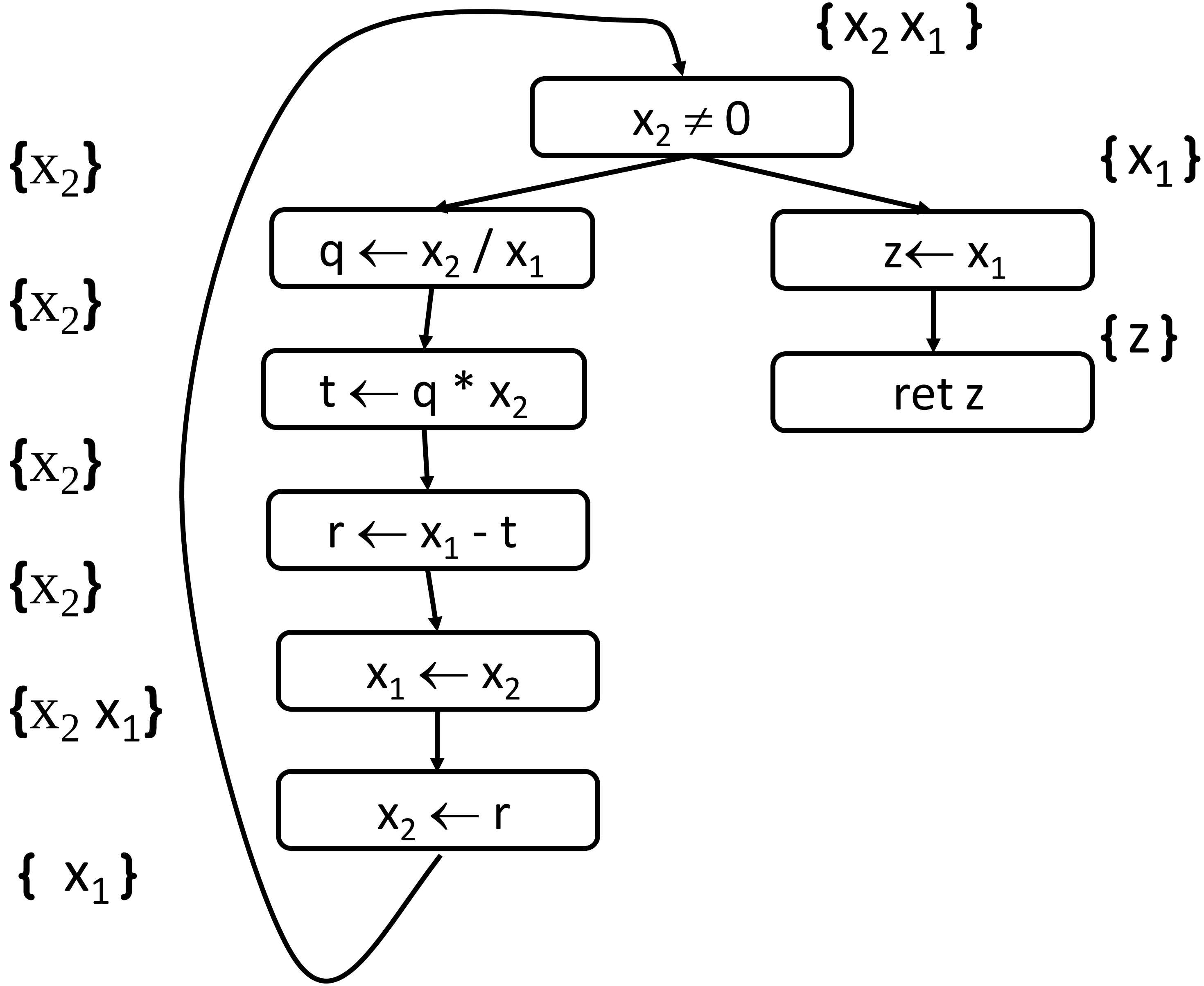
  if v $\in$ Uses(p) – Defs(p) not in In(p)

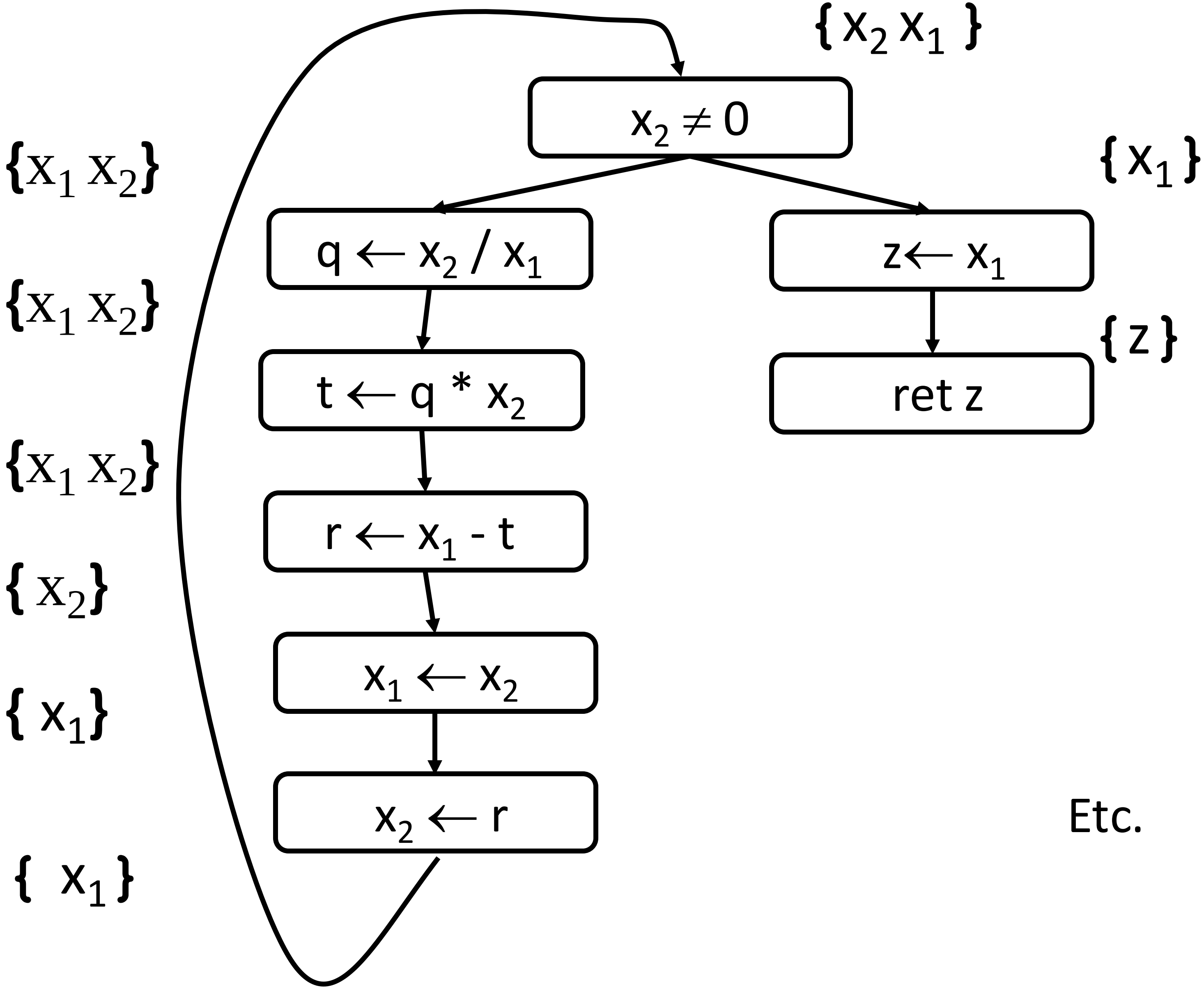
    In(p) $\cup$ { v }

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

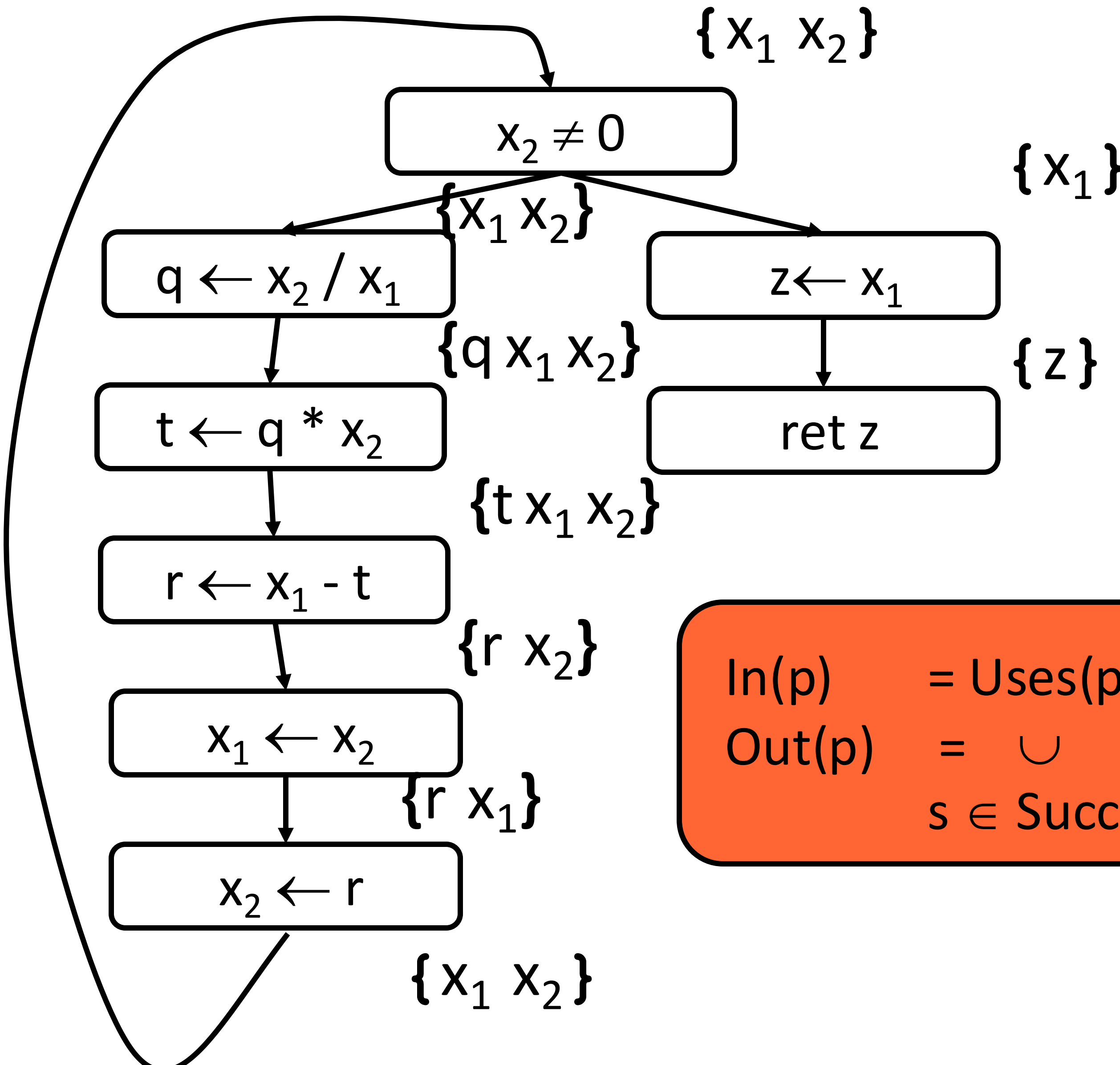

$x_2 \neq 0$
$q \leftarrow x_2 / x_1$
$z \leftarrow x_1$
$t \leftarrow q * x_2$
{ z }
ret z
$r \leftarrow x_1 - t$
$x_1 \leftarrow x_2$
$x_2 \leftarrow r$


{ x1 }
x2 ≠ 0
{ x1 }
q ← x2 / x1
z← x1
{ z }
t ← q * x2
ret z
r ← x1 - t
x1 ← x2
{ x1}
x2 ← r
{ x1}


{ x2 x1 }
x2 ≠ 0
{x2}
{ x1 }
q ← x2 / x1
z← x1
{x2}
{ z }
t ← q * x2
ret z
{x2}
r ← x1 - t
{x2}
x1 ← x2
{x2 x1}
x2 ← r
{ x1 }


{ x2 x1 }
x2 ≠ 0
{ x1 }
{x1 x2}
q ← x2 / x1
z ← x1
{x1 x2}
{ z }
t ← q * x2
ret z
{x1 x2}
r ← x1 - t
{ x2}
x1 ← x2
{ x1}
x2 ← r
{ x1 }
Etc.

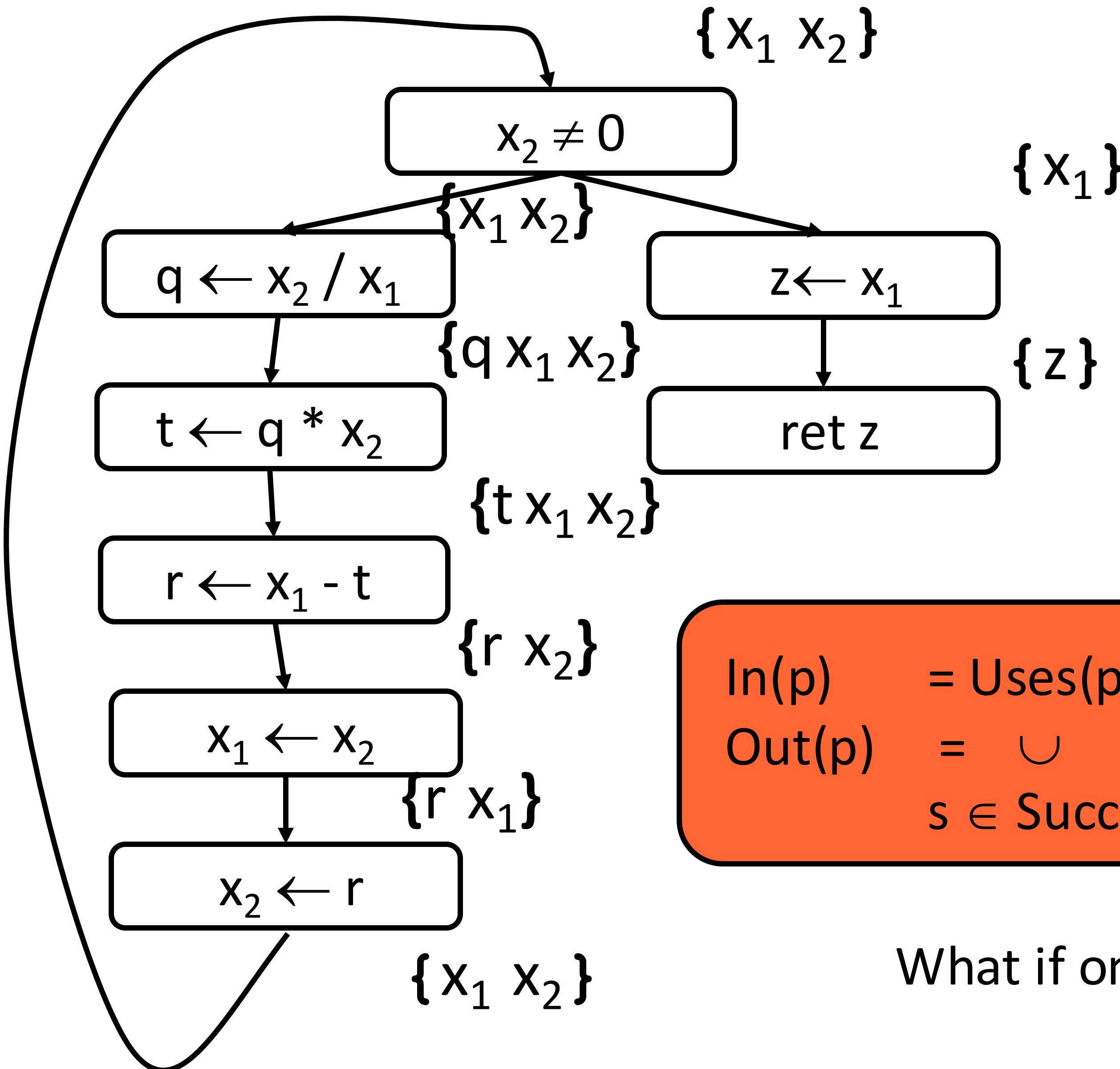
Another Alternative



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

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

Improving Algorithm

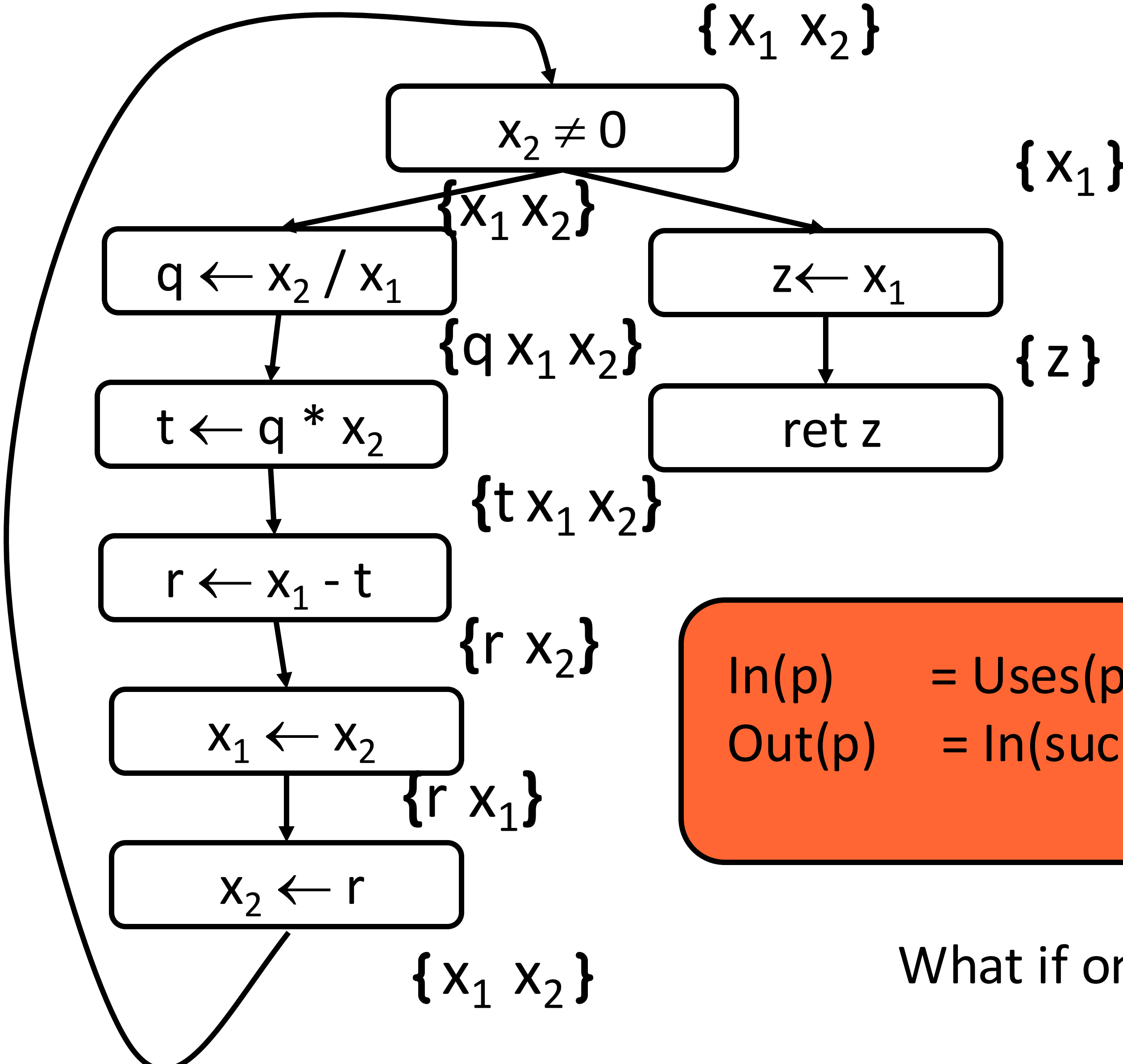


$$\text{In}(p) = \text{Uses}(p) \cup (\text{Out}(p) - \text{Defs}(p))$$

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

What if only 1 successor?

Improving Algorithm

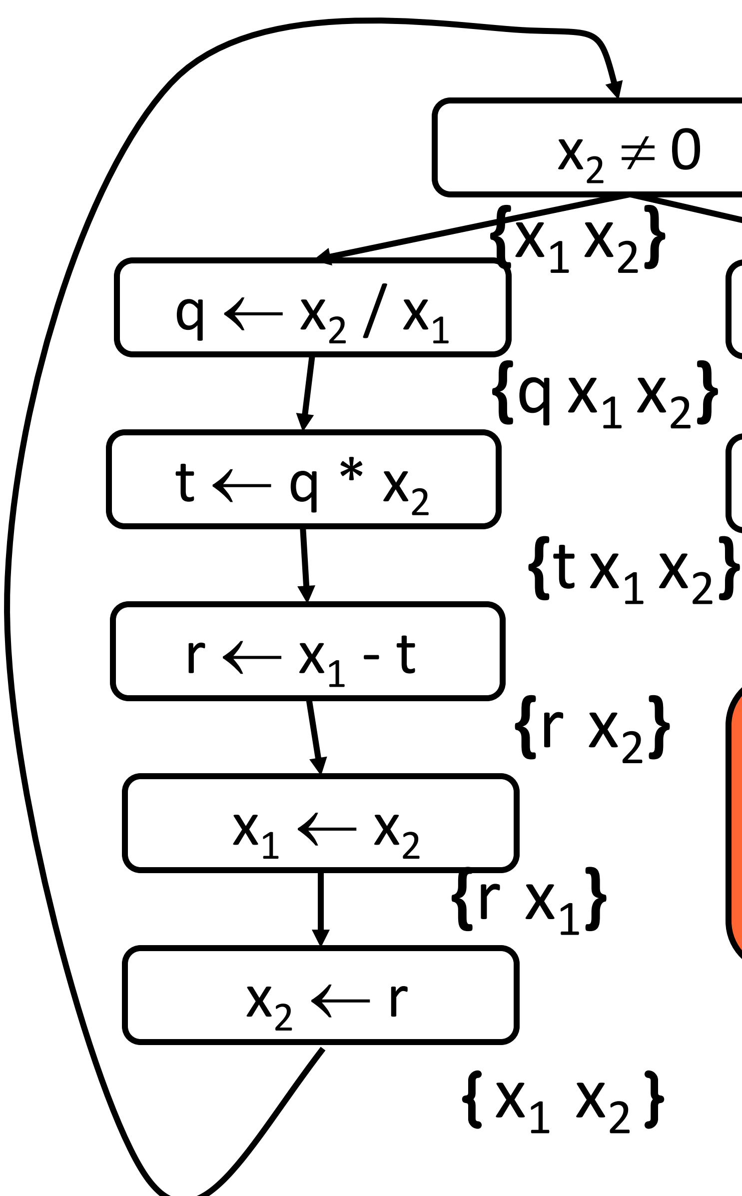


$\text{In}(p) = \text{Uses}(p) \cup (\text{Out}(p) - \text{Defs}(p))$

$\text{Out}(p) = \text{In}(\text{succ}(p))$

What if only 1 successor?

Improving Algorithm



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

What if only 1 successor?

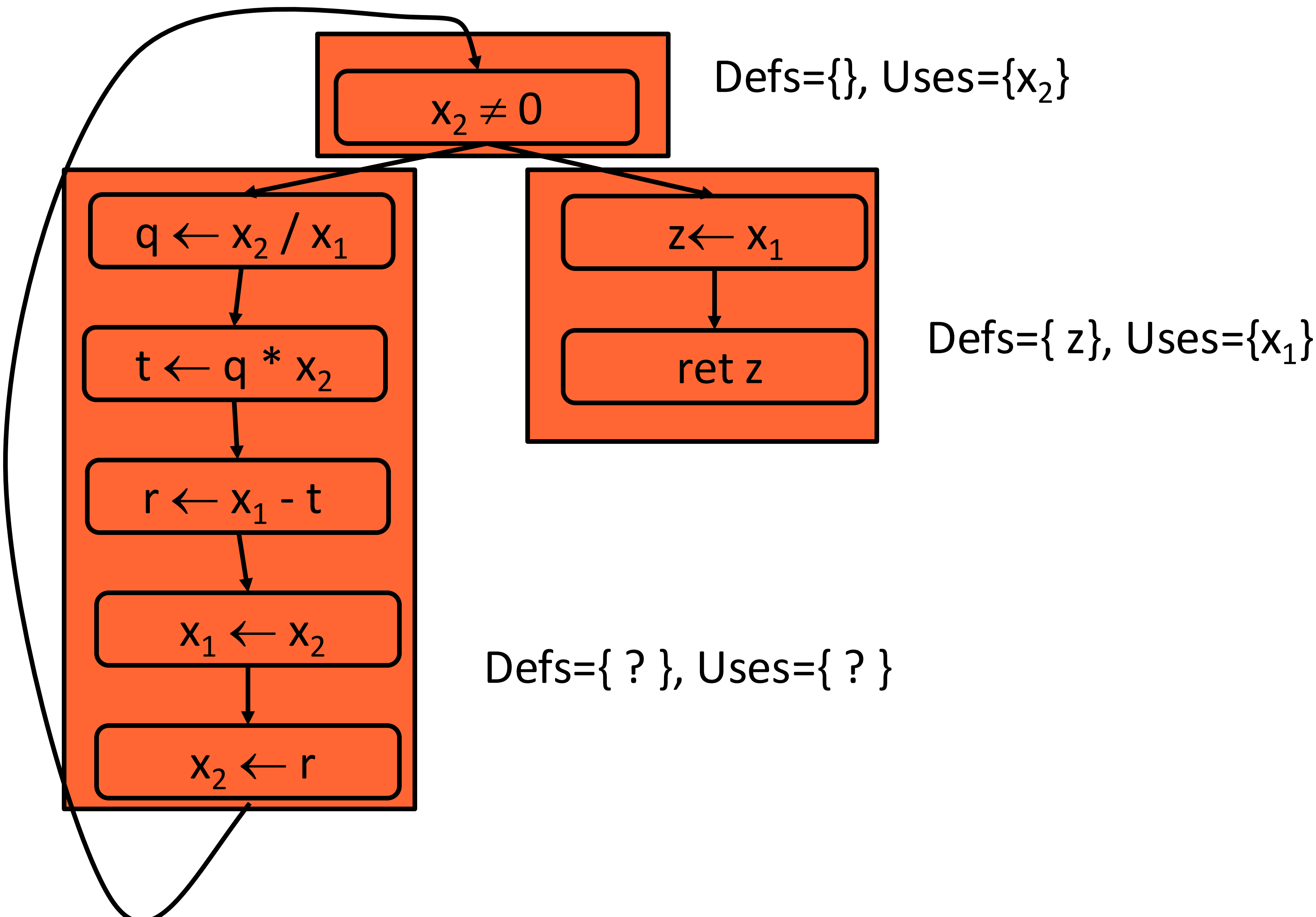
Using BasicBlocks

- 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:

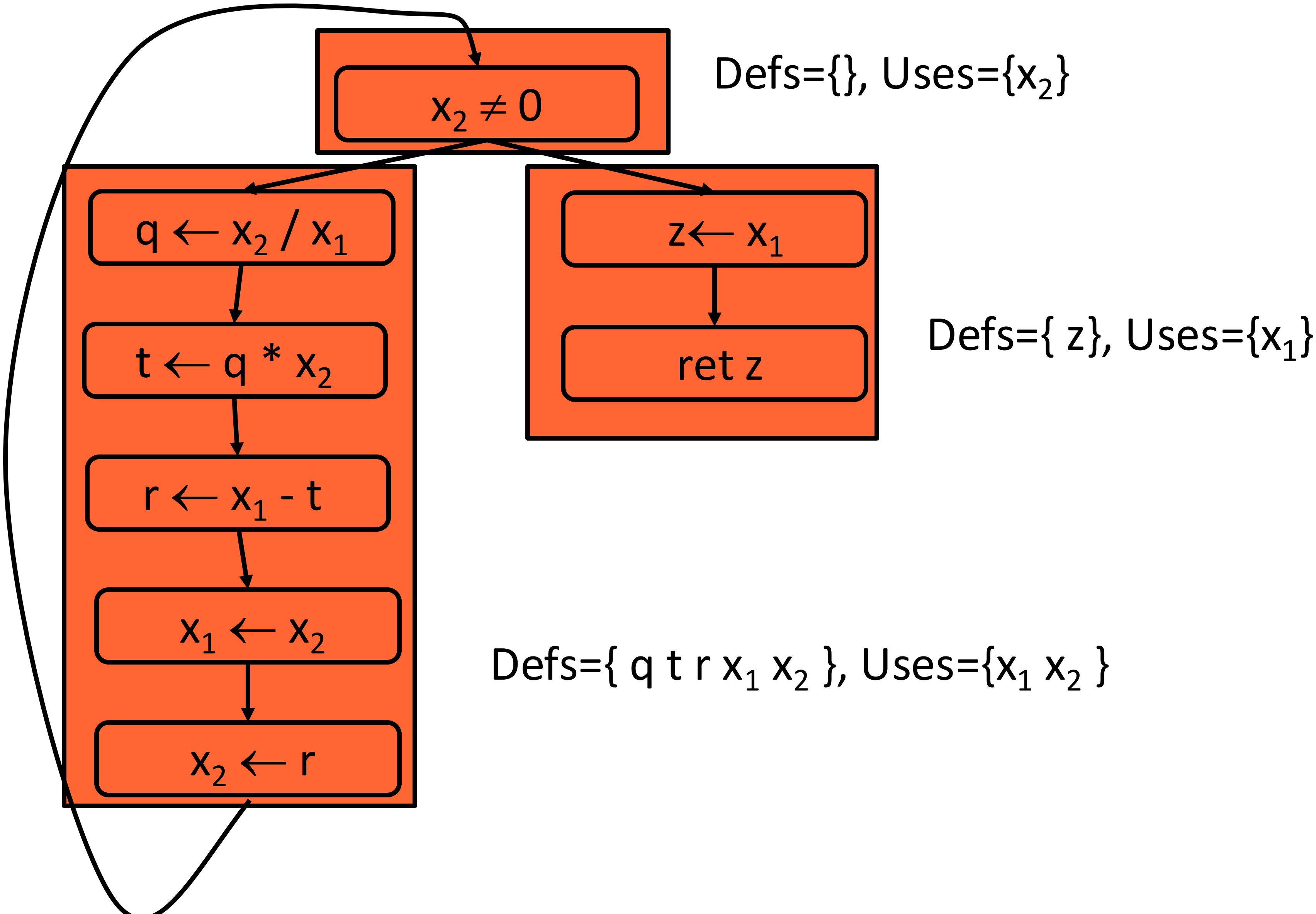
  In(p) = Uses(p) $\cup$ (In(**succ(p)** – Defs(p))
- And, then instead of in and out sets for each program point, you get ones for each basic block

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



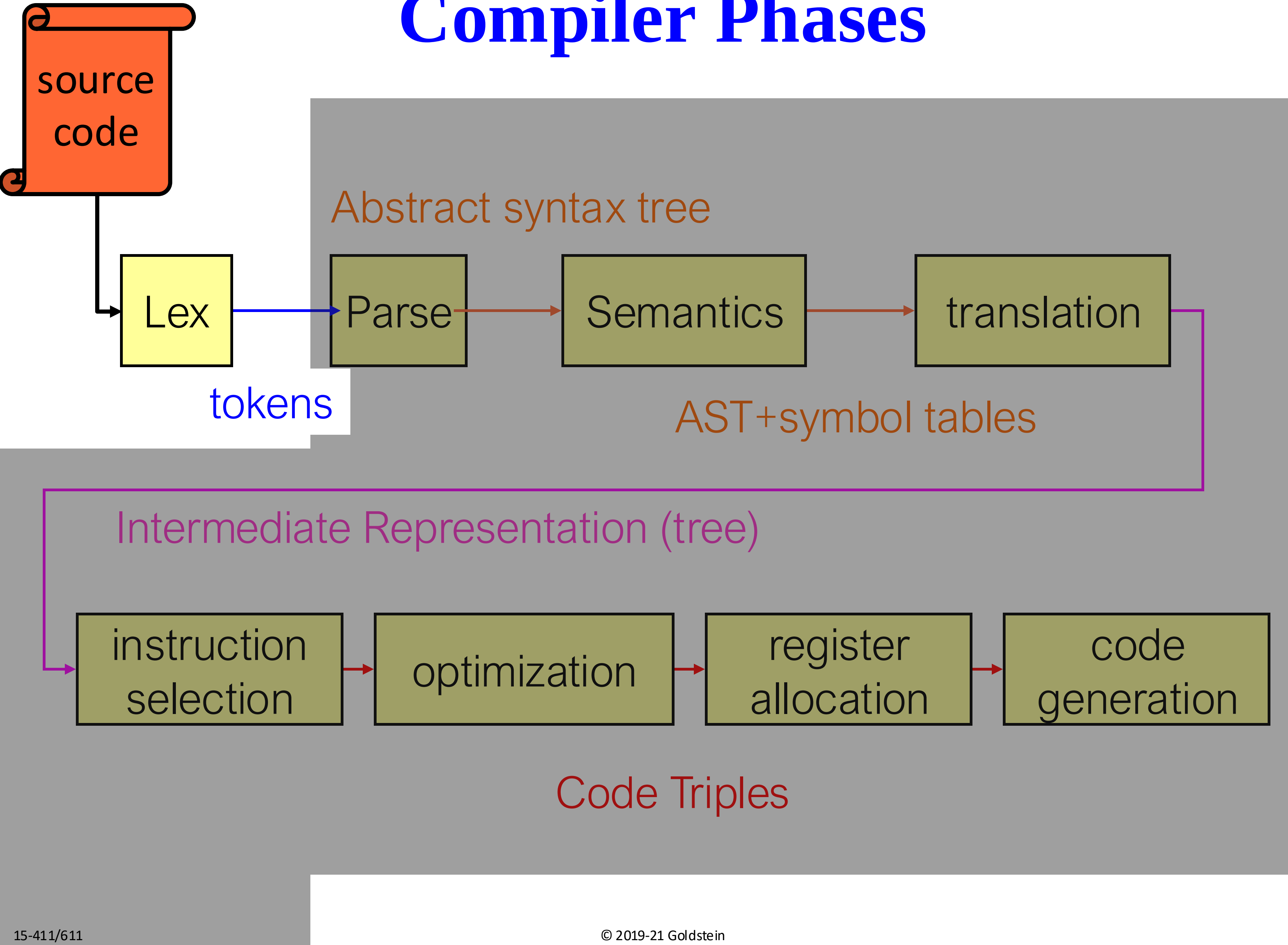
$x_2 \neq 0$
$q \leftarrow x_2 / x_1$
$t \leftarrow q * x_2$
$r \leftarrow x_1 - t$
$x_1 \leftarrow x_2$
$x_2 \leftarrow r$
$z \leftarrow x_1$
ret z
Defs={}, Uses={x_2}
Defs={ z}, Uses={x_1}
Defs={ ? }, Uses={ ? }


$x_2 \neq 0$
Defs={}, Uses={x_2}
$q \leftarrow x_2 / x_1$
$t \leftarrow q * x_2$
$r \leftarrow x_1 - t$
$x_1 \leftarrow x_2$
$x_2 \leftarrow r$
$z \leftarrow x_1$
ret z
Defs={ z}, Uses={x_1}
Defs={ q t r x_1 x_2 }, Uses={x_1 x_2 }

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:

```
digit     [0-9]
letter    [A-Za-z]
id        {letter} ({letter}|{digit})*

word      [^ \t\n]+
```

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

```
whitespace        [ \t\n]
%%

{whitespace}      ;

.                 { ECHO; }

%%

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

name

empty action

RE

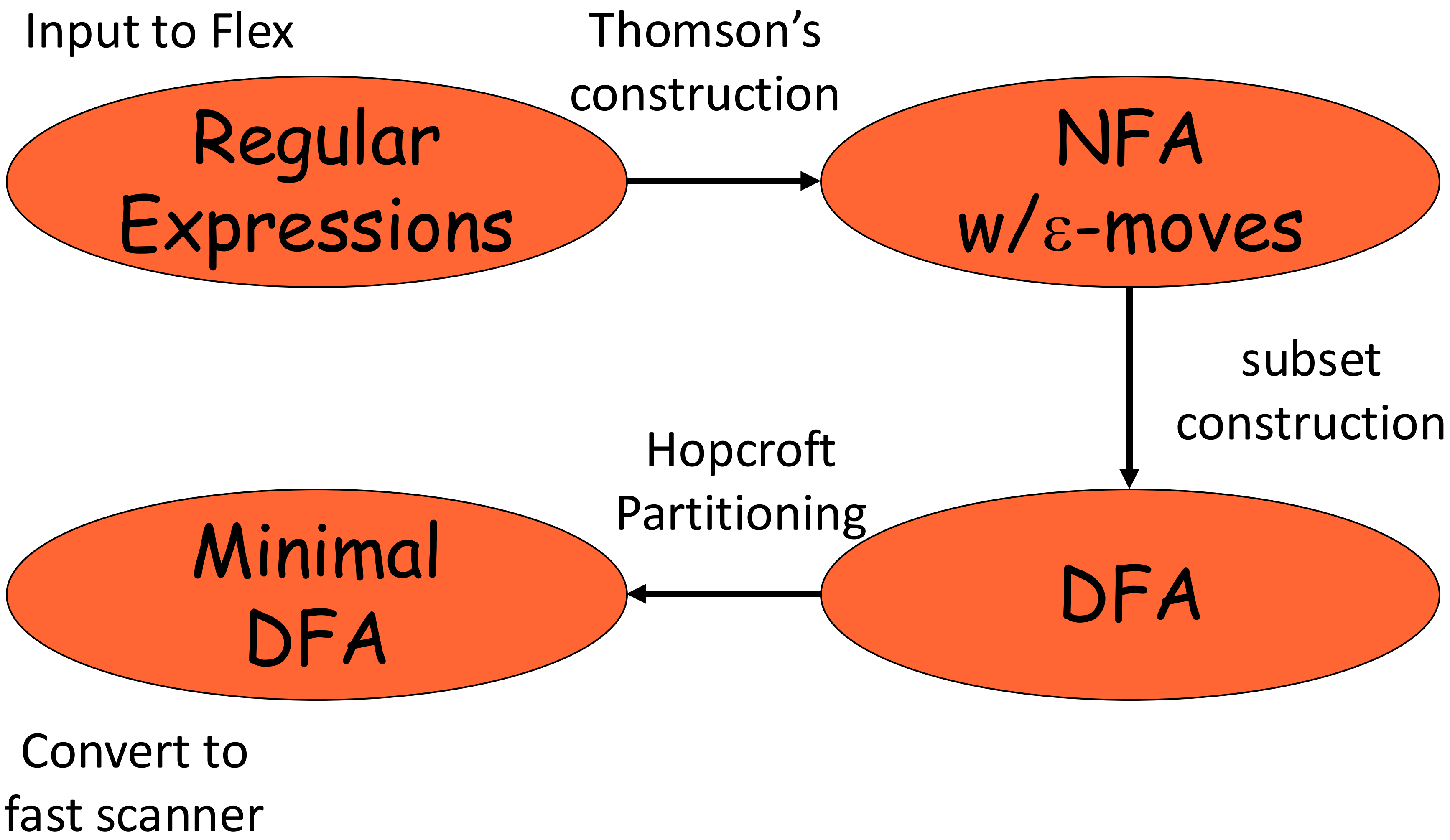
ECHO macro

Today – part 1

- Lexing
- Flex & other scanner generators
- **Regular Expressions**
- Finite Automata
- RE -> NFA
- NFA -> DFA
- DFA -> 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 $R_1 | R_2$
  - 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

  $R^A\ R^B$