Instruction Selection

15-411/15-611 Compiler Design

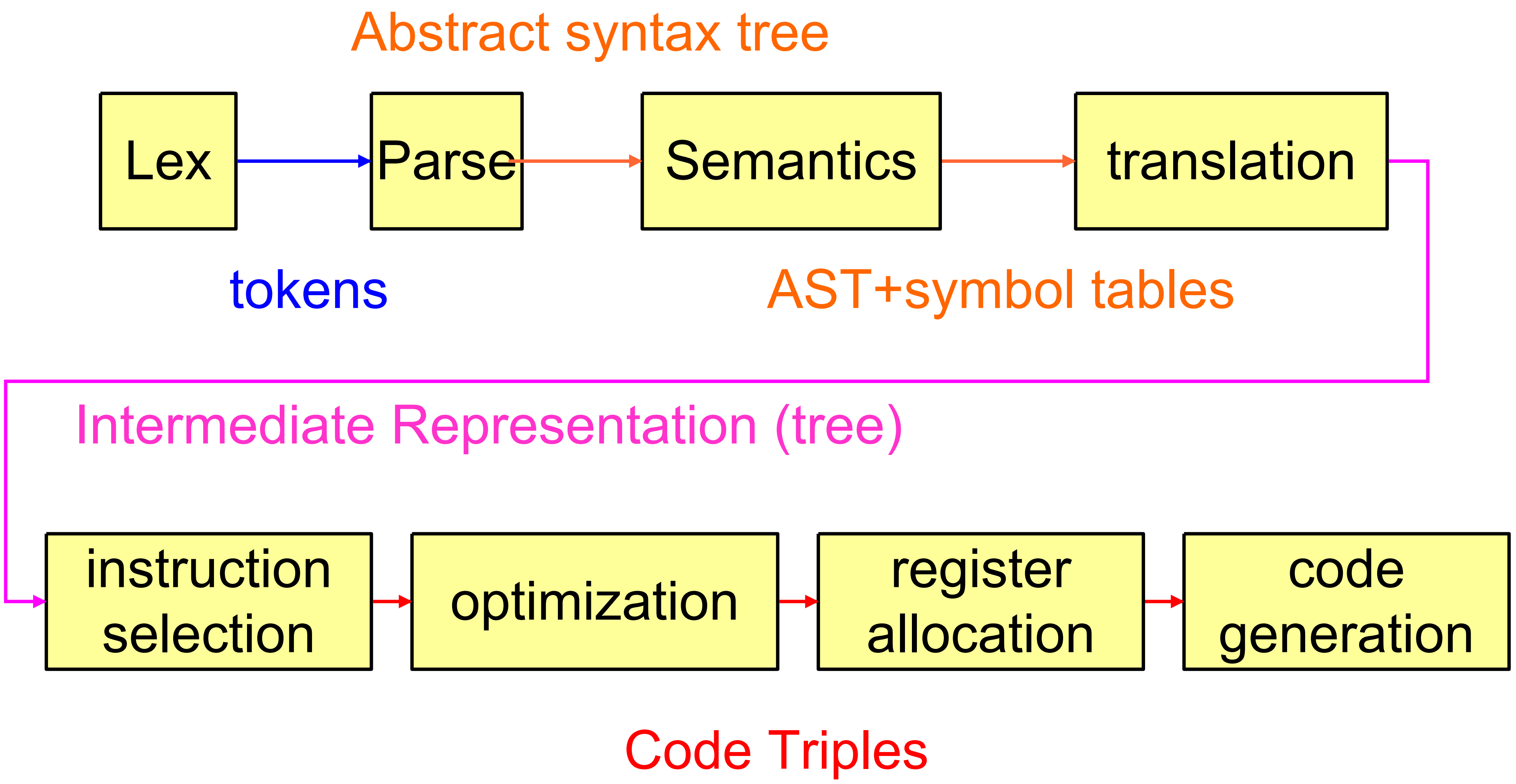
Seth Copen Goldstein

January 23, 2025

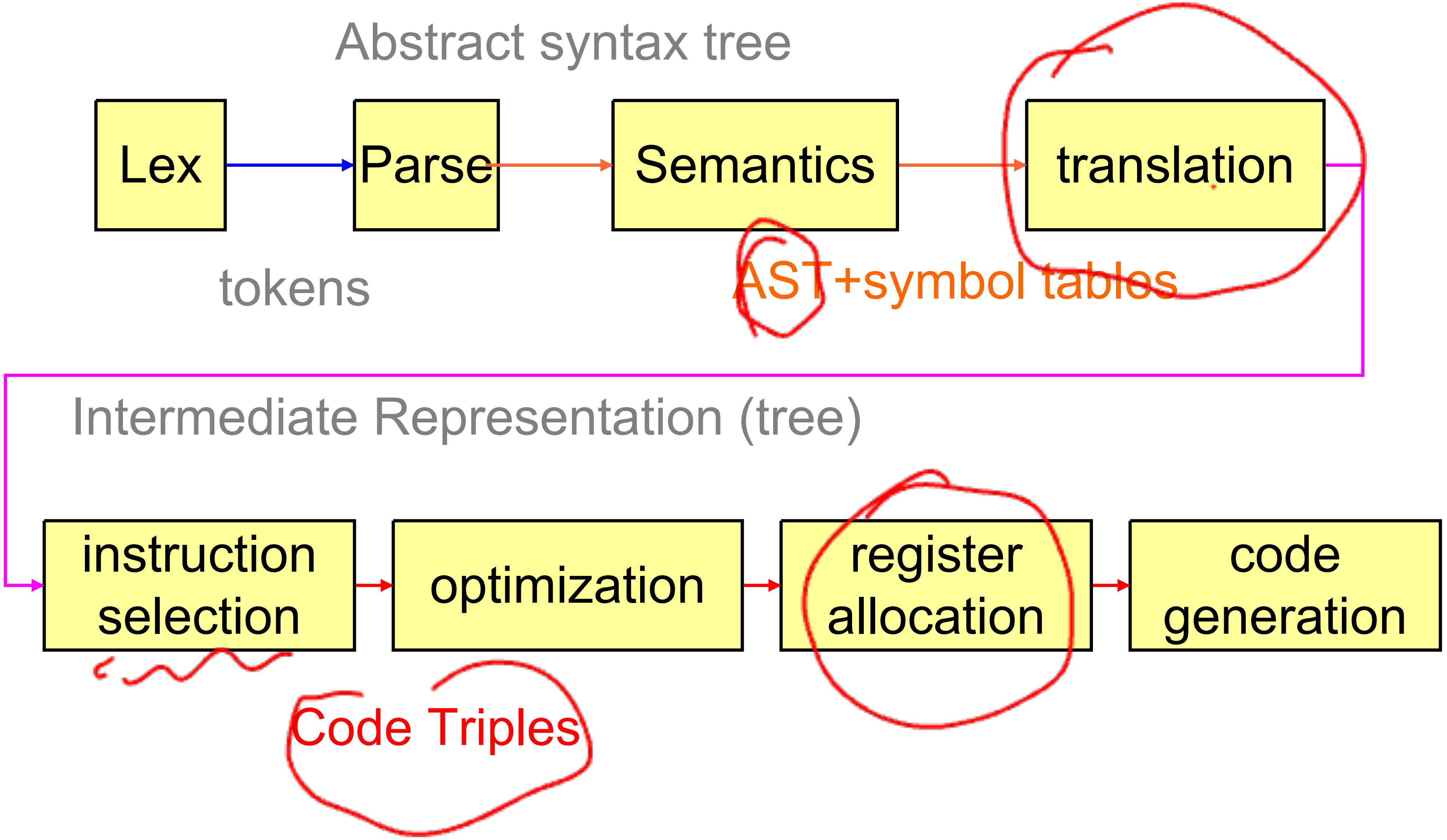
Today

- Context
- Abstract Assembly
- AST ⍰ IR
- Maximal Munch
- Issues
- Simple SSA
- x86 and 2-adr Instructions

Cartoon Compiler



Cartoon Compiler



Simple Source Language

- A language of assignments, expressions, and a return statement.
- Straight-line code
- Basically lab1 subset of C0

Simple Source Language

program := s_1 **;** s_2 **;** … s_n **;** sequence of statements

s := v **=** e assignment

| **return** ereturn

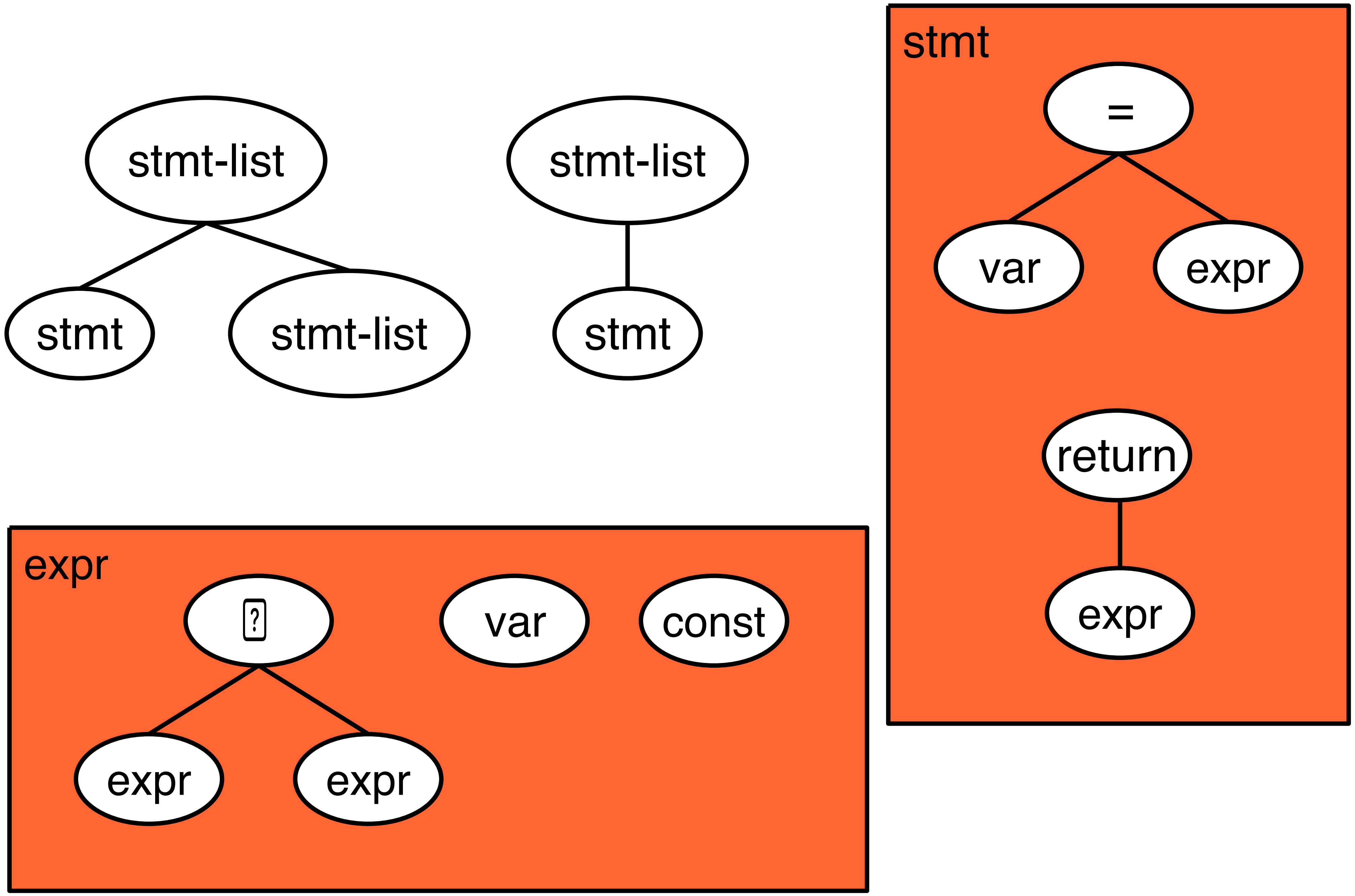
e := c constant

| v variable

| e_1 ⍰ e_2 binary operation

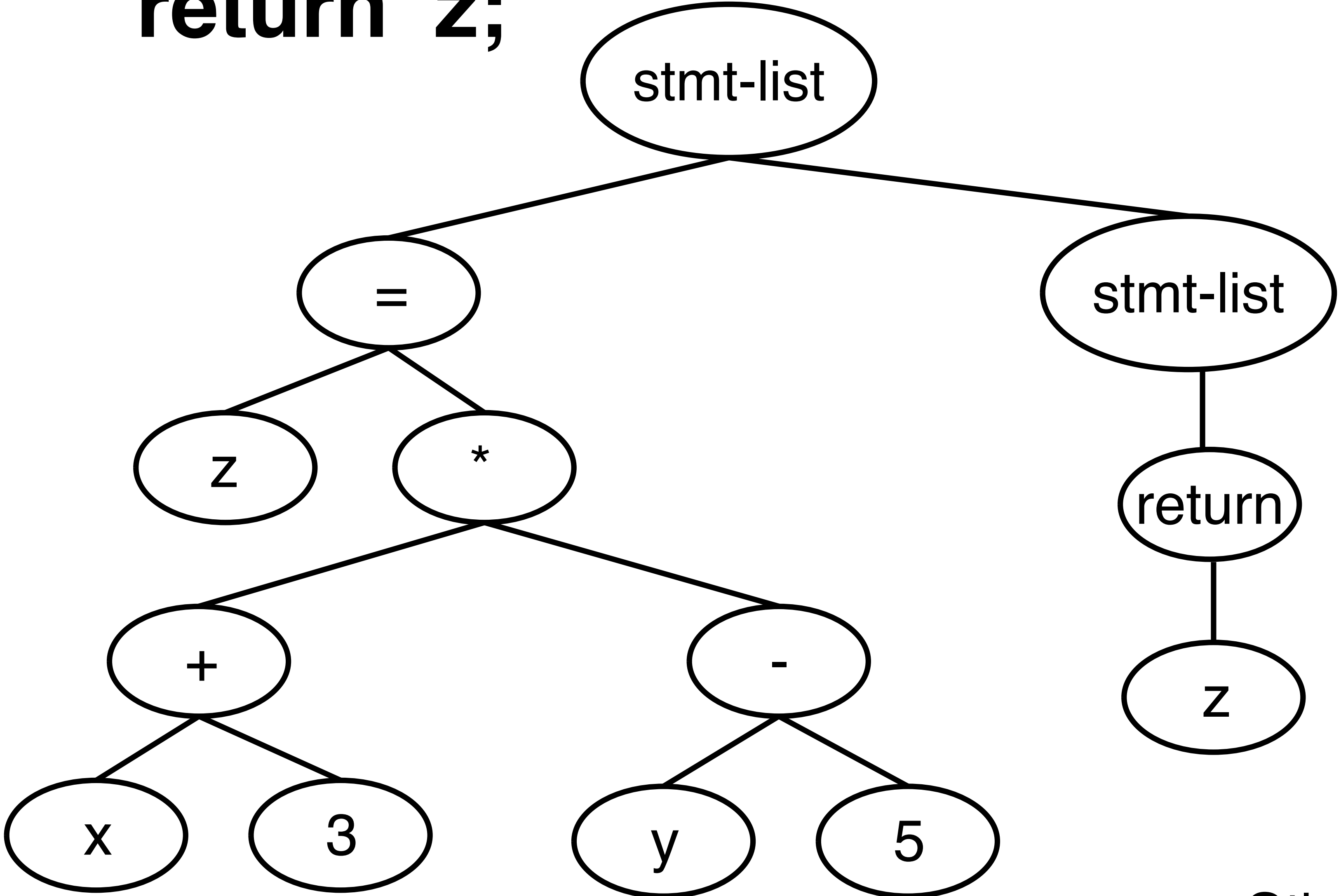
⍰ **+ | - |* | / | %**

Abstract Syntax Tree



Example

z = x + 3 * y – 5;
return z;



Other possibilites?

Today

- Context
- Abstract Assembly
- AST ⍰ IR
- Maximal Munch
- Issues
- Simple SSA
- x86 and 2-adr Instructions

Abstract Assembly as IR

- Lowering of AST
- Facilitate
  - Analysis & optimizations
  - Translation to actual assembly
- Features:
  - Unlimited number of “temporaries”
  - May not restrict how memory is used
  - Simple operations
  - May not restrict how constants are used
  - May specify certain “special registers”

Abstract Assembly as IR

- Features:
  - Unlimited number of “registers” (aka “temps”)
  - May (or may not) restrict how memory is used
  - Simple operations
  - May not restrict how constants are used
  - May specify certain “special registers”
- Form:

dest $\boxed{?}$ src_1 operator src_2

dest $\boxed{?}$ operator src_1

operator

src can be:

- constant
- temp
- special register
- memory

Abstract Assembly

program := i_1 i_2 … i_n seq of instructions

i := d $\boxed{?}$ s move

| d $\boxed{?}$ s_1 $\boxed{?}$ s_2 binop

| **return** return what is in **rax**

s := c intermediate ✓

| t temporary ✓

| r register ✓

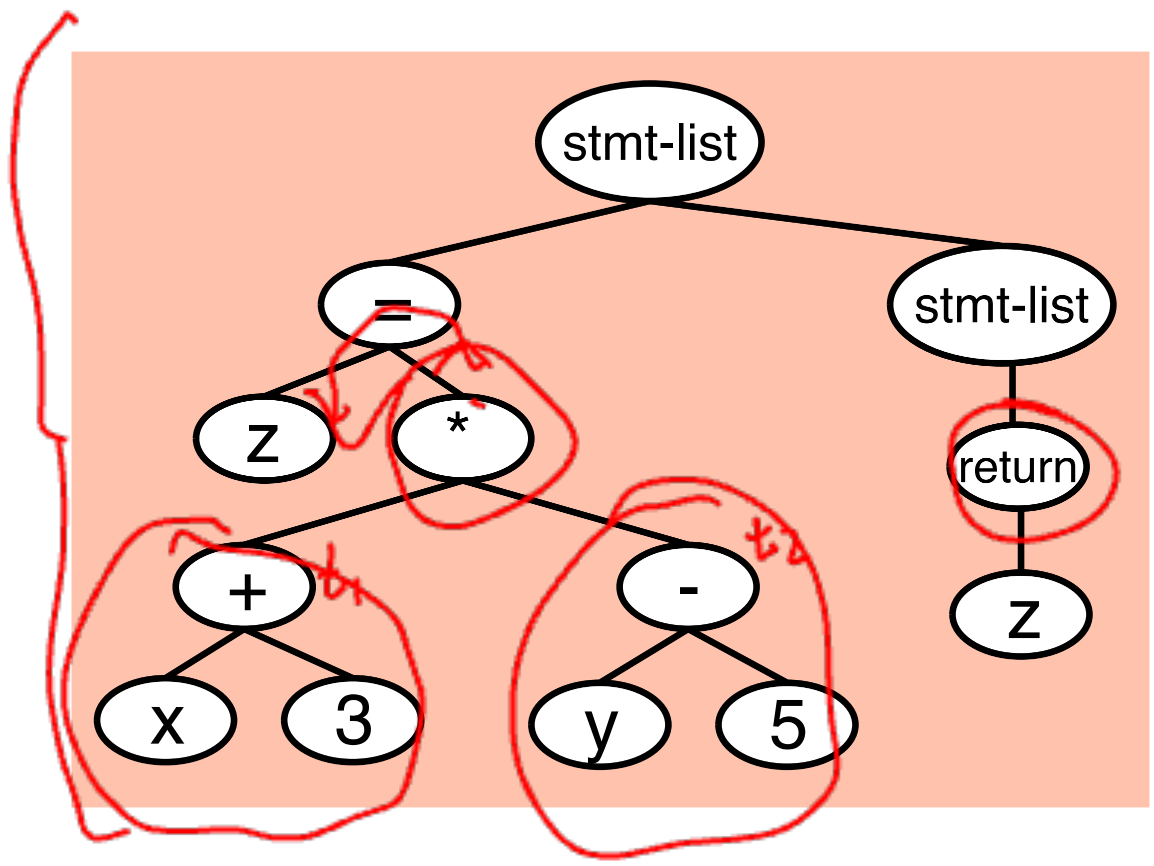
d := t

| r

$\boxed{?}$ **+ | - | * | / | %**

Example Goal

```
z = (x + 3) * (y – 5);
return z;
```

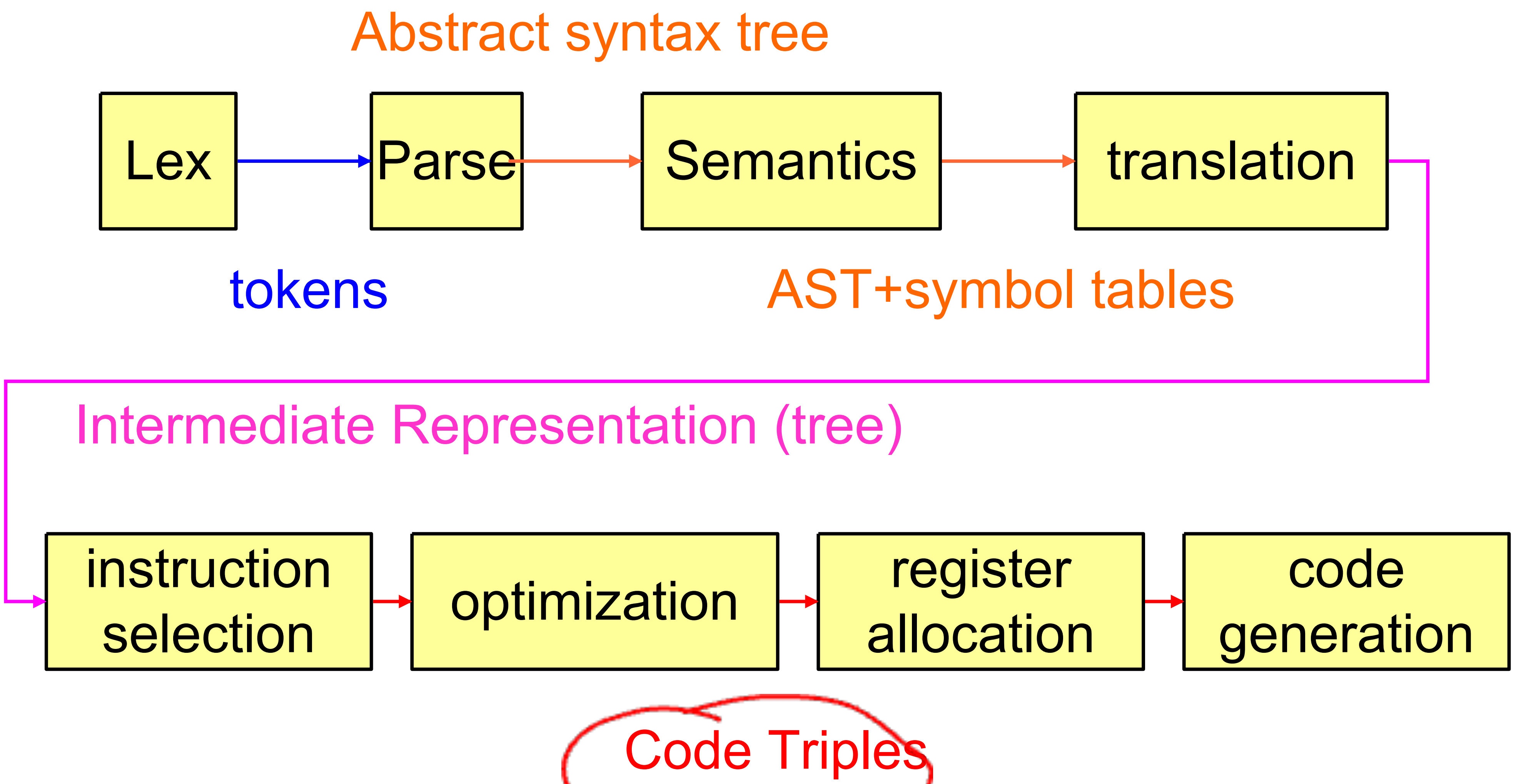


```
t1   ?  x + 3
t2   ?  y – 5
z    ?  t1 * t2
rax  ?  z
        return
```

Today

- Context
- Abstract Assembly
- AST ⍰ IR
- Maximal Munch
- Issues
- Simple SSA
- x86 and 2-adr Instructions

Cartoon Compiler

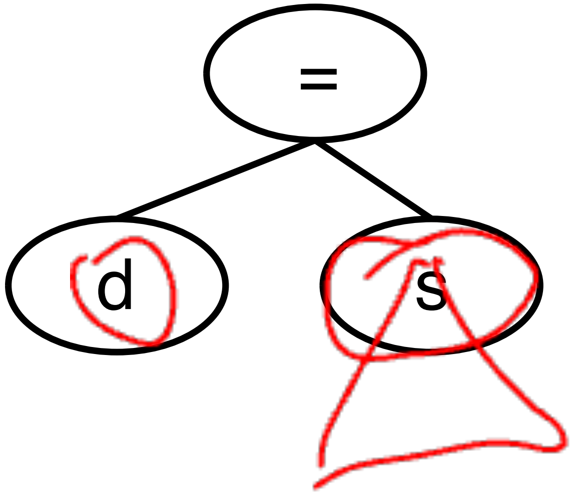


Alternatives abound

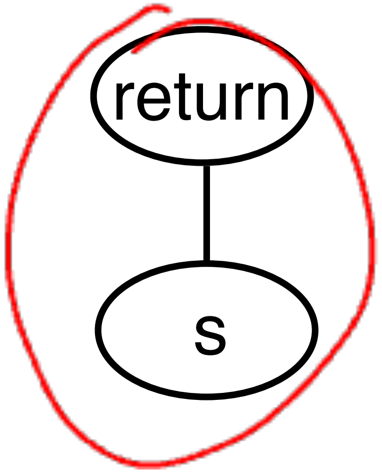
Translating AST to IR

- Converting from tree structured IR to sequence of instructions
  - Create temporary locations to store values
  - choose which operations we want
    - can combine or
    - breakup original operations
- Match portions of tree and convert to triple

Tree Patterns (aka Tiles)

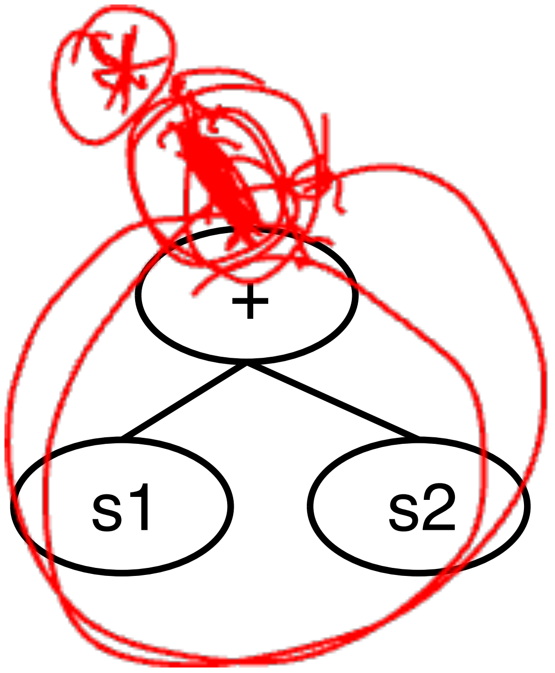


d ? s



rax ? s
ret

Tree Patterns

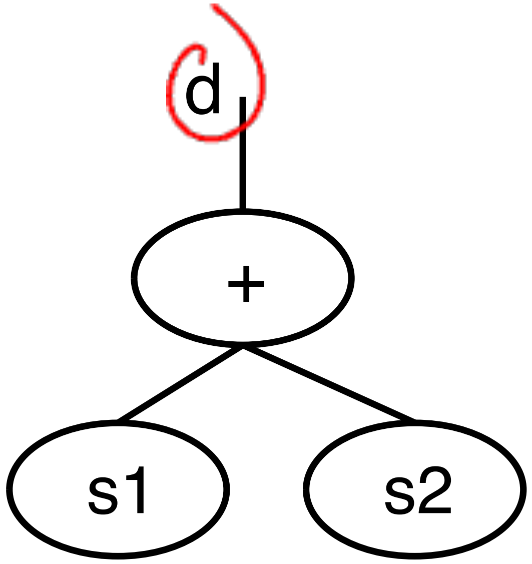


fresh t

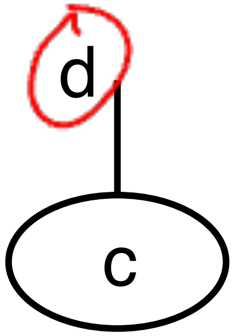
$t \leftarrow s1 + s2$

$d \leftarrow s1 + s2$

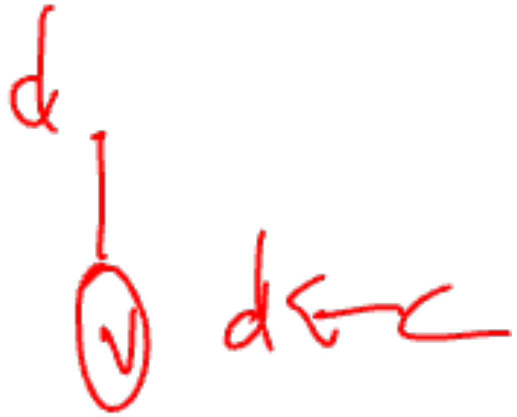
Tree Patterns



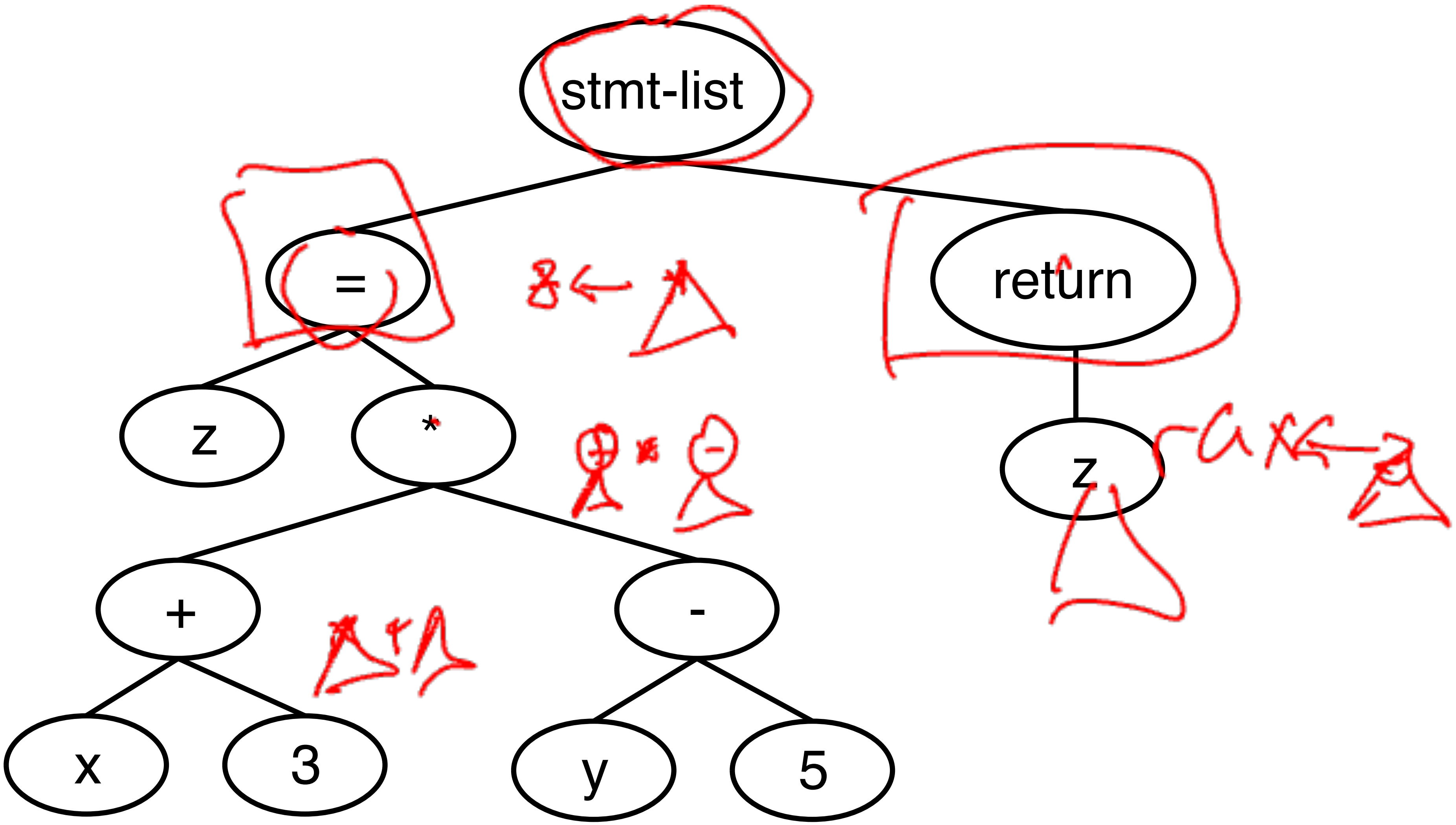
$d \boxed{?} s_1 + s_2$



$d \boxed{?} c$



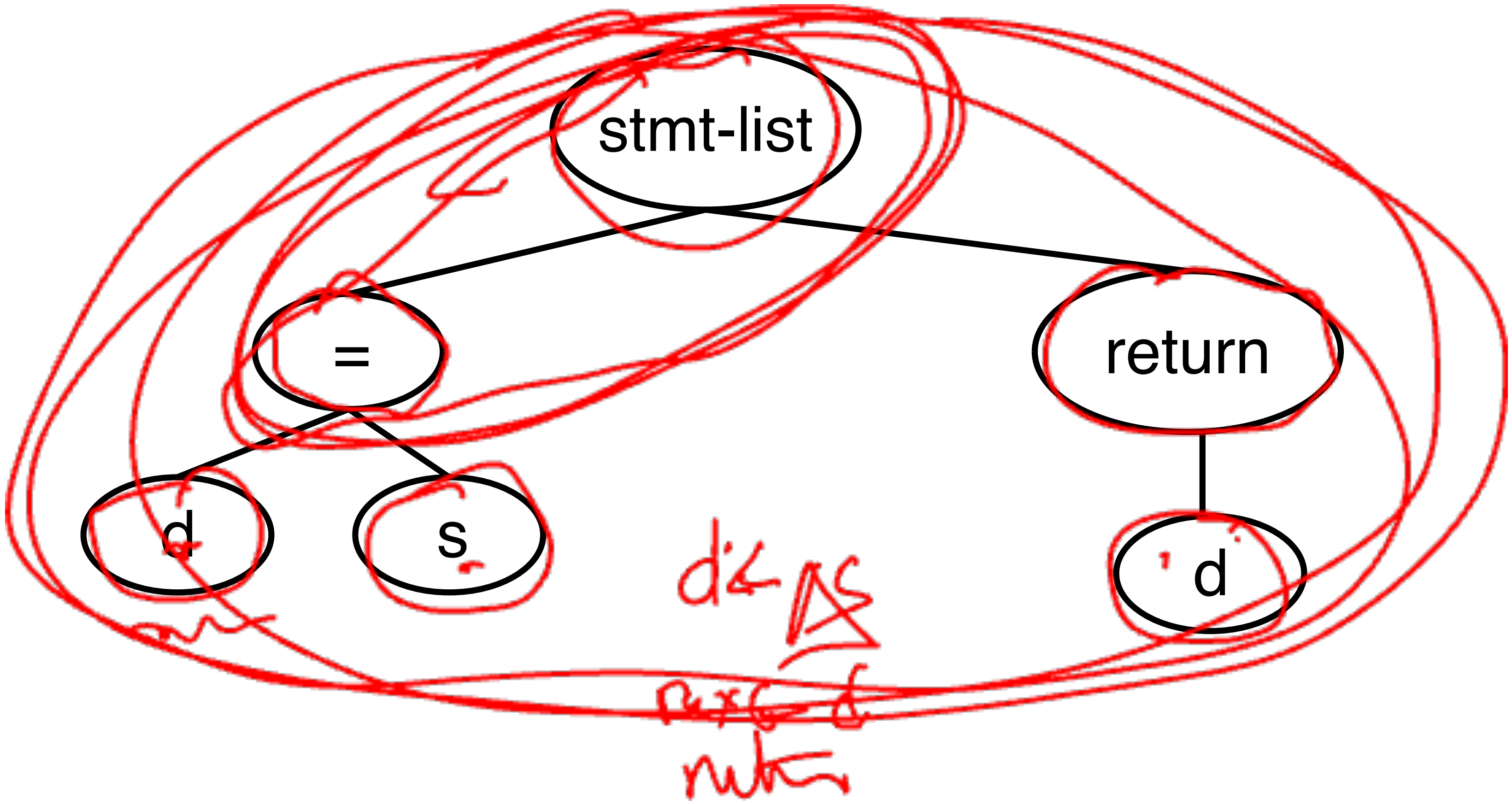
Tiling a Tree



$t_1 \leftarrow x$; $t_2 \leftarrow 3$; $t_3 \leftarrow t_1 + t_2$; $t_4 \leftarrow y$; $t_5 \leftarrow 5$; $t_6 \leftarrow t_4 - t_5$

$t_{?} \leftarrow t_3 * t_6$; $z \leftarrow t_{?}$; [illegible] $t_{?} \leftarrow z$; ret

Better Tiles



rax ? s
return

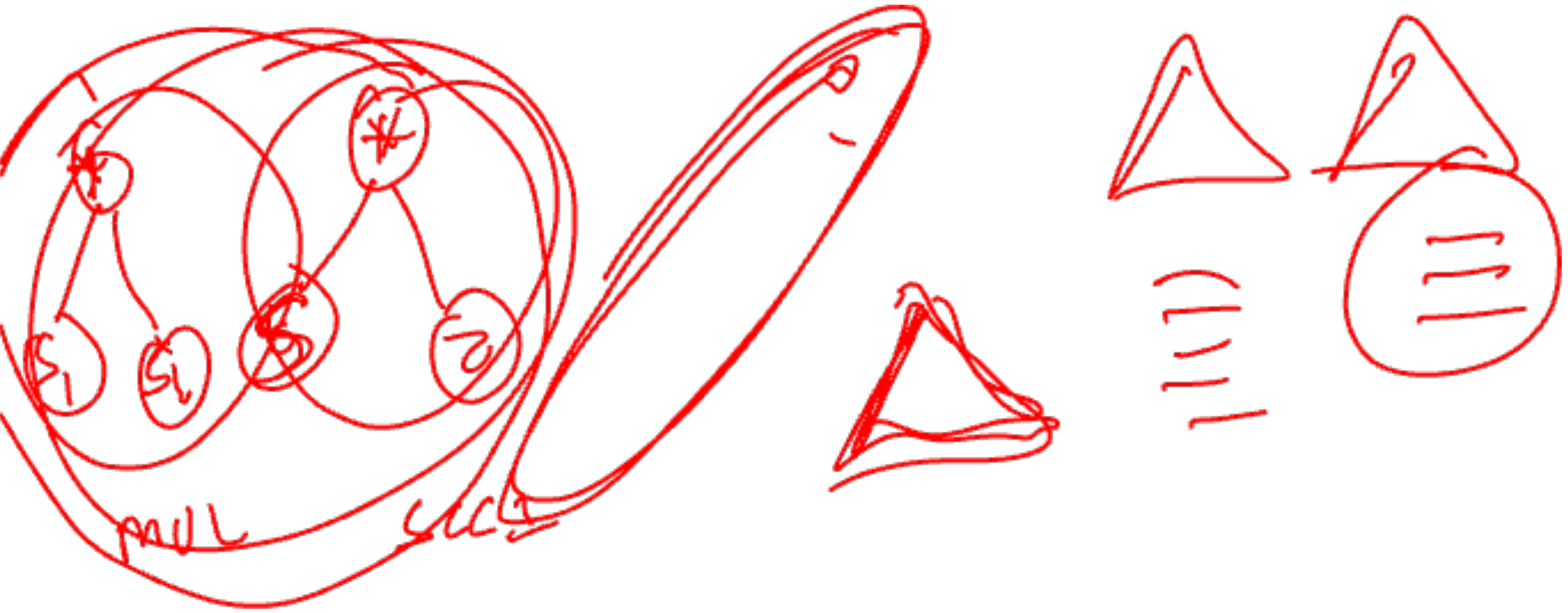
Correct?
If correct: better or worse?

Today

- Context
- Abstract Assembly
- AST ⍰ IR
- Maximal Munch
- Issues
- Simple SSA
- x86 and 2-adr Instructions

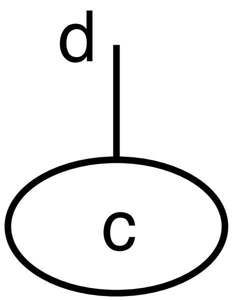
Maximal Munch

- recursively match tree
- At each step, pick "best" tile

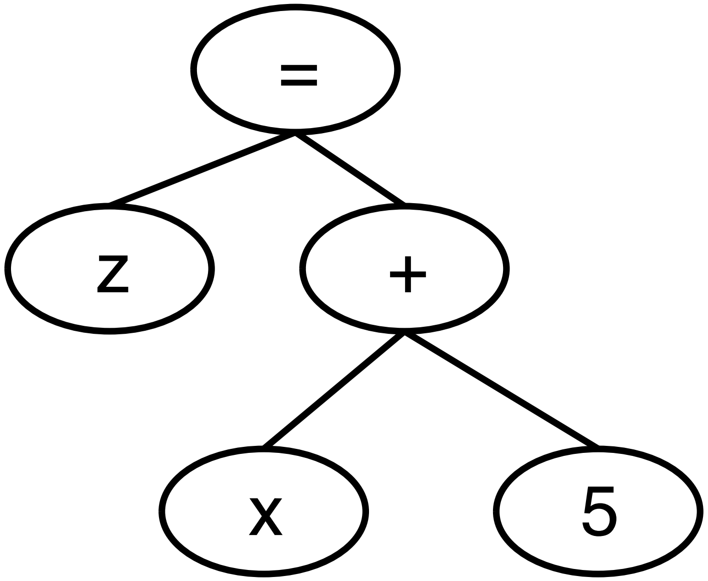


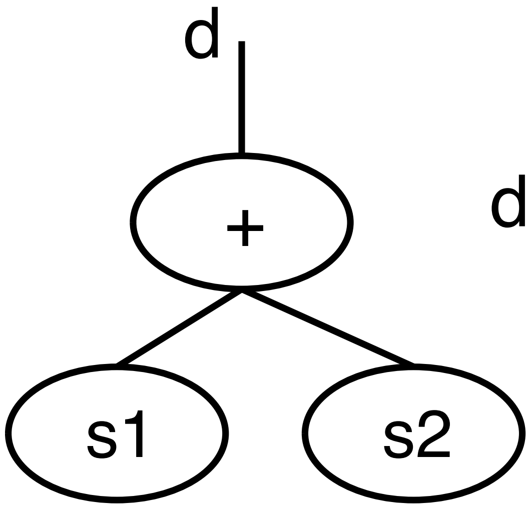
Maximal Munch

- recursively match tree
- At each step, pick “best” tile

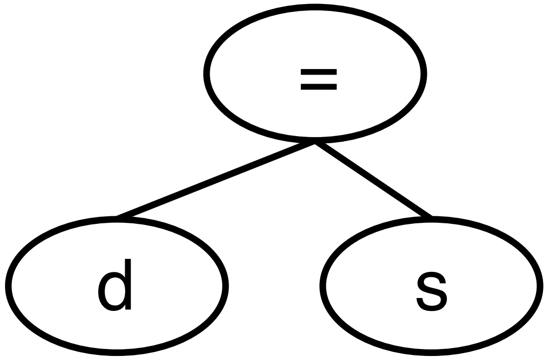


$d \boxed{?} c$





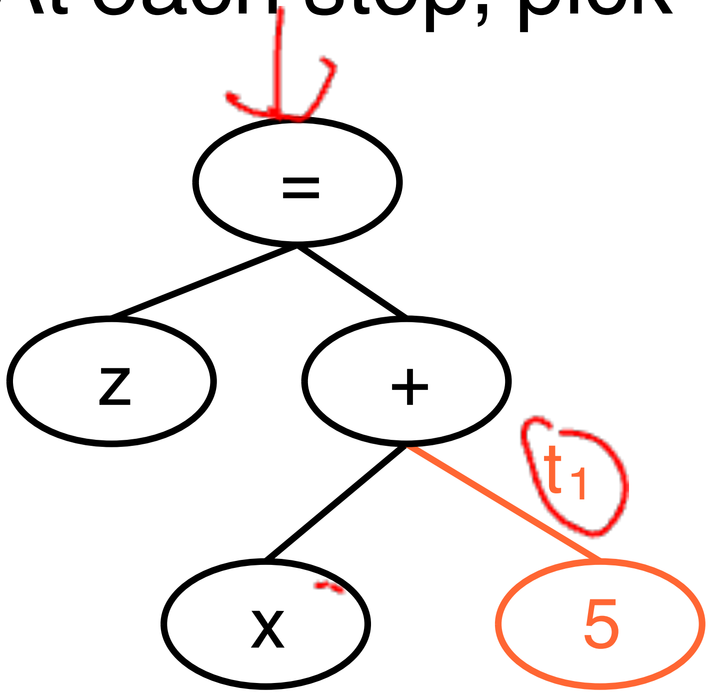
$d \boxed{?} s_1 + s_2$



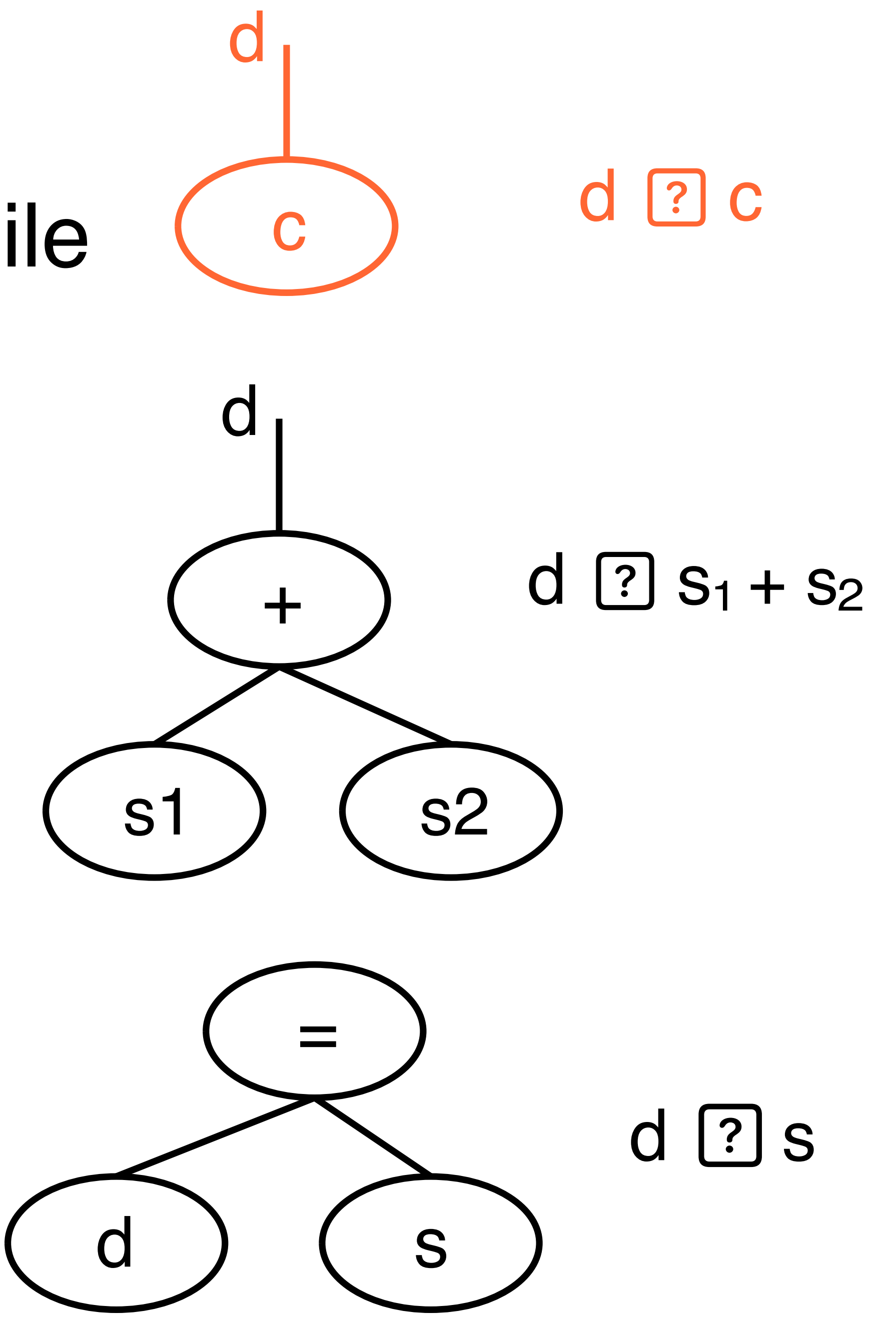
$d \boxed{?} s$

Maximal Munch

- recursively match tree
- At each step, pick “best” tile

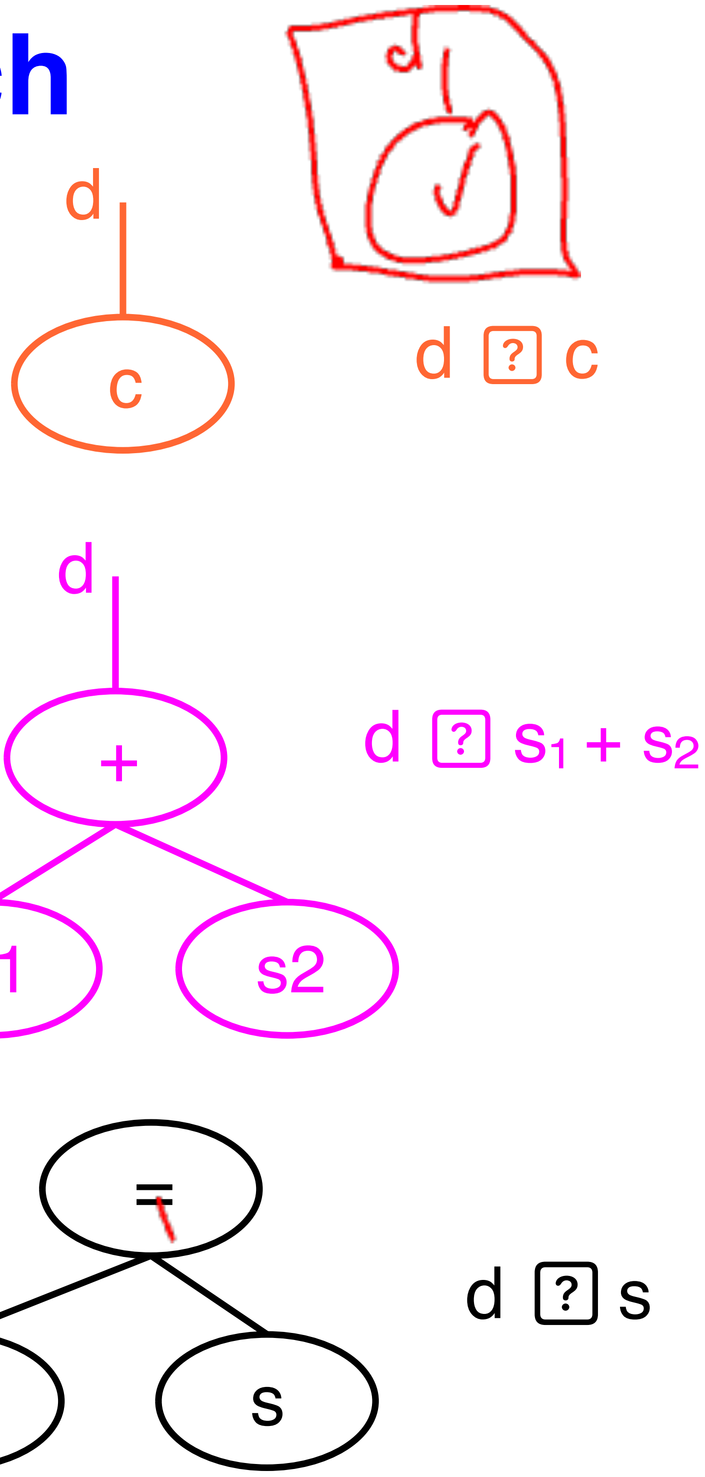


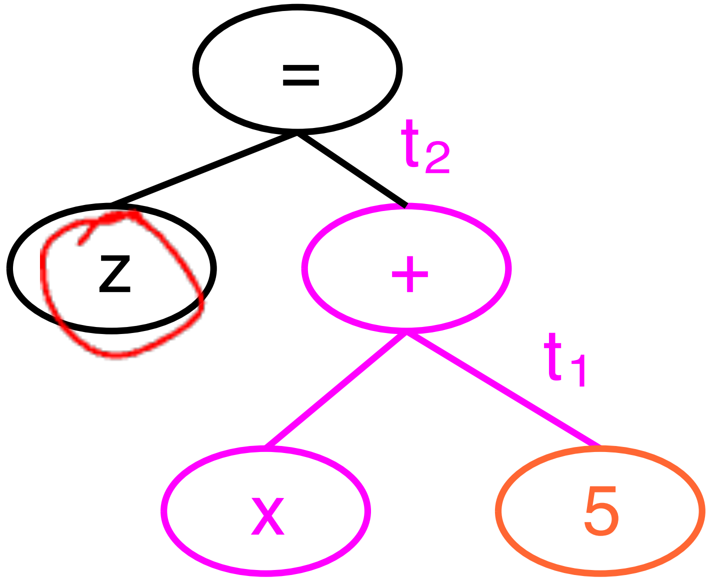
t_1 ⍰ 5



Maximal Munch

- recursively match tree
- At each step, pick "best" tile



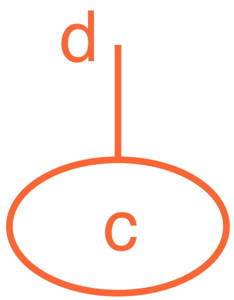


t₁ ? 5

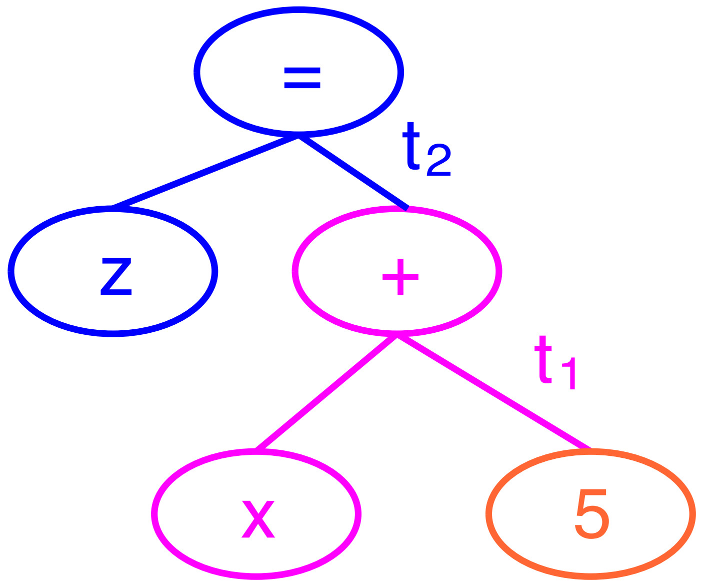
t₂ ? x + t₁

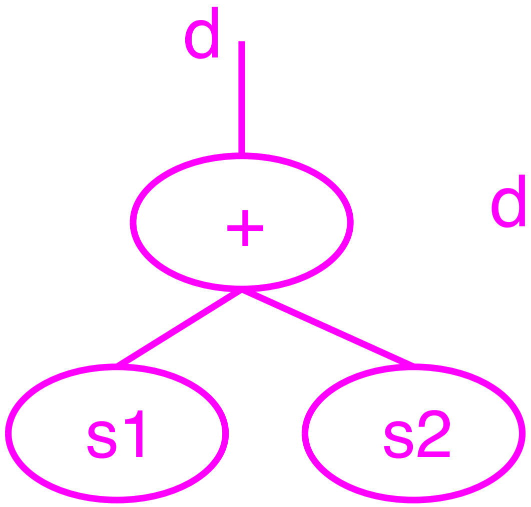
Maximal Munch

- recursively match tree
- At each step, pick “best” tile

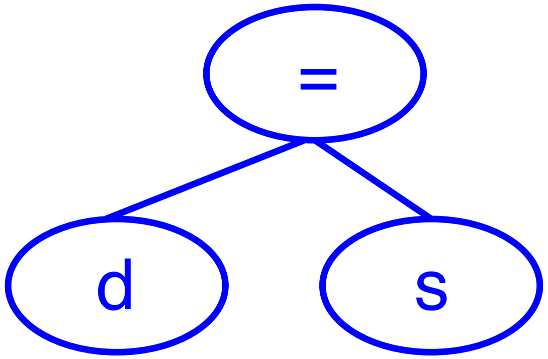


$d \boxed{?} c$





$d \boxed{?} s_1 + s_2$



$d \boxed{?} s$

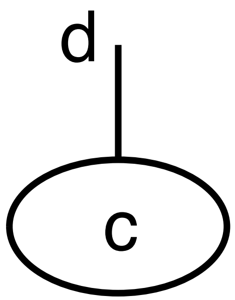
$t_1 \boxed{?} 5$

$t_2 \boxed{?} x + t_1$

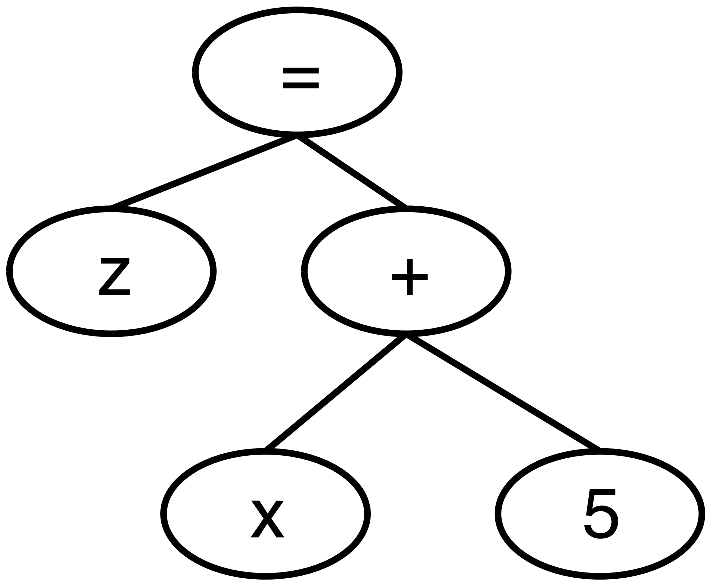
$z \boxed{?} t_2$

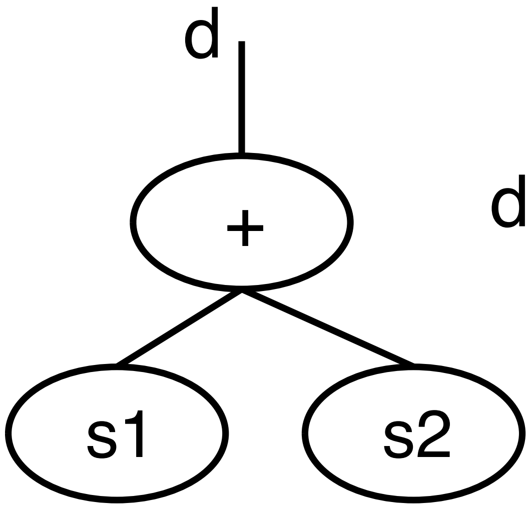
Maximal Munch

- recursively match tree
- At each step, pick “best” tile

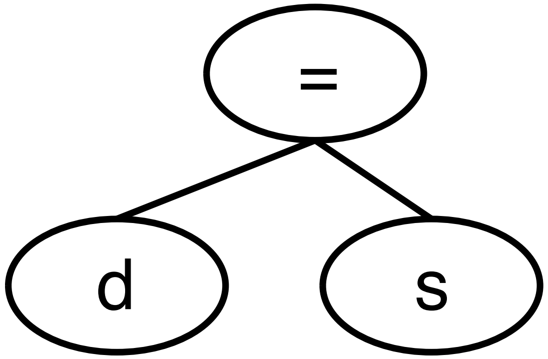


d ⍰ c





d ⍰ $s_1 + s_2$



d ⍰ s

t_1 ⍰ 5

t_2 ⍰ $x + t_1$

z ⍰ t_2

Maximal Munch

- recursively match tree
- At each step, pick "best" tile
- need to indicate what destinations are
  - choose either to supply destination
  - or generate a destination

codegen

$\textcircled{v}$

$\textcircled{t}$

e	codegen(d, e)
c	$d \leftarrow c$
v	$d \leftarrow v$

$e_1 \boxed{?} e_2$	codegen($\textcircled{t_1}$, e_1)
	codegen(t_2, e_2)
	$d \leftarrow t_1 \oplus t_2$

t_1, t_2, [illegible]

b_2 [illegible]

$$\begin{tikzcd} & \oplus \arrow[dl, dash] \arrow[dr, dash] & \\ v & & e \end{tikzcd}$$

s	codegen(s)
v = e	$v \leftarrow e$
return e	rax $\leftarrow e$ return

codegen

e	codegen(d, e)
c	d $\boxed{?}$ c
v	d $\boxed{?}$ v

e_1 $\boxed{?}$ e_2	codegen(t_1 , e_1)
	codegen(t_2 , e_2)
	d $\boxed{?}$ t_1 $\boxed{?}$ t_2

s	codegen(s)
v = e	codegen(v, e)

return e	codegen(rax, e)
	return

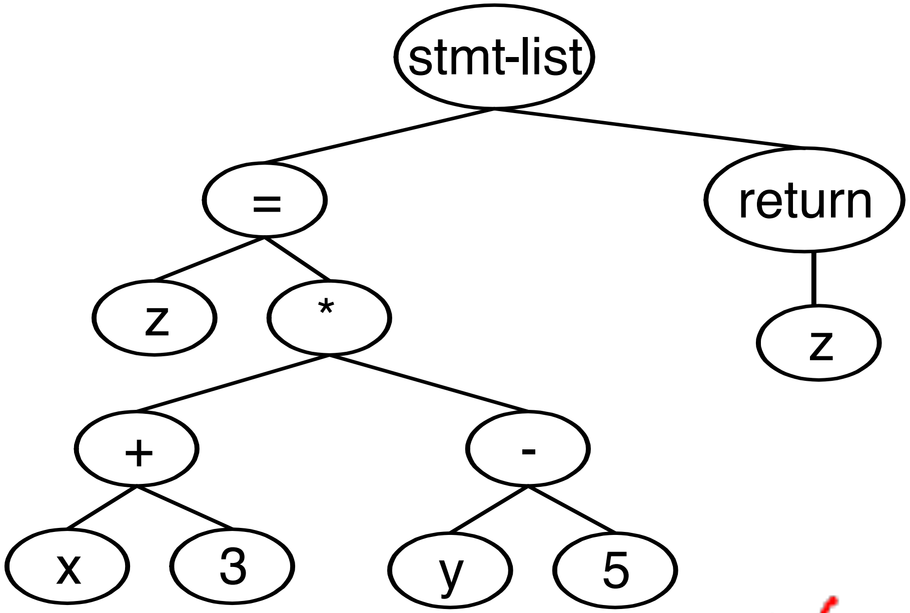
e	codegen(d, e)
c	d ¬ c
v	d ¬ x

e_1 Å e_2	codegen(t_1 , e_1)
	codegen(t_2 , e_2)
	d ¬ t_1 Å t_2

s	codegen(s)
v = e	codegen(v, e)

return e	codegen(rax, e)
	return

Example



e	codegen(d, e)
c	d ← c
v	d ← x
$e_1 \oplus e_2$	codegen(t_1, e_1) codegen(t_2, e_2) d ← $t_1 \oplus t_2$

s	codegen(s)
v = e	codegen(v, e)
return e	codegen(rax, e) return

codegen(= △) codegen(; return △)

cg(z, △) codegen(rax, z);

return

cg(t_1, + △) cg(t_2, - △) z ← t_1 * t_2

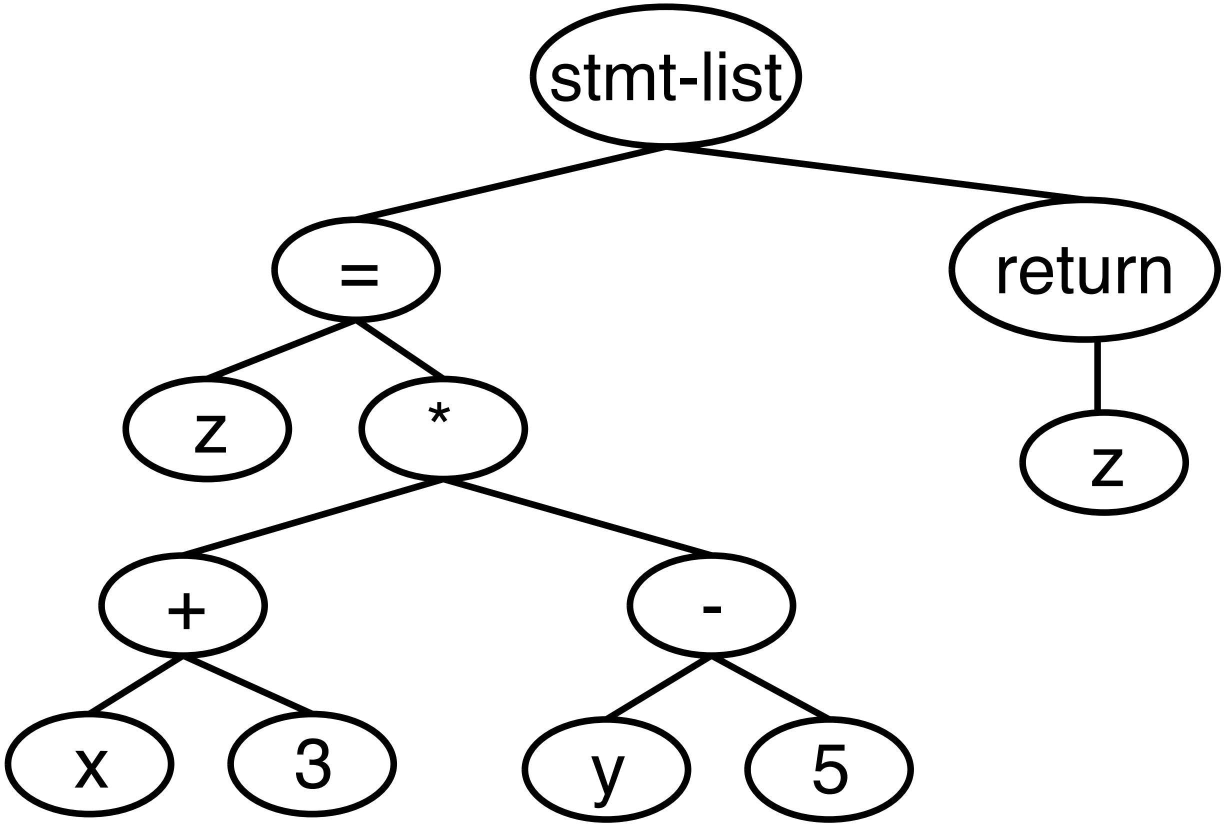
rax ← z
return

cg(t_3, x) cg(t_4, 3) t_1 ← t_3 + t_4

t_3 ← x t_4 ← 3 t_1 ← t_3 + t_4 cg(t_5, y) cg(t_6, 5) t_2 ← t_5 - t_6

⋮

Result



$t_3 \boxed{?} x$

$t_4 \boxed{?} 3$

$t_1 \boxed{?} t_3 + t_4$

$t_5 \boxed{?} y$

$t_6 \boxed{?} 5$

$t_2 \boxed{?} t_5 * t_6$

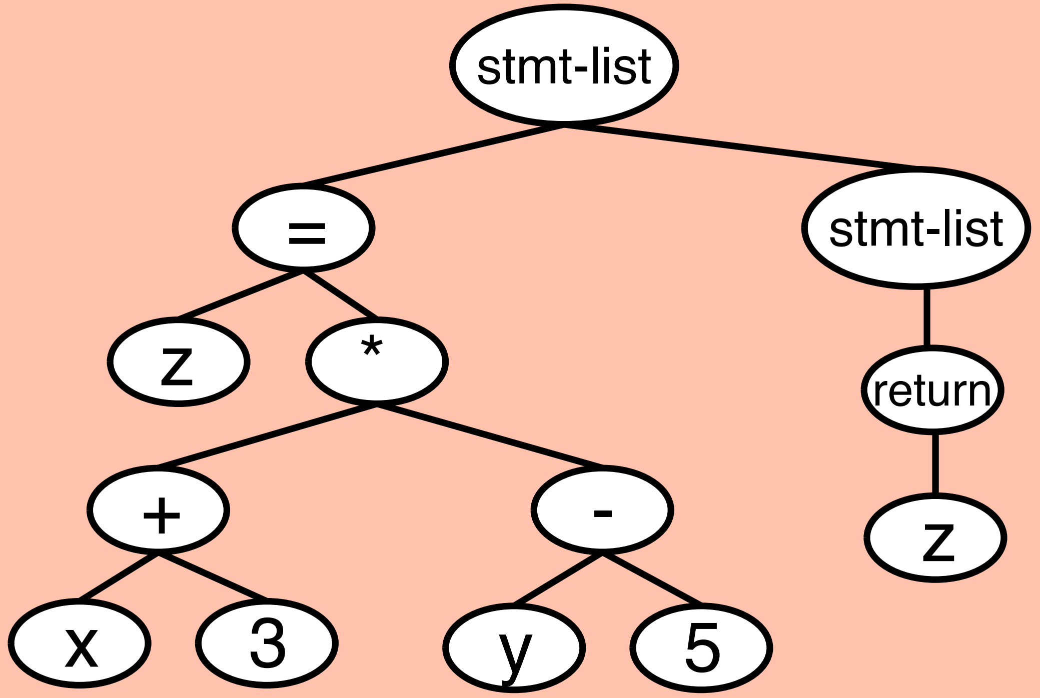
$z \boxed{?} t_1 * t_2$

$rax \boxed{?} z$

ret

Example Goal

```
z = x + 3 * y – 5;
return z;
```



```
t1   ?  x + 3
t2   ?  y – 5
z    ?  t1 * t2
rax  ?  z
        return
```

Goal

```
t1 ⍰ x + 3
t2 ⍰ y – 5
z  ⍰ t1 * t2
rax   ⍰ z
      return
```

What we got

t_3 ⍰ x

t_4 ⍰ 3

t_1 ⍰ t_3 + t_4

t_5 ⍰ y

t_6 ⍰ 5

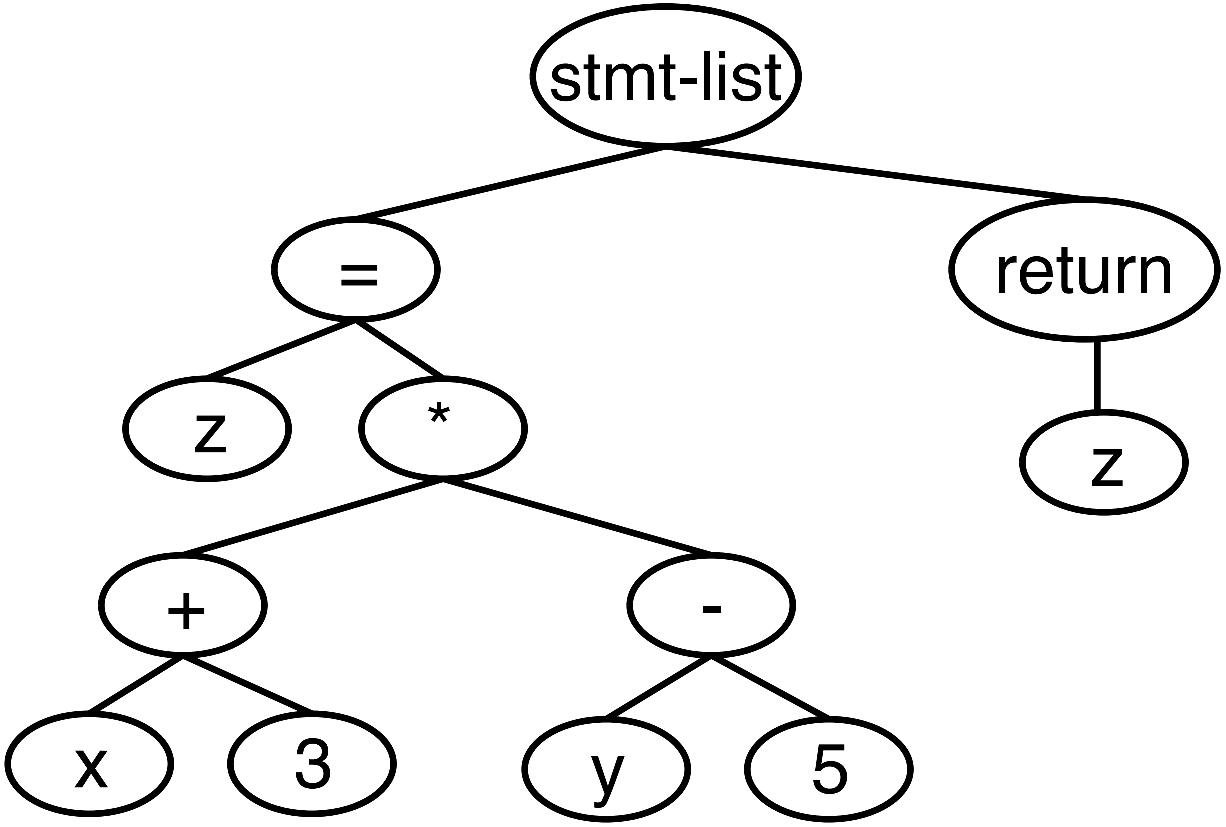
t_2 ⍰ t_5 * t_6

z ⍰ t_1 * t_2

rax ⍰ z

ret

How Can we Improve this?



$t_3 \boxed{?} x$

$t_4 \boxed{?} 3$

$t_1 \boxed{?} t_3 + t_4$

$t_5 \boxed{?} y$

$t_6 \boxed{?} 5$

$t_2 \boxed{?} t_5 * t_6$

$z \boxed{?} t_1 * t_2$

rax $\boxed{?}$ z

ret

How Can we Improve this?

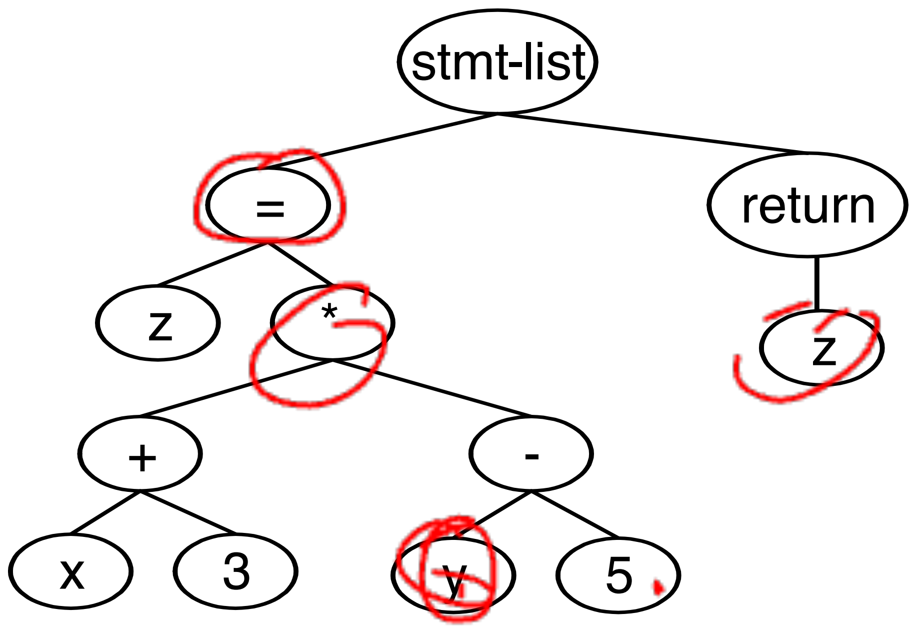
- Investigate generating a source operand
- Special cases
- Don’t bother?

Generating Destinations

e	codegen(e)	up
c		c
v		v
e_1 $\boxed{?}$ e_2	t_1 - codegen(e_1) $\boxed{?}$ codegen(e_2)	t_1

s	codegen(s)
v = e	v $\boxed{?}$ codegen(e)
return e	rax $\boxed{?}$ codegen(e) return

Example



e	codegen(e)	up
c		c
v		v
$e_1 \oplus e_2$	t_1 = codegen(e_1) $\oplus$ codegen(e_2)	t_1

s	codegen(s)
v = e	v ← codegen(e)
return e	rax ← codegen(e) return

cgen(= △)

cg z ← cg(* △)

rax ← E
ret

cg(* △) ⇒ t1 ← cg(+ △) * (cg(- △)) t3 ← y - 5

t2 ← cg(x △) + cg(3 △)

t2 ← x + 3

z ← t2 * t3

Special Cases

e	codegen(d, e)
c	d ⍰ c
v	d ⍰ x
c ⍰ e_2	codegen(t_2 , e_2) d ⍰ c ⍰ t_2
e_1 ⍰ c	codegen(t_1 , e_1) d ⍰ t_1 ⍰ c
v ⍰ e_2	codegen(t_2 , e_2) d ⍰ v ⍰ t_2
e_1 ⍰ v	codegen(t_1 , e_1) d ⍰ t_1 ⍰ v

e_1 ⍰ e_2	codegen(t_1 , e_1)
	codegen(t_2 , e_2)
	d ⍰ t

Generally not recommended

The “don’t bother” case

- What should we really do?

Separation of Concerns

KISS

$t_3 \boxed{?} x$

$t_4 \boxed{?} 3$

$t_1 \boxed{?} t_3 + t_4$

$t_5 \boxed{?} y$

$t_6 \boxed{?} 5$

$t_2 \boxed{?} t_5 * t_6$

$z \boxed{?} t_1 * t_2$

$rax \boxed{?} z$

ret

	e	codegen(d, e)
	c	d ⍰ c
	v	d ⍰ x
Constant Propagation	c ⍰ e_2	codegen(t_2 , e_2) d ⍰ c ⍰ t_2
	e_1 ⍰ c	codegen(t_1 , e_1) d ⍰ t_1 ⍰ c
Copy Propagation	v ⍰ e_2	codegen(t_2 , e_2) d ⍰ v ⍰ t_2
	e_1 ⍰ v	codegen(t_1 , e_1) d ⍰ t_1 ⍰ v

	e_1 ⍰ e_2	codegen(t_1 , e_1)
		codegen(t_2 , e_2)
		d ⍰ t_1 ⍰ t_2

Constant Propogation

$t_3 \boxed{?}\ x$

~~$t_4 \boxed{?}\ 3$~~

$t_1 \boxed{?}\ t_3 + \cancel{t_4}\ 3$

$t_5 \boxed{?}\ y$

~~$t_6 \boxed{?}\ 5$~~

$t_2 \boxed{?}\ t_5 * \cancel{t_6}\ 5$

$z \boxed{?}\ t_1 * t_2$

$rax \boxed{?}\ z$

ret

Copy Propogation

~~t_3 ⍰ x~~

t_1 ⍰ $\cancel{t_3}$ x + 3

~~t_5 ⍰ y~~

t_2 ⍰ $\cancel{t_5}$ y * 5

z ⍰ t_1 * t_2

rax ⍰ z

ret

Have to be careful

- Constant propagation:

  $x \boxed{?} 5$

  $y \boxed{?} x - 4$

  ~~$x \boxed{?} y + 7$~~

  $z \boxed{?} \cancel{x}$ 8

- Copy Propagation:

  $x \boxed{?} y$

  $y \boxed{?} u - 4$

  $z \boxed{?} \underset{y}{\cancel{x}} + 7$

Have to be careful

- Constant propagation:
  - Can't just replace all x's with 5
  - Stop if x is redefined

$x \boxed{?} 5$

$y \boxed{?} x - 4$

$x \boxed{?} y + 7$

$z \boxed{?} x$

- Copy Propagation:

$x \boxed{?} y$

$y \boxed{?} u - 4$

$z \boxed{?} x + 7$

Have to be careful

- Constant propagation:
  - Can't just replace all x's with 5
  - Stop if x is redefined

$x \boxed{?} 5$

$y \boxed{?} x - 4$

$x \boxed{?} y + 7$

$z \boxed{?} x$

- Copy Propagation:
  - Can't just replace all x's with y's
  - Stop if x or y is redefined

$x \boxed{?} y$

$y \boxed{?} u - 4$

$z \boxed{?} x + 7$

Today

- Context
- Abstract Assembly
- AST ⍰ IR
- Maximal Munch
- Issues
- Simple SSA
- x86 and 2-adr Instructions

Static Single Assignment

- Must keep track of what definition each use refers to in order to properly do constant/copy propagation.
- Much simpler if only one definition for each name.
- SSA: Each name is assigned in only one location.

Static Single Assignment

- Must keep track of what definition each use refers to in order to properly do constant/copy propogation.
- Much simpler if only one definition for each name.
- SSA: Each name is **assigned** in only one location.
- Easy for fresh temporaries

e	codegen(d, e)

$e_1 \boxed{?} e_2$	codegen(t_1 , e_1)
	codegen(t_2 , e_2)
	$d \boxed{?} t_1 \boxed{?} t_2$

Static Single Assignment

- Must keep track of what definition each use refers to in order to properly do constant/copy propogation.
- Much simpler if only one definition for each name.
- SSA: Each name is **assigned** in only one location.
- Easy for fresh temporaries
- What about variables?

SSA for Straight-line code

- Give each variable a version number.
- Scan code in program order
- Whenever we encounter a definition, increment the version number
- Whenever we encounter a use, use the most recently assigned version number.

$x \; \boxed{?} \; 5$	$x_0 \; \boxed{?} \; 5$
$y \; \boxed{?} \; x - 4$	$y \; \boxed{?} \; x - 4$
$x \; \boxed{?} \; y + 7$	$x \; \boxed{?} \; y + 7$
$z \; \boxed{?} \; x$	$z \; \boxed{?} \; x$

SSA for Straight-line code

- Give each variable a version number.
- Scan code in program order
- Whenever we encounter a definition, increment the version number
- Whenever we encounter a use, use the most recently assigned version number.

$x \boxed{?} 5$	$x_0 \boxed{?} 5$
$y \boxed{?} x - 4$	$y \boxed{?} x_0 - 4$
$x \boxed{?} y + 7$	$x \boxed{?} y + 7$
$z \boxed{?} x$	$z \boxed{?} x$

SSA for Straight-line code

- Give each variable a version number.
- Scan code in program order
- Whenever we encounter a definition, increment the version number
- Whenever we encounter a use, use the most recently assigned version number.

$x \; \boxed{?} \; 5$	$x_0 \; \boxed{?} \; 5$
$y \; \boxed{?} \; x - 4$	$y_0 \; \boxed{?} \; x_0 - 4$
$x \; \boxed{?} \; y + 7$	$x \; \boxed{?} \; y + 7$
$z \; \boxed{?} \; x$	$z \; \boxed{?} \; x$

SSA for Straight-line code

- Give each variable a version number.
- Scan code in program order
- Whenever we encounter a definition, increment the version number
- Whenever we encounter a use, use the most recently assigned version number.

$x \boxed{?} 5$	$x_0 \boxed{?} 5$
$y \boxed{?} x - 4$	$y_0 \boxed{?} x_0 - 4$
$x \boxed{?} y + 7$	$x_1 \boxed{?} y_0 + 7$
$z \boxed{?} x$	$z \boxed{?} x$

$x_0 \leftarrow 5$
$y_0 \leftarrow x_0 / \emptyset$

SSA for Straight-line code

- Give each variable a version number.
- Scan code in program order
- Whenever we encounter a definition, increment the version number
- Whenever we encounter a use, use the most recently assigned version number.

$\emptyset$
y / x

$x / \emptyset$

while (cond) {
}

x ⍰ 5	x_0 ⍰ 5
y ⍰ x - 4	y_0 ⍰ ~~x_0 - 4~~ 1
x ⍰ y + 7	x_1 ⍰ ~~y_0 + 7~~ 8
z ⍰ x	z_0 ⍰ x_1

Now easy

- Constant propagation:
  - Can replace all x_0 with 5.

$x_0 \boxed{?} 5$

$y_0 \boxed{?} x_0 - 4$

$x_1 \boxed{?} y_0 + 7$

$z_0 \boxed{?} x_1$

- Copy Propagation:
  - Can replace all x_0 with y_0

$x_0 \boxed{?} y_0$

$y_1 \boxed{?} u_0 - 4$

$z_0 \boxed{?} x_0 + 7$

Today

- Context
- Abstract Assembly
- AST ⍰ IR
- Maximal Munch
- Issues
- Simple SSA
- x86 and 2-adr Instructions

Real Assembly on x86

- x86 doesn't have 3 address instructions!

$$d \boxed{?} s_1 + s_2$$

Real Assembly on x86

- x86 doesn't have 3 address instructions!

$$d \ \boxed{?} \ s_1 + s_2$$

Triples	2-adr	x86
$d \boxed{?} s_1 + s_2$	$d \boxed{?} s_1$ $d \boxed{?} d + s_2$	MOVx s_1, d ADDx s_2, d

$t_1 \leftarrow S_1$
$t_2 \leftarrow S_2$
$t_1 \leftarrow t_1 + t_2$

Real Assembly on x86

- x86 doesn't have 3 address instructions!

Triples	2-adr	x86
$d \boxed{?} s_1 + s_2$	$d \boxed{?} s_1$ $d \boxed{?} d + s_2$	MOVx s_1, d ADDx s_2, d

- All kinds of special register requirements

$$d \boxed{?} s_1 * s_2$$

What about edx?

Triples	2-adr	x86
$d \boxed{?} s_1 * s_2$	$d \boxed{?} s_1$ $d \boxed{?} d * s_2$	MOVL s_1, rax IMUL s_2 MOVL rax, d

From AST to Machine Assembly

- Implied Approach:
  - AST ? Triples using unlimited temporaries
  - Map temporaries to registers/memory
  - Lower Triples to real assembly
- What about Interaction between registers and instructions?
- Cost model?
- KISS:
  - Keep things simple, but
  - Prepare for other passes to fix things up.