Instruction Selection

15-411/15-611 Compiler Design

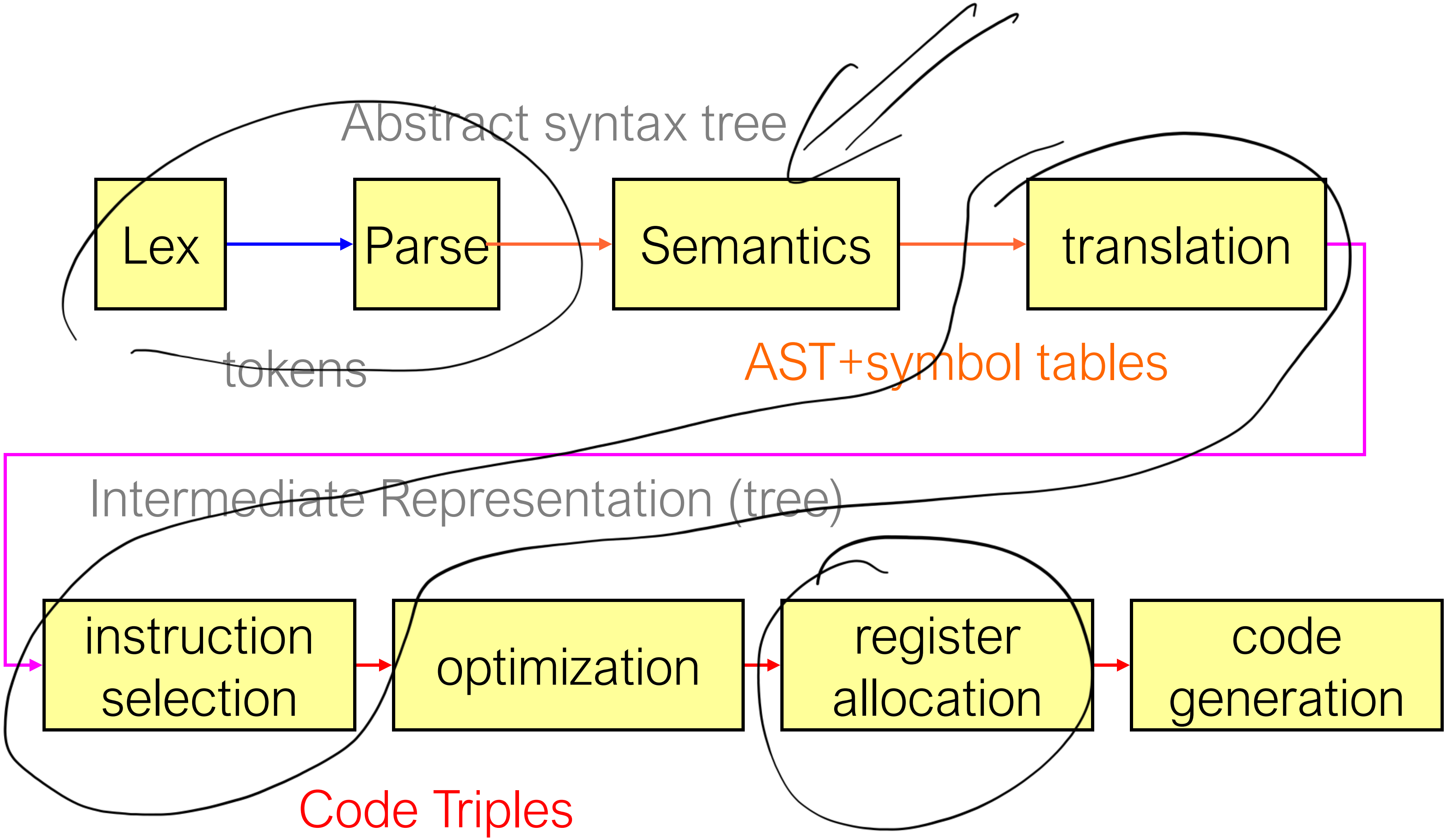
Seth Copen Goldstein

January 22, 2026

Today

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

Cartoon Compiler



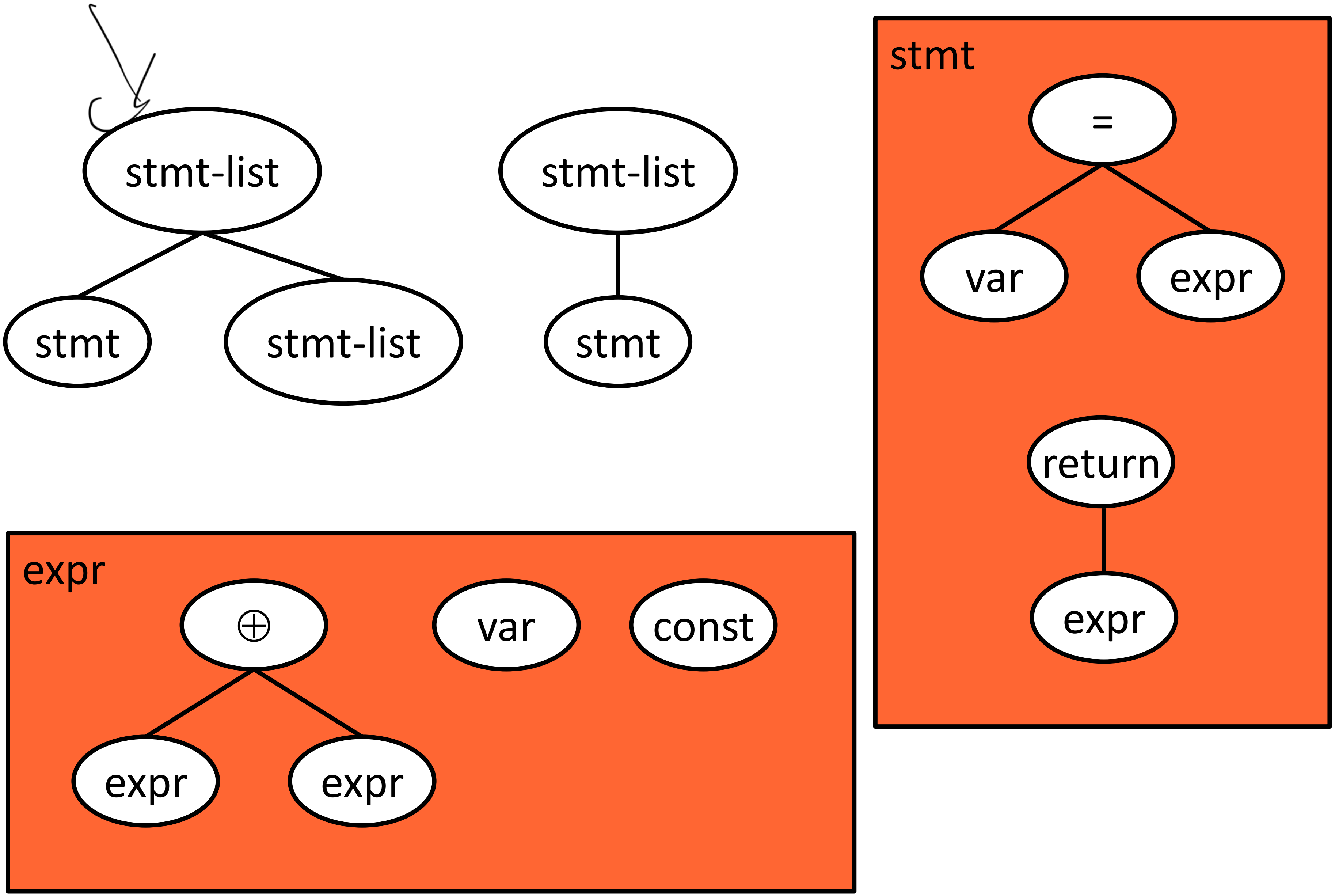
Simple Source Language

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

Simple Source Language

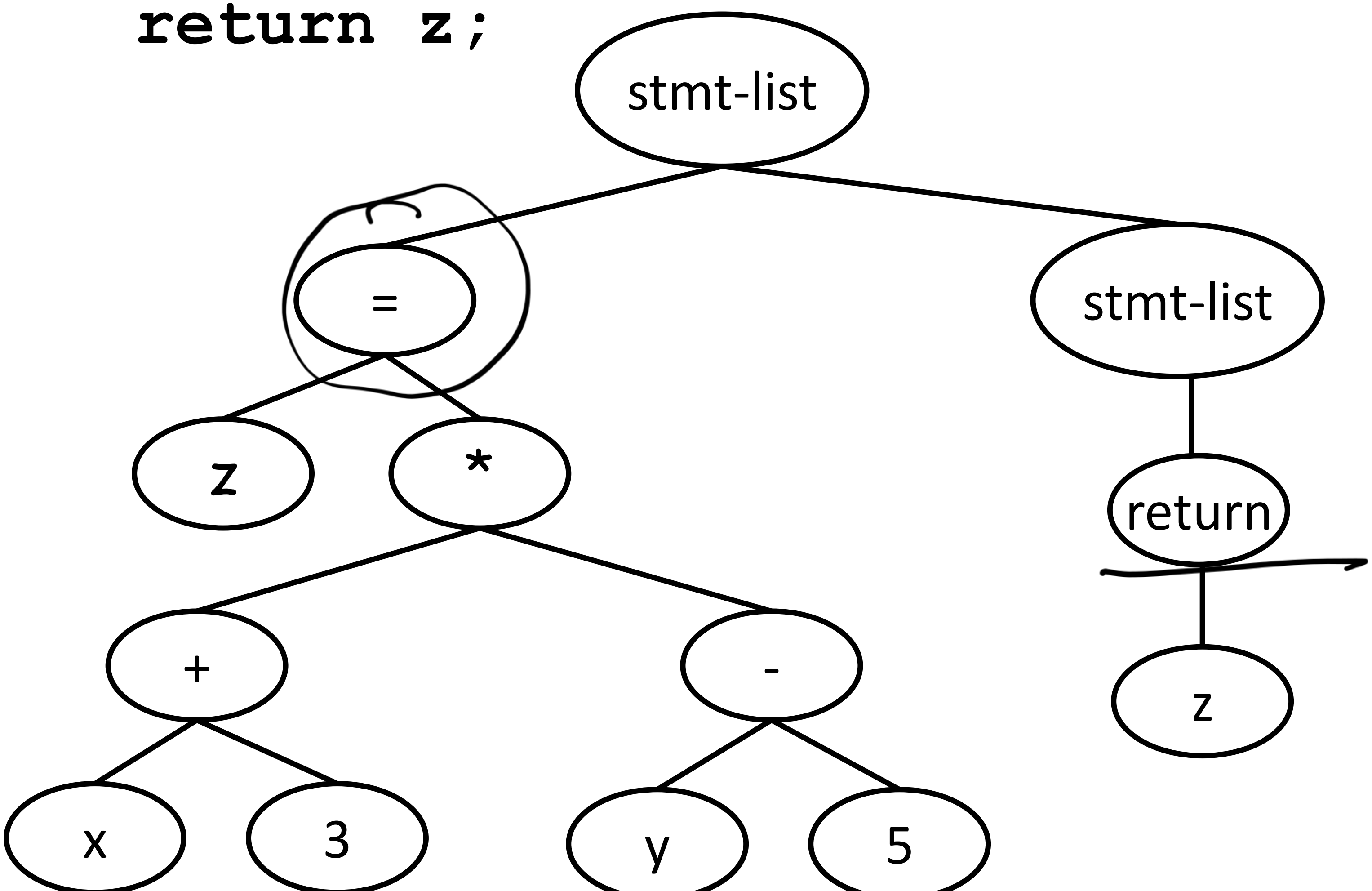
program	:=	s_1 **;** s_2 **;** … s_n **;**	sequence of statements
s	:=	v **=** e	assignment
	|	**return** e	return
e	:=	c	constant
	|	v	variable
	|	$e_1 \oplus e_2$	binary operation
$\oplus$	:=	**+** | **-** | ***** | **/** | **%**	

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 $\leftarrow$ src_1 operator src_2

dest $\leftarrow$ operator src_1

operator

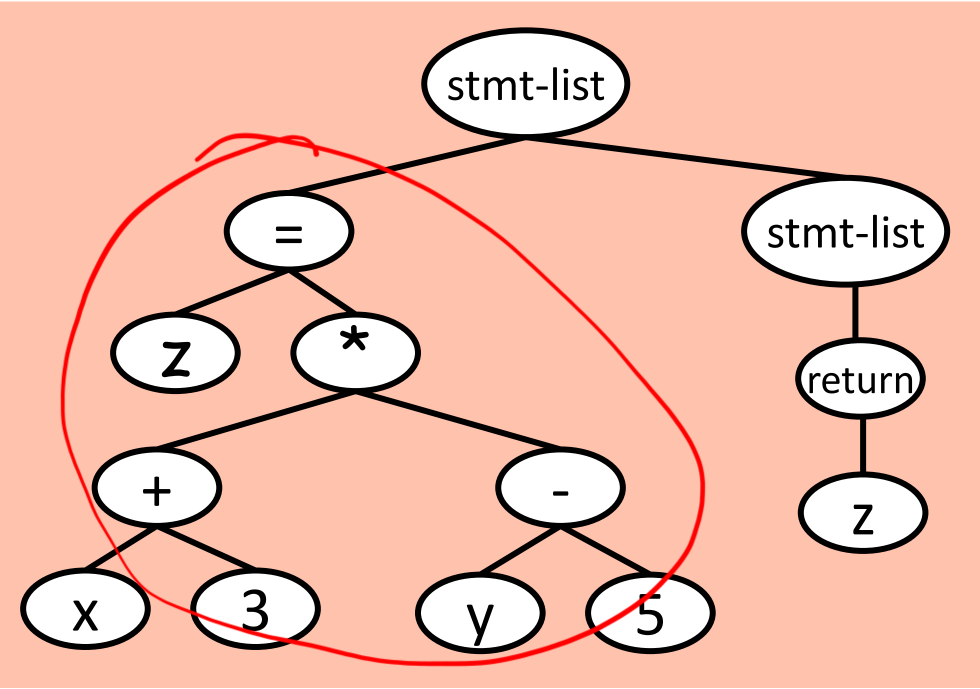
src can be:
- constant
- temp
- special register
- memory

Abstract Assembly

program	:=	$i_1\ i_2\ \ldots\ i_n$	seq of instructions
i	:=	$d \leftarrow s$	move
	|	$d \leftarrow s_1 \oplus s_2$	binop
	|	~~**return**~~ S	return what is in **`rax`** The ABI Reg
s	:=	c	intermediate
	|	t	temporary
	|	r	register
d	:=	t	
	|	r	
$\oplus$	:=	**+ | - | * | / | %**	

Example Goal

```
z = x + 3 * y - 5;
return z;
```

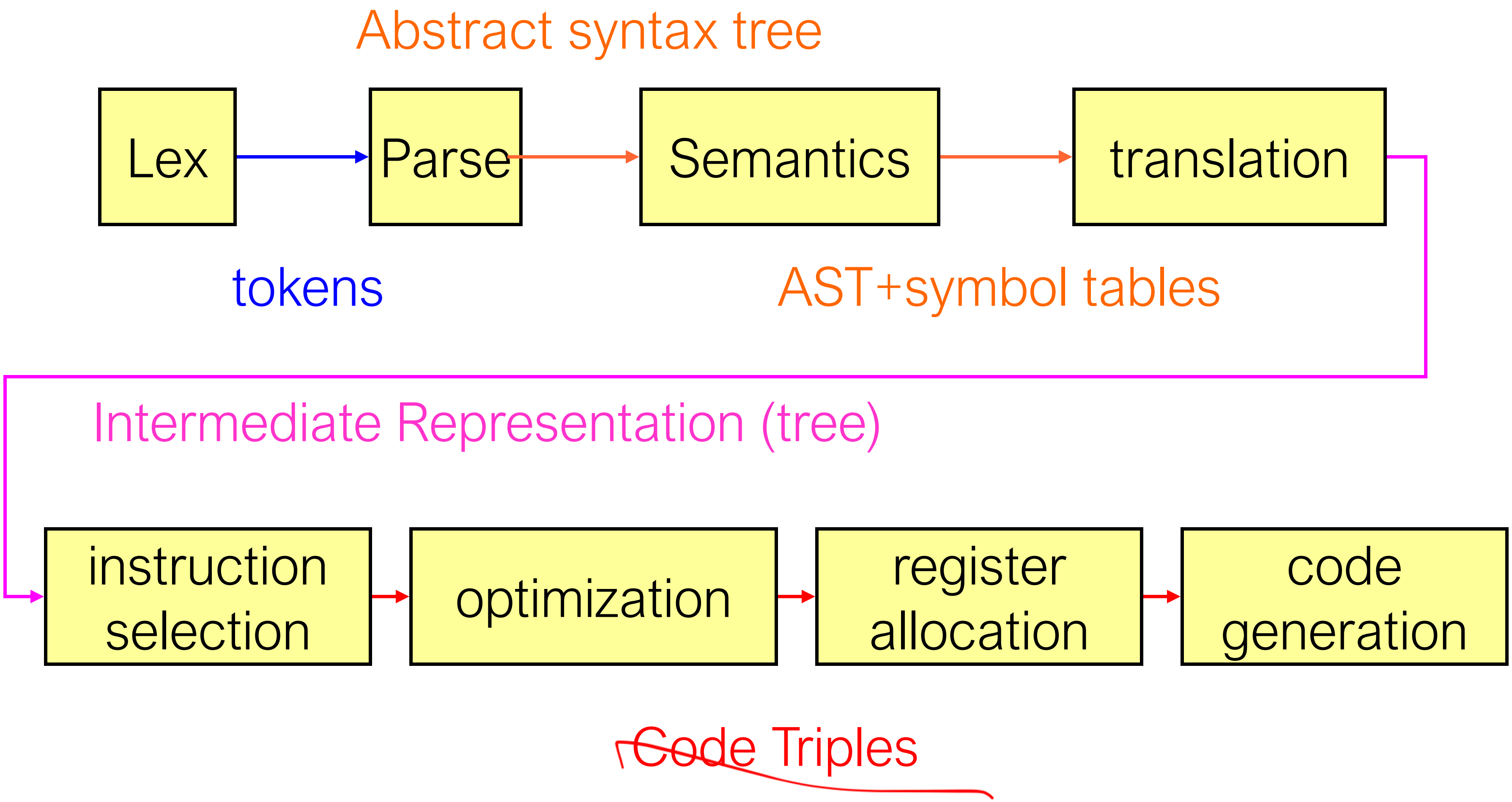



```
t1   ←  x + 3
t2   ←  y - 5
z    ←  t1 * t2
rax  ←  z
        return
```

Today

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

Cartoon Compiler

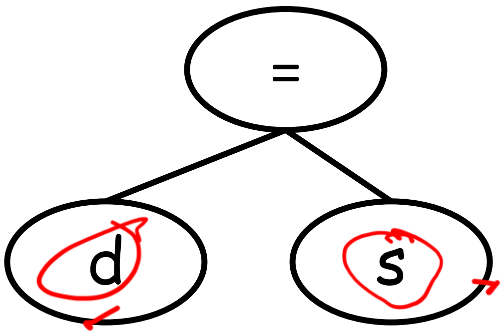


Alternatives abound

Translating AST to IR

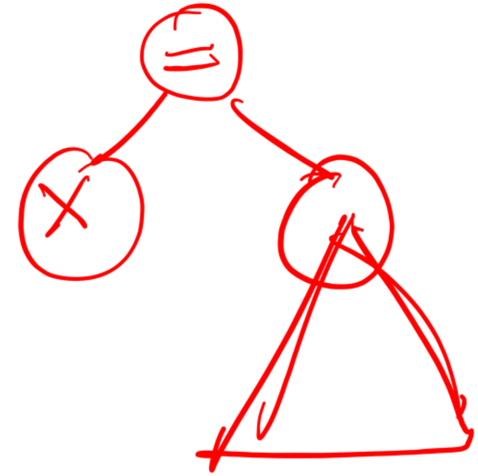
- Converting from tree structured IR to sequence of instructions
  - Create temporary locations to store values
  - choose which operations we want
    - can combine or
    - breakup original operations
- Match portions of tree and convert to triple

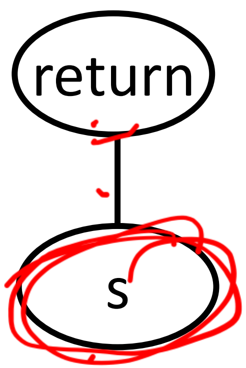
Tree Patterns (aka Tiles)



$d \leftarrow s$

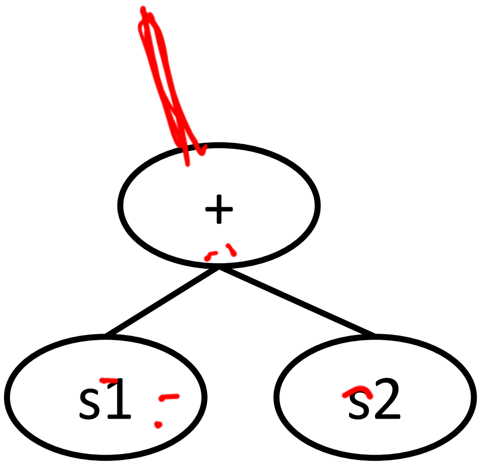
mov s, d





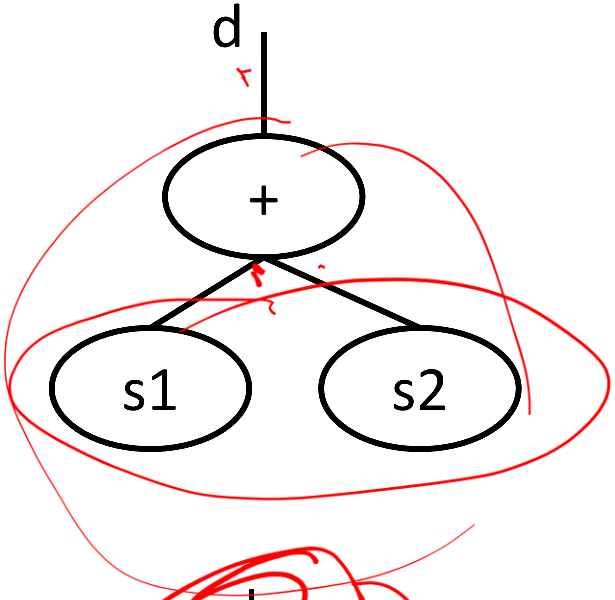
$rax \leftarrow s$
ret

Tree Patterns

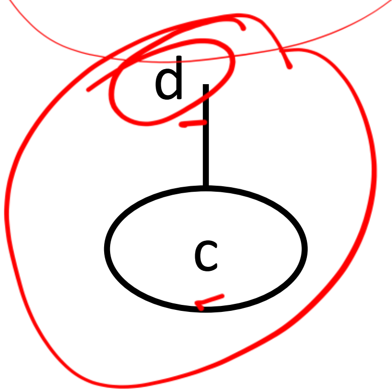


S1 + S2

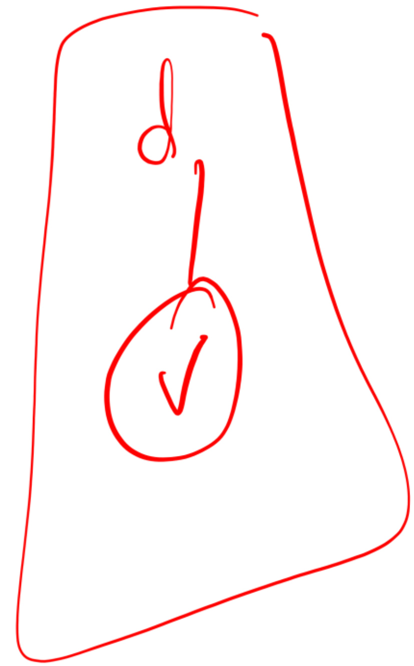
Tree Patterns



$d \leftarrow s_1 + s_2$



$d \leftarrow c$



Tiling a Tree

$t_2 \leftarrow x + 3$

$t_1 \leftarrow 3$

$t_2 \leftarrow x + t_1$

$t_3 \leftarrow 5$

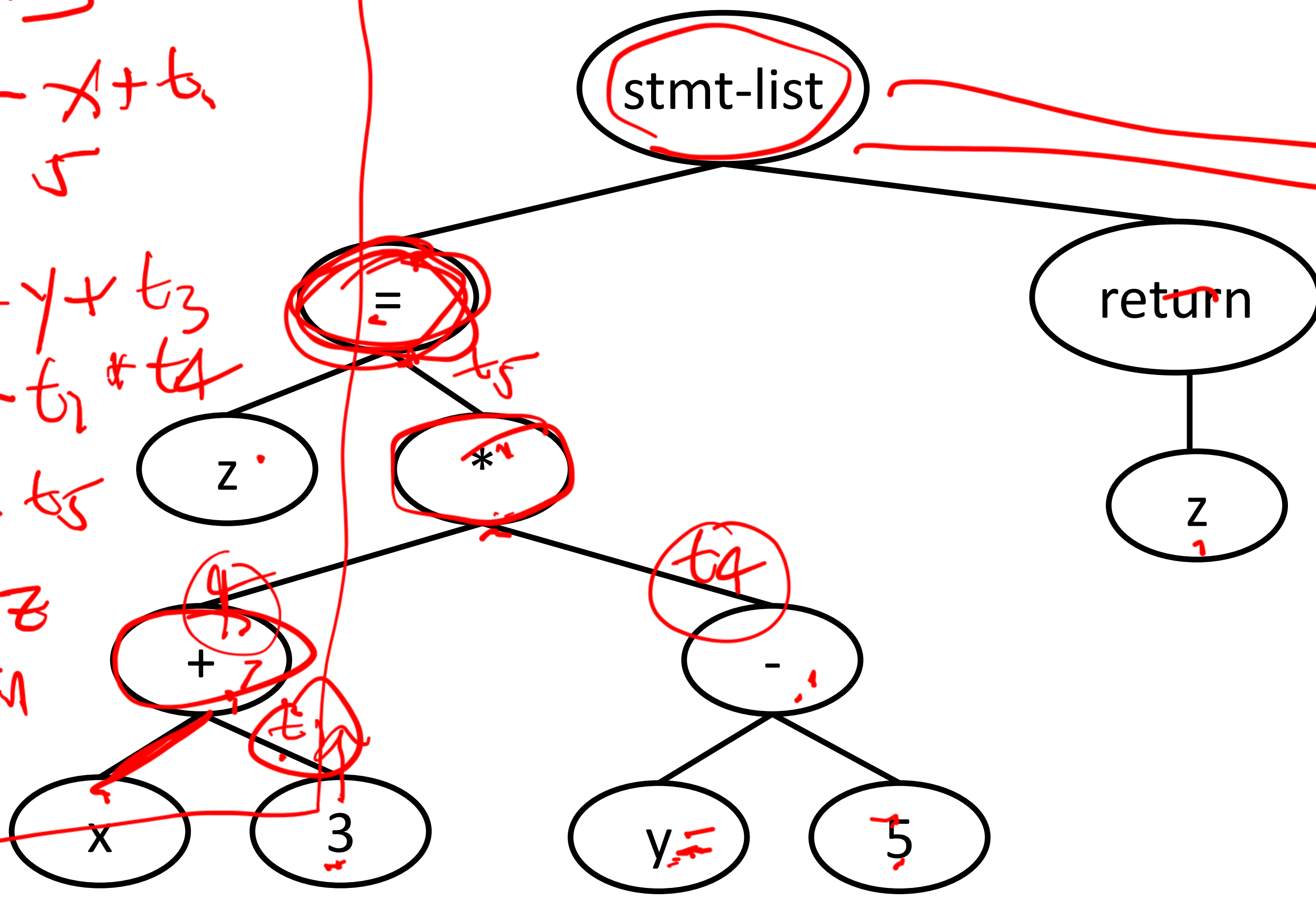
$t_4 \leftarrow y + t_3$

$t_5 \leftarrow t_2 * t_4$

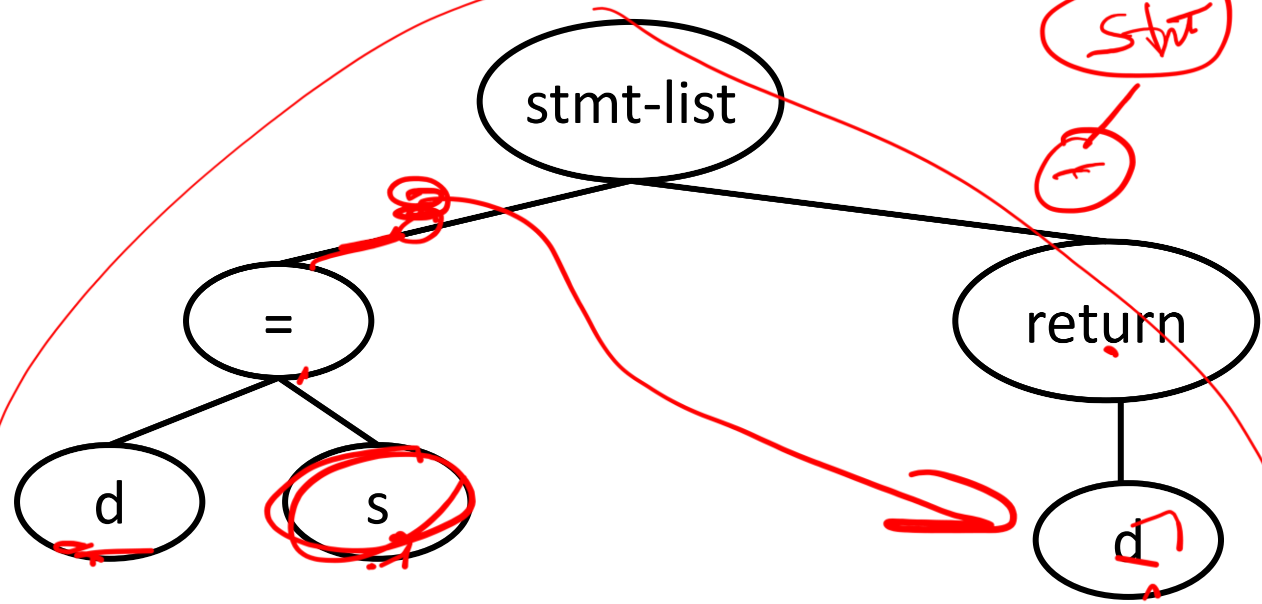
$z \leftarrow t_5$

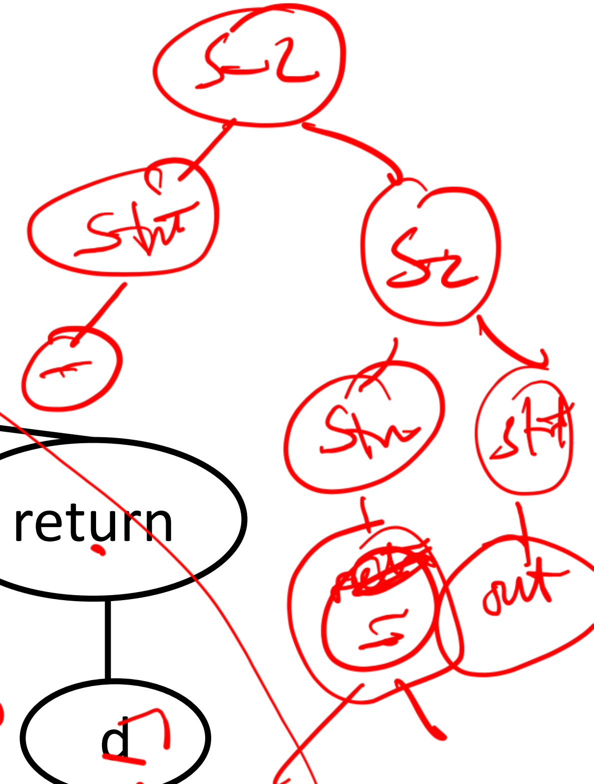
$rax \leftarrow z$

return



Better Tiles





```
rax ← s
        return
```

Correct?
If correct: better or worse?

Today

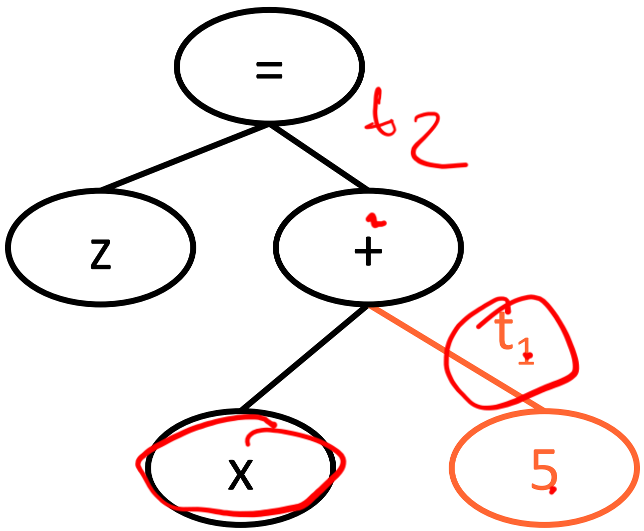
- Context
- Abstract Assembly
- AST → IR
- Maximal Munch
- Issues
- Simple SSA
- x86 and 2-adr Instructions

Maximal Munch

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

Maximal Munch

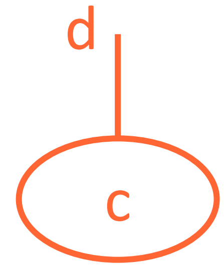
- recursively match tree
- At each step, pick "best" tile



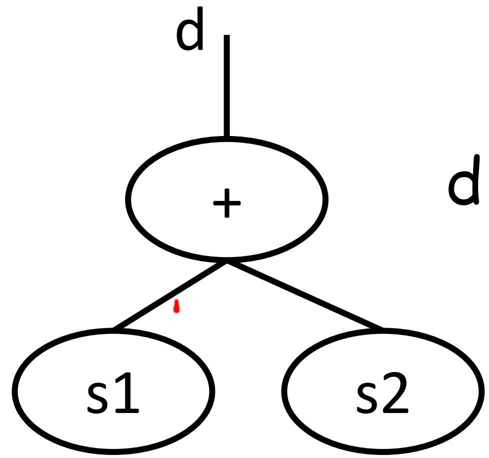
$t_1 \leftarrow 5$

$t_2 \leftarrow x + t_1$

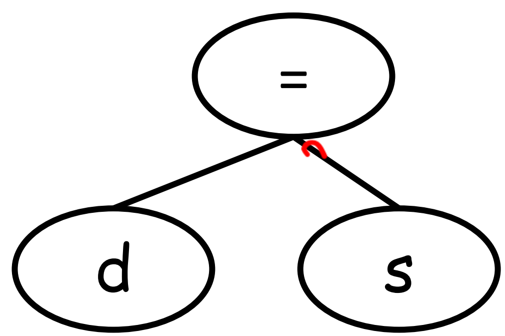
$z \leftarrow t_2$



$d \leftarrow c$



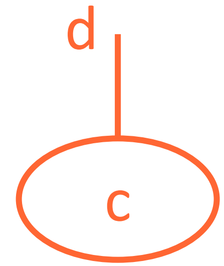
$d \leftarrow s_1 + s_2$



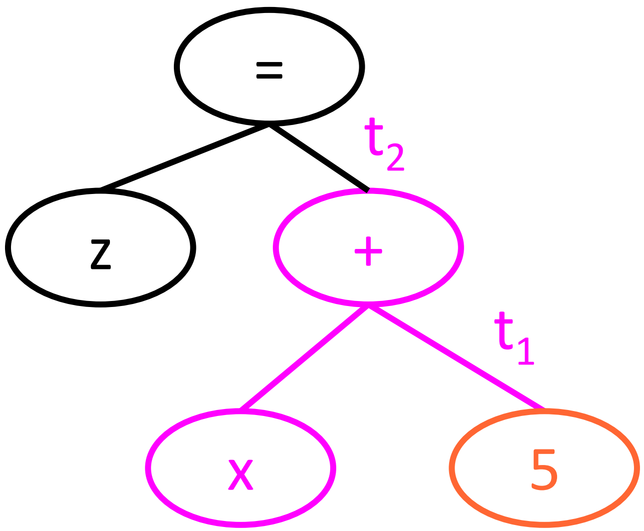
$d \leftarrow s$

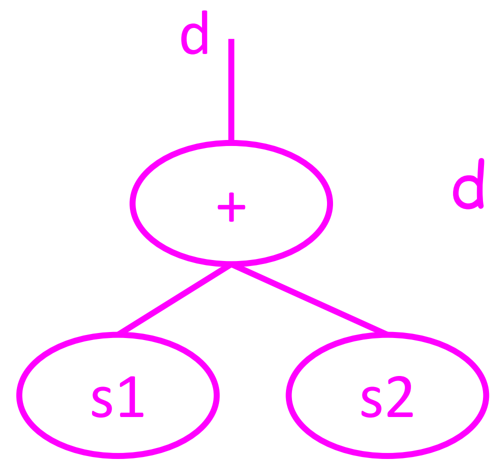
Maximal Munch

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



$d \leftarrow c$

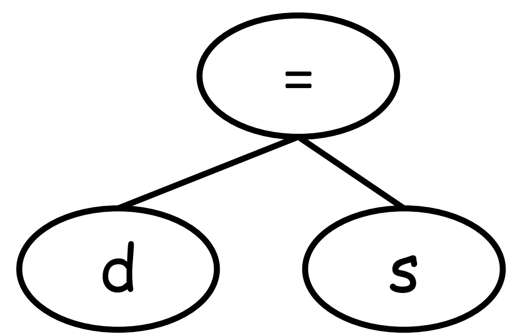




$d \leftarrow s_1 + s_2$

$t_1 \leftarrow 5$

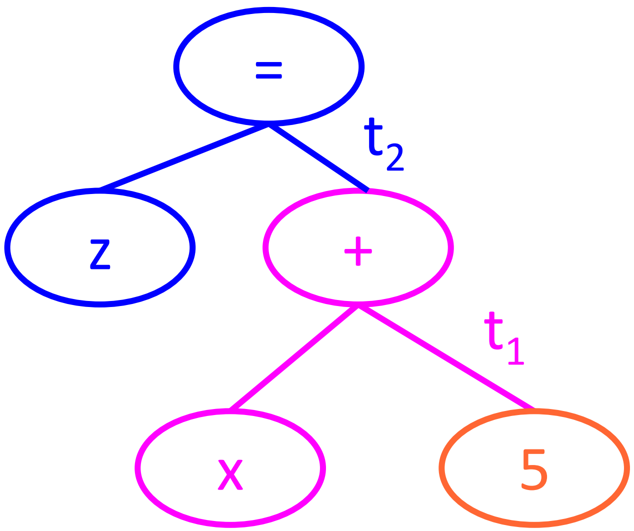
$t_2 \leftarrow x + t_1$

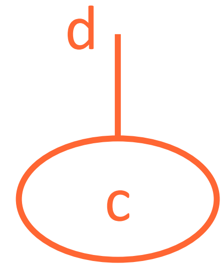


$d \leftarrow s$

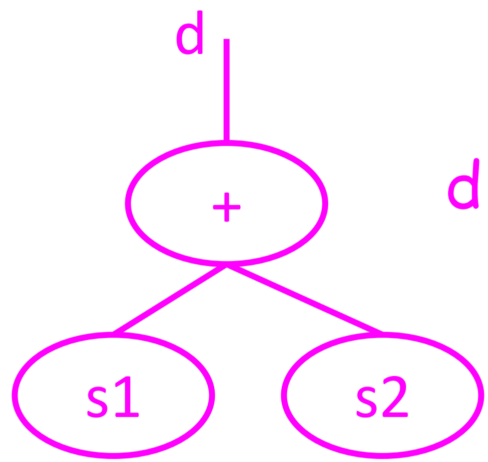
Maximal Munch

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

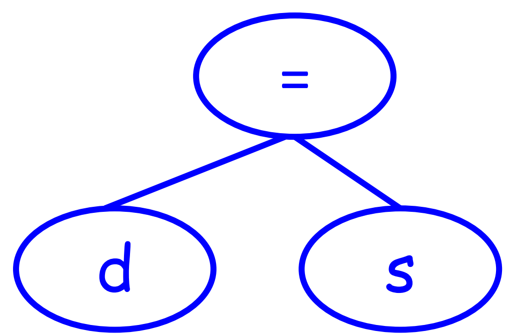




$d \leftarrow c$



$d \leftarrow s_1 + s_2$



$d \leftarrow s$

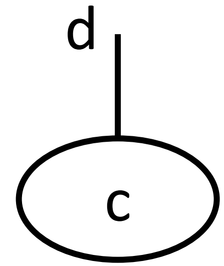
$t_1 \quad \leftarrow 5$

$t_2 \quad \leftarrow x + t_1$

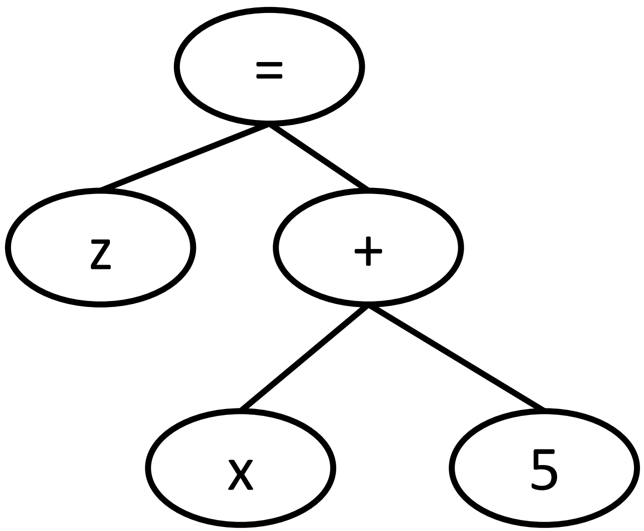
$z \quad \leftarrow t_2$

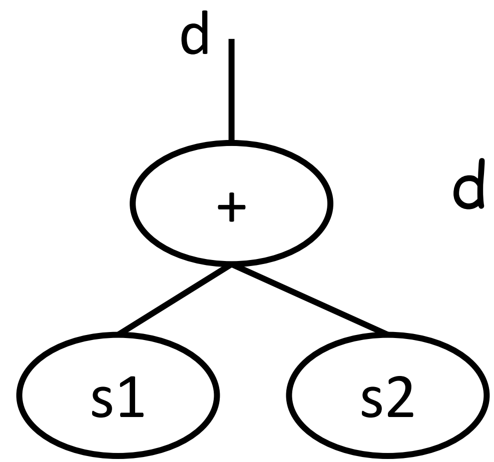
Maximal Munch

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

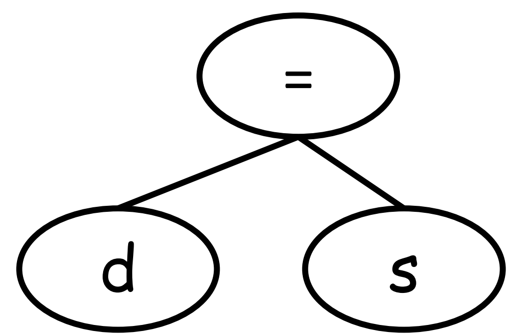


$d \leftarrow c$





$d \leftarrow s_1 + s_2$



$d \leftarrow s$

$t_1 \leftarrow 5$

$t_2 \leftarrow x + t_1$

$z \leftarrow t_2$

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

Codegen(t, e,)

e	codegen(d, e)
c	d ← c
v	d ← z
$e_1 \oplus e_2$	Codegen(t_1, e_1) fresh t_1 & t_2 Codegen(t_2, e_2) d ← t_1 ⊕ t_2

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

z

d

5

f

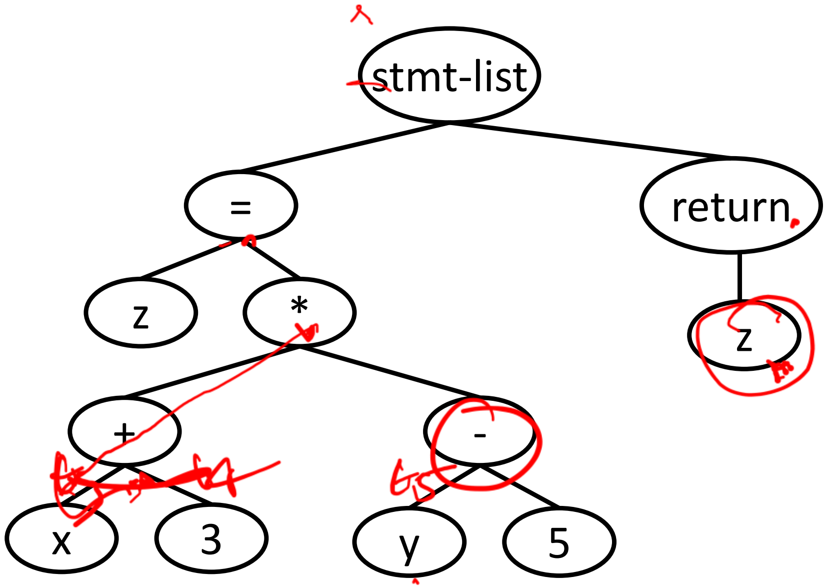
e_1 e_2

codegen

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

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(t, ⊕)

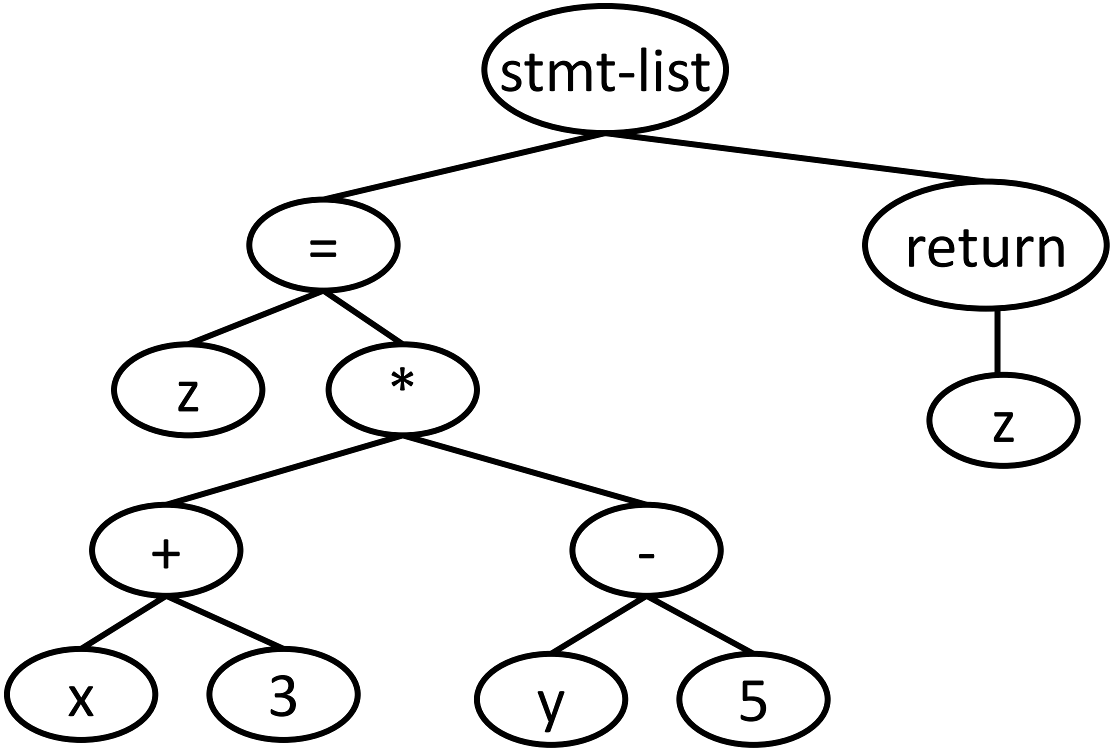
cg(t1, ⊕) ; cg(t2, ⊕) ; d ...

cg(t3, x) ; cg(t4, 3) ; d ...

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

$z \leftarrow t_1 * t_2$
rax ← z
return

Result



$t_3 \leftarrow x$

$t_4 \leftarrow 3$

$t_1 \leftarrow t_3 + t_4$

$t_5 \leftarrow y$

$t_6 \leftarrow 5$

$t_2 \leftarrow t_5 * t_6$

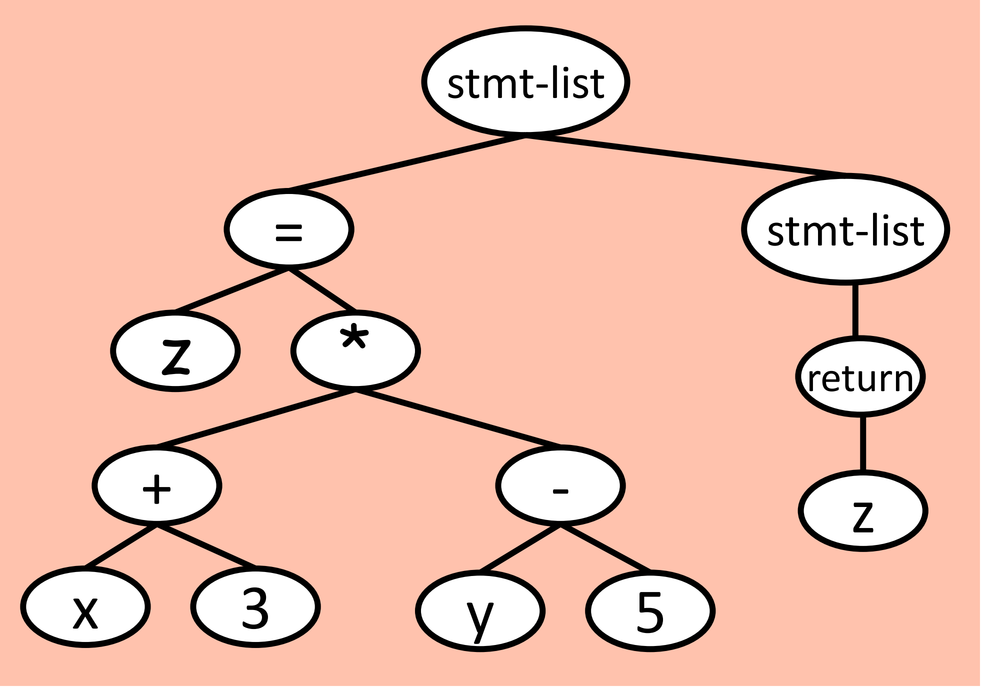
$z \leftarrow t_1 * t_2$

$rax \leftarrow z$

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

$t_3 \leftarrow x$

$t_4 \leftarrow 3$

$t_1 \leftarrow t_3 + t_4$

$t_5 \leftarrow y$

$t_6 \leftarrow 5$

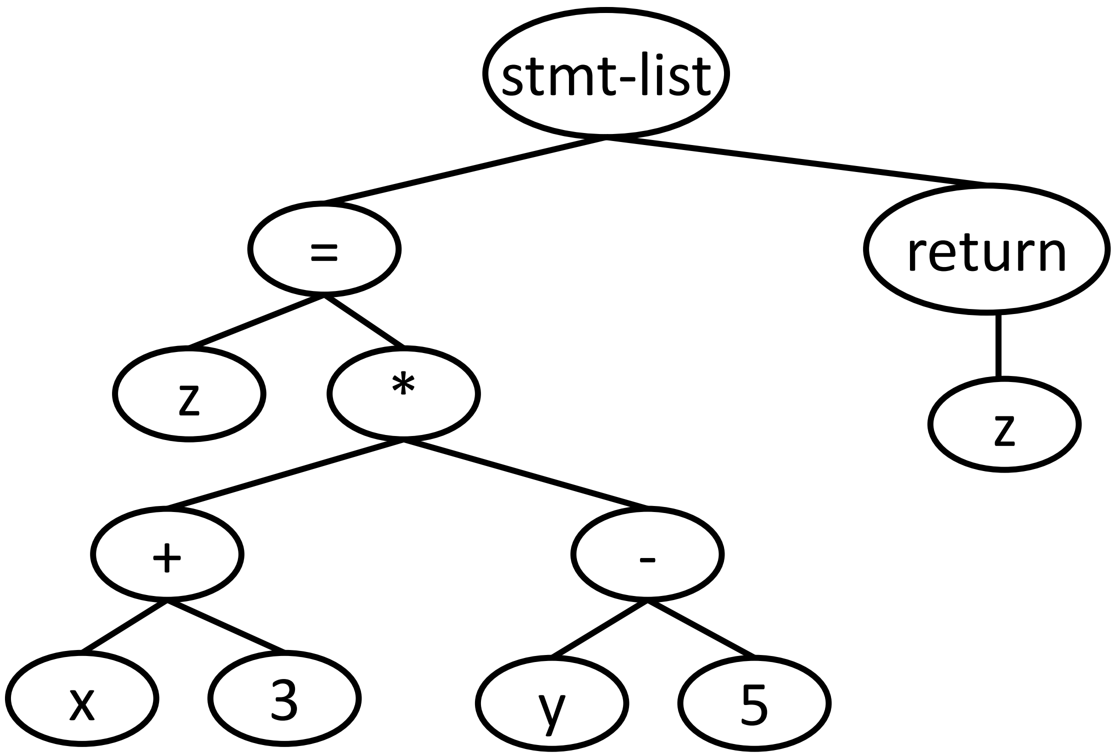
$t_2 \leftarrow t_5 * t_6$

$z \leftarrow t_1 * t_2$

$rax \leftarrow z$

ret

How Can we Improve this?



$t_3 \leftarrow x$

$t_4 \leftarrow 3$

$t_1 \leftarrow t_3 + t_4$

$t_5 \leftarrow y$

$t_6 \leftarrow 5$

$t_2 \leftarrow t_5 * t_6$

$z \leftarrow t_1 * t_2$

$rax \leftarrow z$

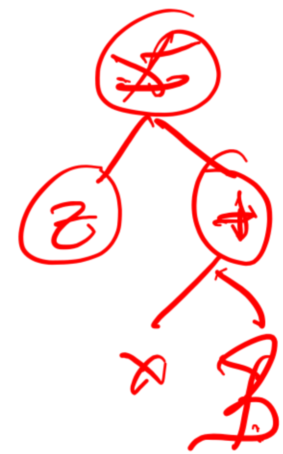
ret

How Can we Improve this?

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

Generating Destinations

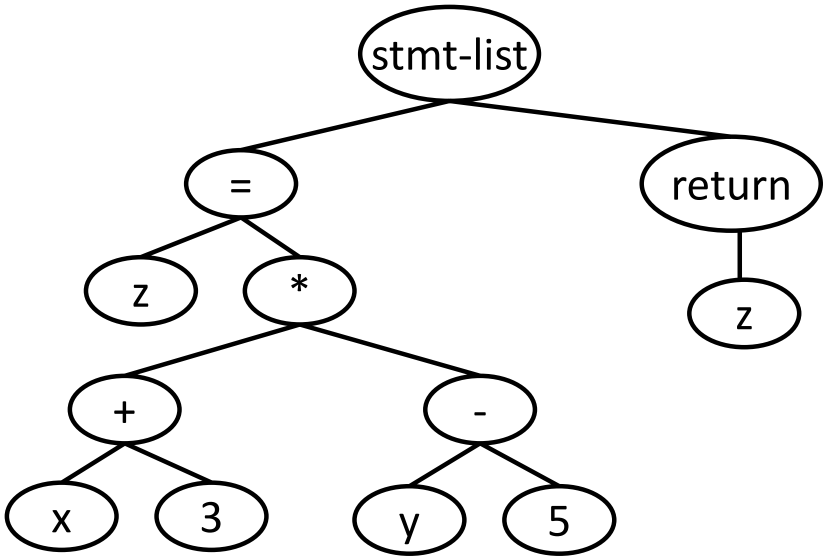
e	codegen(e)	up
c		c
v		v
$e_1 \oplus e_2$	$t_1 = \overset{x}{\text{codegen}(e_1)} \oplus$ $\underset{3}{\text{codegen}(e_2)}$	t_1



$t_1 = x + 3$

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

Special Cases

e	codegen(d, e)
c	$d \leftarrow c$
v	$d \leftarrow x$
$c \oplus e_2$	$\text{codegen}(t_2, e_2)$ $d \leftarrow c \oplus t_2$
$e_1 \oplus c$	$\text{codegen}(t_1, e_1)$ $d \leftarrow t_1 \oplus c$
$v \oplus e_2$	$\text{codegen}(t_2, e_2)$ $d \leftarrow v \oplus t_2$
$e_1 \oplus v$	$\text{codegen}(t_1, e_1)$ $d \leftarrow t_1 \oplus v$
$e_1 \oplus e_2$	$\text{codegen}(t_1, e_1)$ $\text{codegen}(t_2, e_2)$ $d \leftarrow t$

Generally not recommended

The "don't bother" case

- What should we really do?

$t_3 \leftarrow x$

$t_4 \leftarrow 3$

$t_1 \leftarrow t_3 + t_4$

$t_5 \leftarrow y$

$t_6 \leftarrow 5$

$t_2 \leftarrow t_5 * t_6$

$z \leftarrow t_1 * t_2$

$rax \leftarrow z$

ret

Constant Propagation

Copy Propagation

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

Constant Propogation

$t_3 \leftarrow x$

$\cancel{t_4 \leftarrow 3}$

$t_1 \leftarrow t_3 + \cancel{t_4}\ 3$

$t_5 \leftarrow y$

$\cancel{t_6 \leftarrow 5}$

$t_2 \leftarrow t_5 * \cancel{t_6}\ 5$

$z \leftarrow t_1 * t_2$

$rax \leftarrow z$

ret

Copy Propogation

~~$t_3 \leftarrow x$~~

$t_1 \leftarrow \cancel{t_3}\ x + 3$

~~$t_5 \leftarrow y$~~

$t_2 \leftarrow \cancel{t_5}\ y * 5$

$z \leftarrow t_1 * t_2$

$rax \leftarrow z$

ret

Have to be careful

- Constant propagation:

$$
\begin{aligned}
x &\leftarrow 5\\
y &\leftarrow \cancel{x} - 4\\
x &\leftarrow \cancel{y} + 7 \quad 8\\
z &\leftarrow \cancel{x}\ 8
\end{aligned}
$$

- Copy Propagation:

$$
\begin{aligned}
x &\leftarrow y\\
y &\leftarrow u - 4\\
z &\leftarrow x + 7
\end{aligned}
$$

Have to be careful

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

$x \leftarrow 5$

$y \leftarrow x - 4$

$x \leftarrow y + 7$

$z \leftarrow x$

- Copy Propagation:

$x \leftarrow y$

$y \leftarrow u - 4$

$z \leftarrow x + 7$

Have to be careful

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

$$
\begin{aligned}
x &\leftarrow 5 \\
y &\leftarrow x - 4 \\
x &\leftarrow y + 7 \\
z &\leftarrow x
\end{aligned}
$$

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

$$
\begin{aligned}
x &\leftarrow y \\
y &\leftarrow u - 4 \\
z &\leftarrow x + 7
\end{aligned}
$$

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)
$e_1 \oplus e_2$	codegen(t_1 , e_1) codegen(t_2 , e_2) $d \leftarrow t_1 \oplus t_2$

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	$\leftarrow 5$	x_0	$\leftarrow 5$
y	$\leftarrow x - 4$	y	$\leftarrow x - 4$
x	$\leftarrow y + 7$	x	$\leftarrow y + 7$
z	$\leftarrow x$	z	$\leftarrow x$

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.

$x \leftarrow 5$	$x_0 \leftarrow 5$
$y \leftarrow x - 4$	$y \leftarrow x_0 - 4$
$x \leftarrow y + 7$	$x \leftarrow y + 7$
$z \leftarrow x$	$z \leftarrow x$

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.

$x \leftarrow 5$	$x_0 \leftarrow 5$
$y \leftarrow x - 4$	$y_0 \leftarrow x_0 - 4$
$x \leftarrow y + 7$	$x \leftarrow y + 7$
$z \leftarrow x$	$z \leftarrow x$

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	$\leftarrow 5$	x_0	$\leftarrow 5$
y	$\leftarrow x - 4$	y_0	$\leftarrow x_0 - 4$
x	$\leftarrow y + 7$	x_1	$\leftarrow y_0 + 7$
z	$\leftarrow x$	z	$\leftarrow x$

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.

$$
\begin{array}{ll@{\qquad\qquad}ll}
x & \leftarrow 5 & x_0 & \leftarrow 5 \\
y & \leftarrow x - 4 & y_0 & \leftarrow x_0 - 4 \\
x & \leftarrow y + 7 & x_1 & \leftarrow y_0 + 7 \\
z & \leftarrow x & z_0 & \leftarrow x_1
\end{array}
$$

Now easy

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

$$
\begin{aligned}
x_0 &\leftarrow 5 \\
y_0 &\leftarrow x_0 - 4 \\
x_1 &\leftarrow y_0 + 7 \\
z_0 &\leftarrow x_1
\end{aligned}
$$

- Copy Propagation:
  – Can replace all x_0 with y_0

$$
\begin{aligned}
x_0 &\leftarrow y_0 \\
y_1 &\leftarrow u_0 - 4 \\
z_0 &\leftarrow x_0 + 7
\end{aligned}
$$

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!

$$d \quad \leftarrow s_1 + s_2$$

Real Assembly on x86

- x86 doesn't have 3 address instructions!

$$d \quad \leftarrow s_1 + s_2$$

Triples	2-adr	x86
$d \leftarrow s_1 + s_2$	$d \leftarrow s_1$ $d \leftarrow d + s_2$	MOVx s_1, d ADDx s_2, d

Real Assembly on x86

- x86 doesn't have 3 address instructions!

Triples	2-adr	x86
$d \leftarrow s_1 + s_2$	$d \leftarrow s_1$ $d \leftarrow d + s_2$	MOVx s_1, d ADDx s_2, d

- All kinds of special register requirements

$$d \leftarrow s_1 * s_2$$

What about edx?

Triples	2-adr	x86

$d \leftarrow s_1 * s_2$	$d \leftarrow s_1$
	$d \leftarrow d * s_2$
		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.