

Instruction Selection

15-411/15-611 Compiler Design

Seth Copen Goldstein

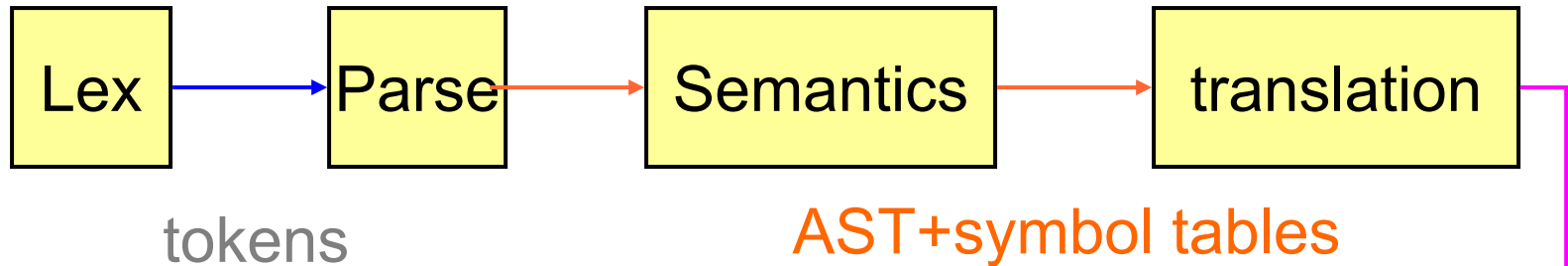
August 21, 2019

Today

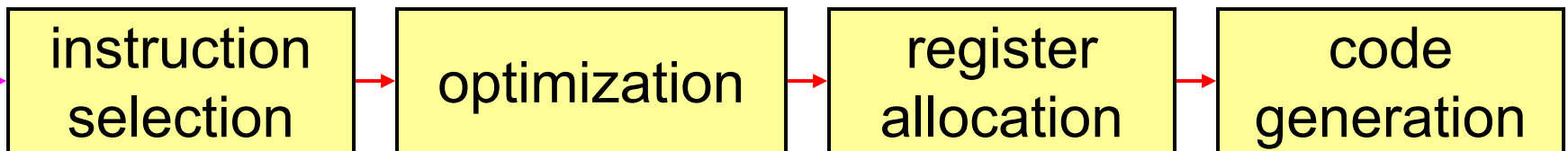
- Context
- Abstract Assembly
- AST $\rightarrow$ IR
- Maximal Munch
- Issues
- Simple SSA
- x86 and 2-adr Instructions

Cartoon Compiler

Abstract syntax tree



Intermediate Representation (tree)



Code Triples

Simple Source Language

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

Simple Source Language

program $:= s_1 ; s_2 ; \dots s_n ;$ sequence of statements

s $:= v = e$ assignment

 | **return** e return

e $:= c$ constant

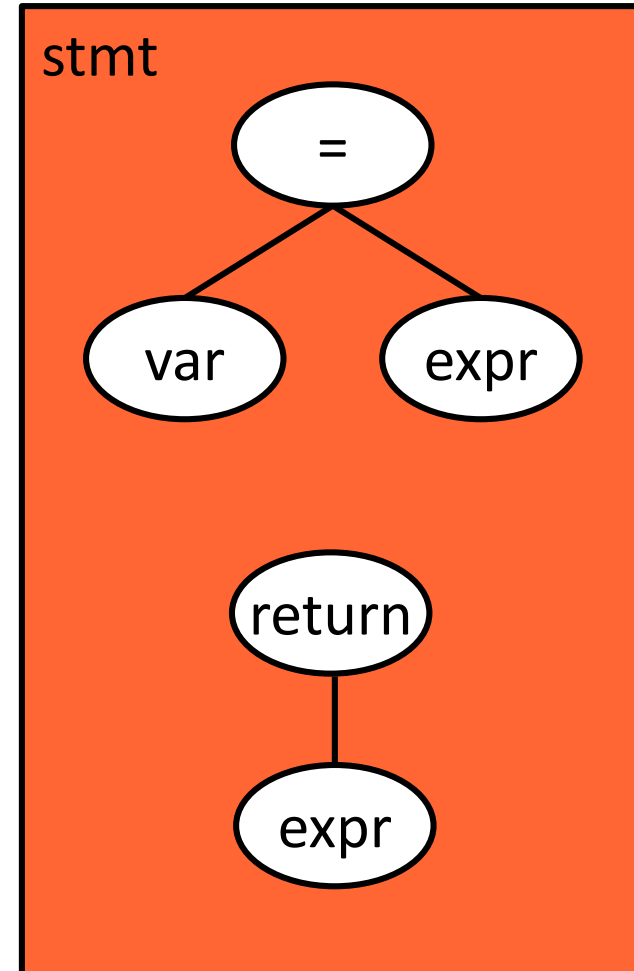
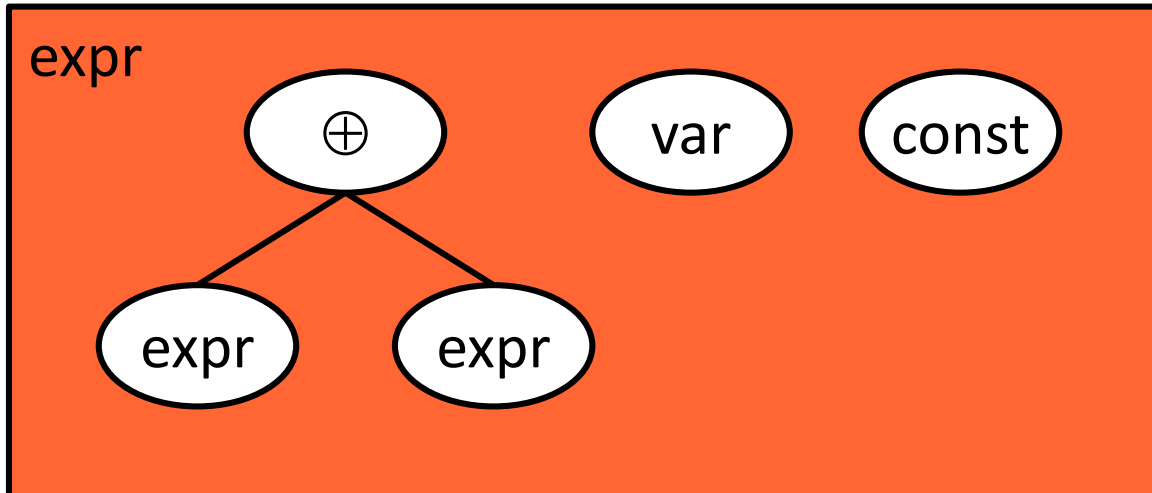
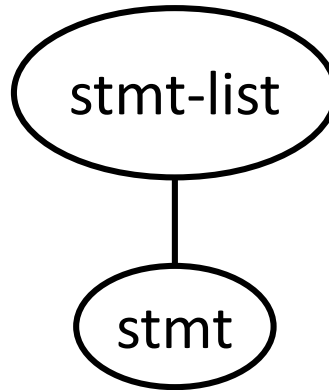
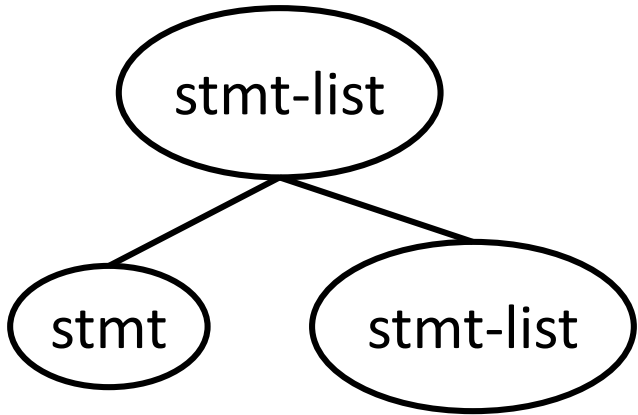
 | v variable

 | $e_1 \oplus e_2$ binary operation

$\oplus$ $:= + \mid - \mid * \mid / \mid \%$

Ambiguity?
Semantics?

Abstract Syntax Tree

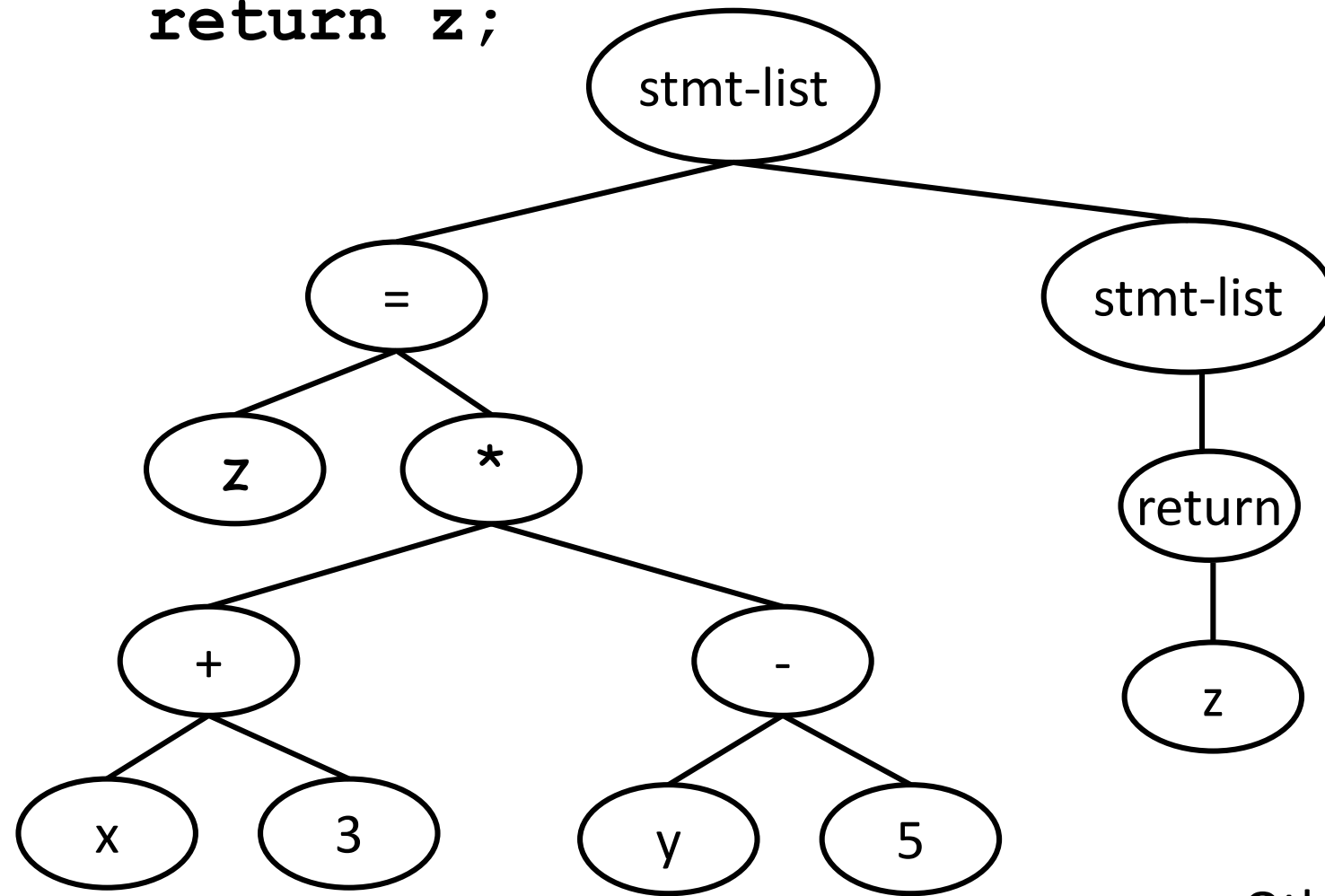


Example

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

Possible parse tree

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



Other possibilities?

Today

- Context
- Abstract Assembly
- $AST \rightarrow IR$
- 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”
- In today's world
aka 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:

$\text{dest} \leftarrow \text{src}_1 \text{ operator } \text{src}_2$

$\text{dest} \leftarrow \text{operator } \text{src}_1$

operator

src can be:

- constant
- temp
- special register
- memory

Abstract Assembly Language

program := $i_1 \ i_2 \ \dots \ i_n$ seq of instructions

```

i      := d ← s           move
      | d ← s1 ⊕ s2      binop
      | return s1         return

```

s	:= c	intermediate	}	values
	t	temporary		
	r	register		
d	:= t		}	locations
	r			

$$\oplus \quad := \quad + \mid - \mid * \mid / \mid \%$$

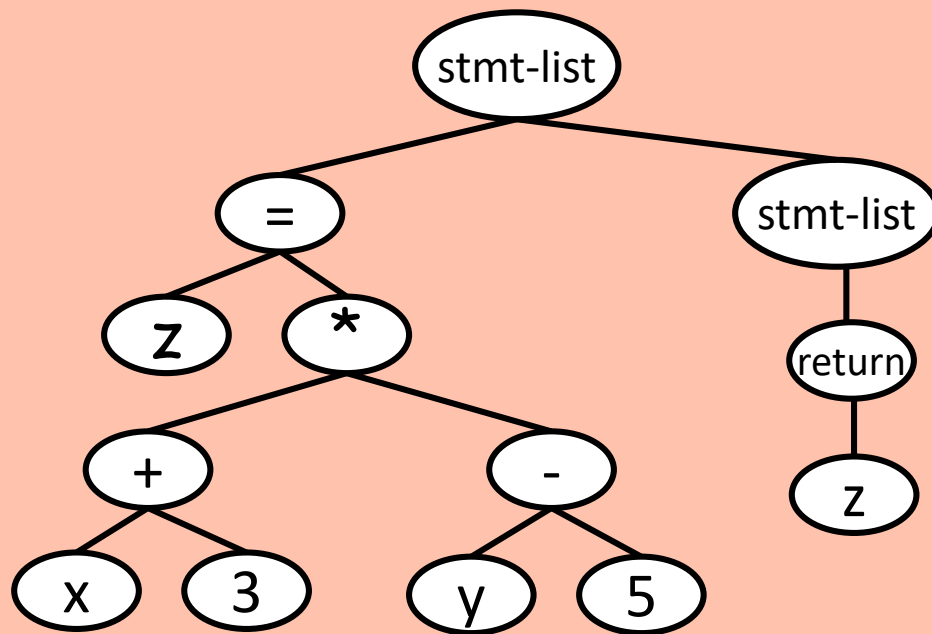
What is right “level”?

Closer to the machine

program	$:= i_1 i_2 \dots i_n$	seq of instructions
i	$:= d \leftarrow s$	move
	$d \leftarrow s_1 \oplus s_2$	binop
	return	return what is in rax
s	$:= c$	intermediate
	t	temporary
	r	register
d	$:= t$	
	r	
$\oplus$	$:= + \mid - \mid * \mid / \mid \%$	

Example Goal

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



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

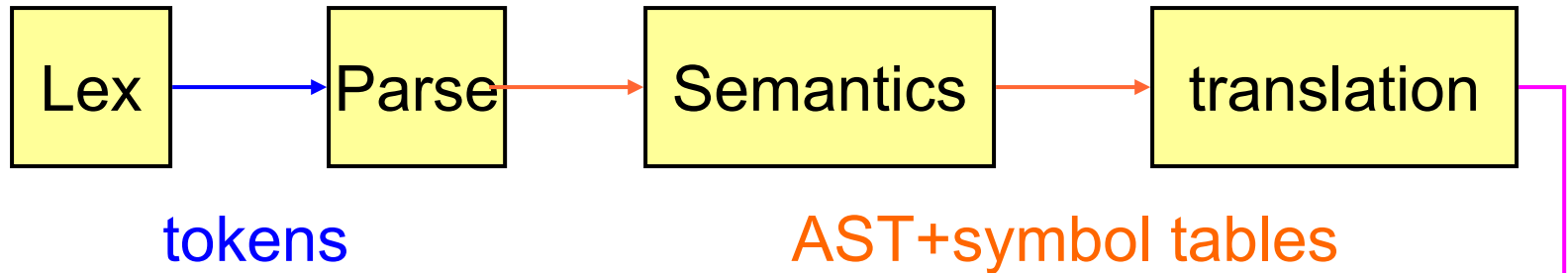
Other possibilities?

Today

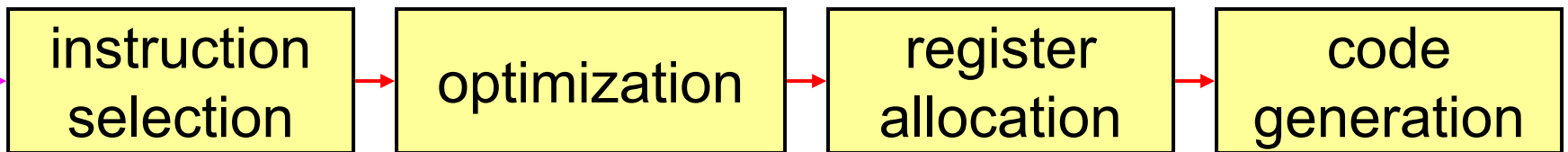
- Context
- Abstract Assembly
- $AST \rightarrow IR$
- Maximal Munch
- Issues
- Simple SSA
- x86 and 2-adr Instructions

Cartoon Compiler

Abstract syntax tree



Intermediate Representation (tree)



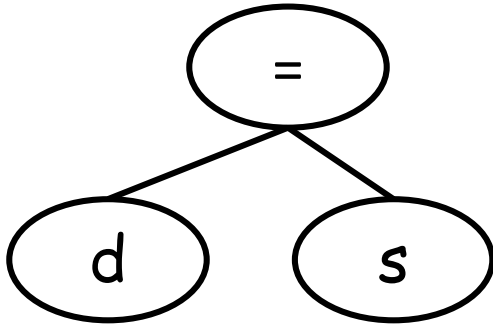
Code Triples

Alternatives abound

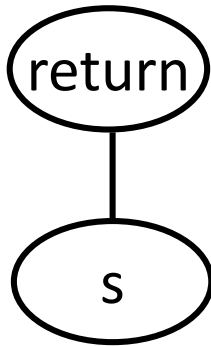
Translating AST to IR

- 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)

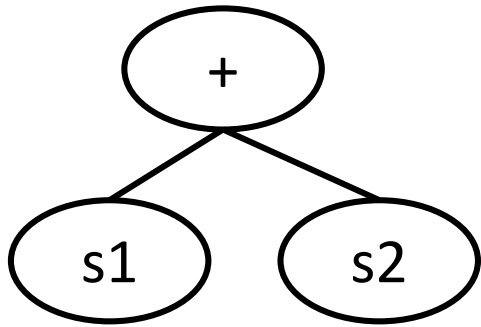


$d \leftarrow s$



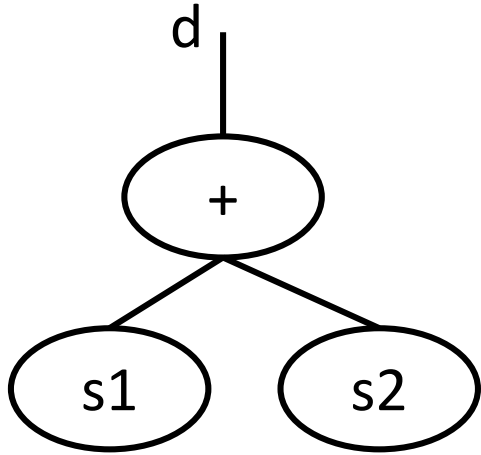
$\text{rax} \leftarrow s$
 ret

Tree Patterns

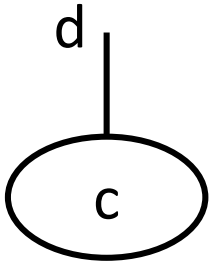


$$? \leftarrow s_1 + s_2$$

Tree Patterns

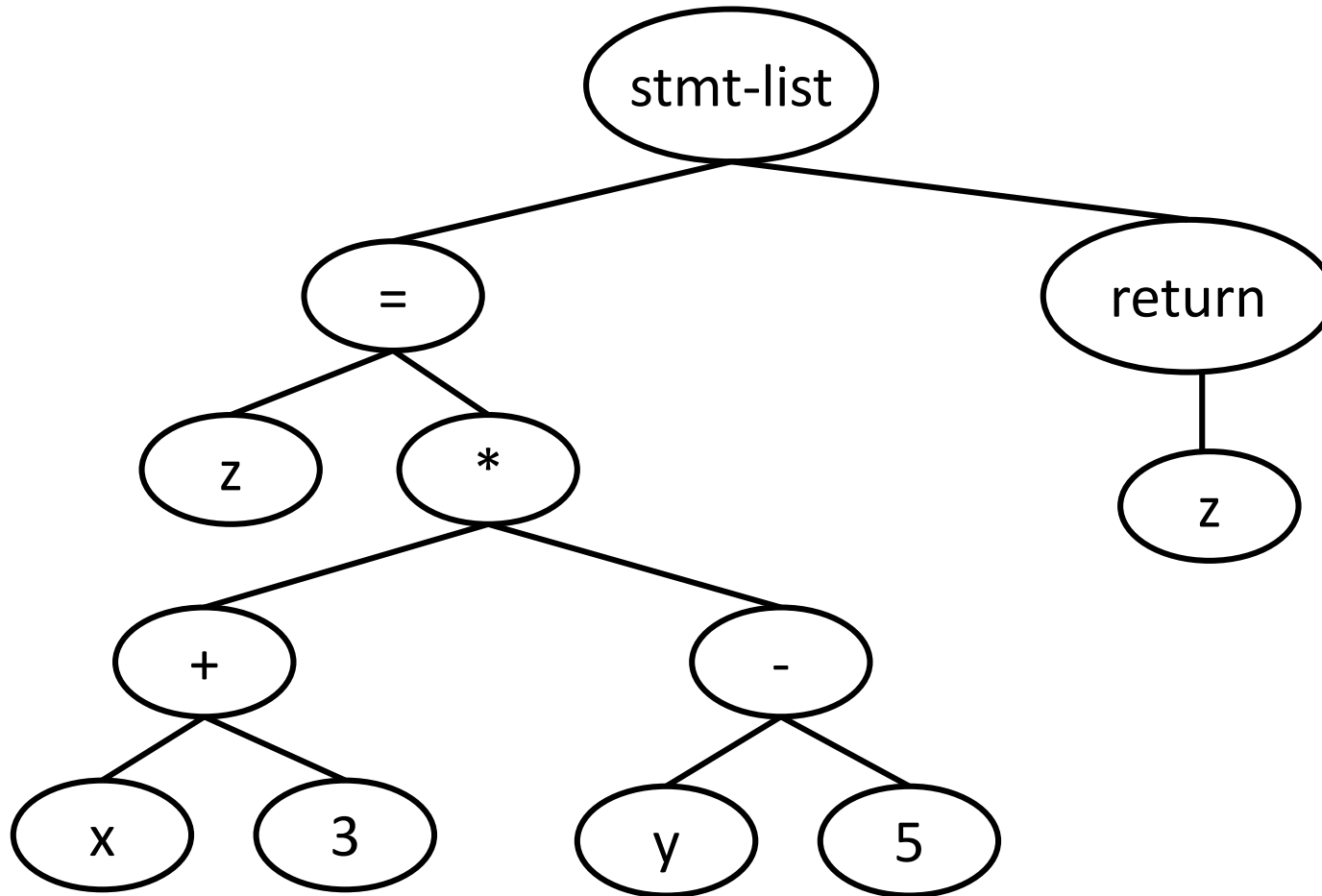


$$d \leftarrow s_1 + s_2$$

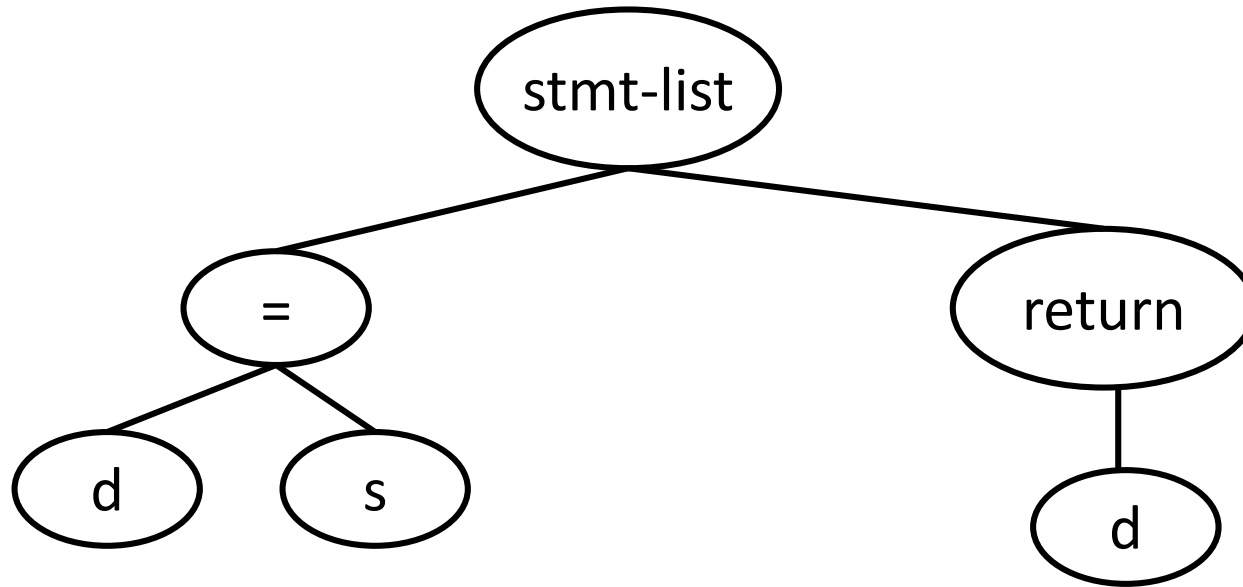


$$d \leftarrow c$$

Tiling a Tree



Better Tiles



Better or worse?

```
rax ← s  
return
```

Today

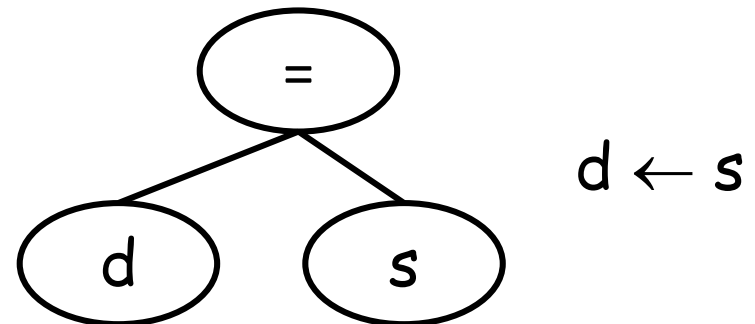
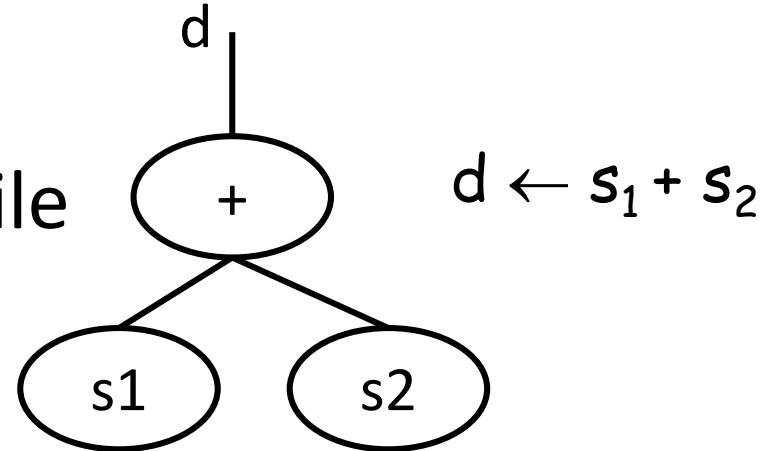
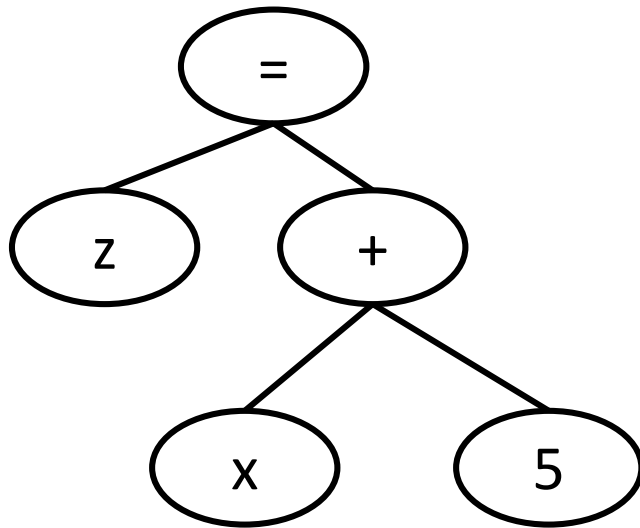
- Context
- Abstract Assembly
- $AST \rightarrow IR$
- Maximal Munch
- Issues
- Simple SSA
- x86 and 2-adr Instructions

Maximal Munch

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

Maximal Munch

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



Maximal Munch

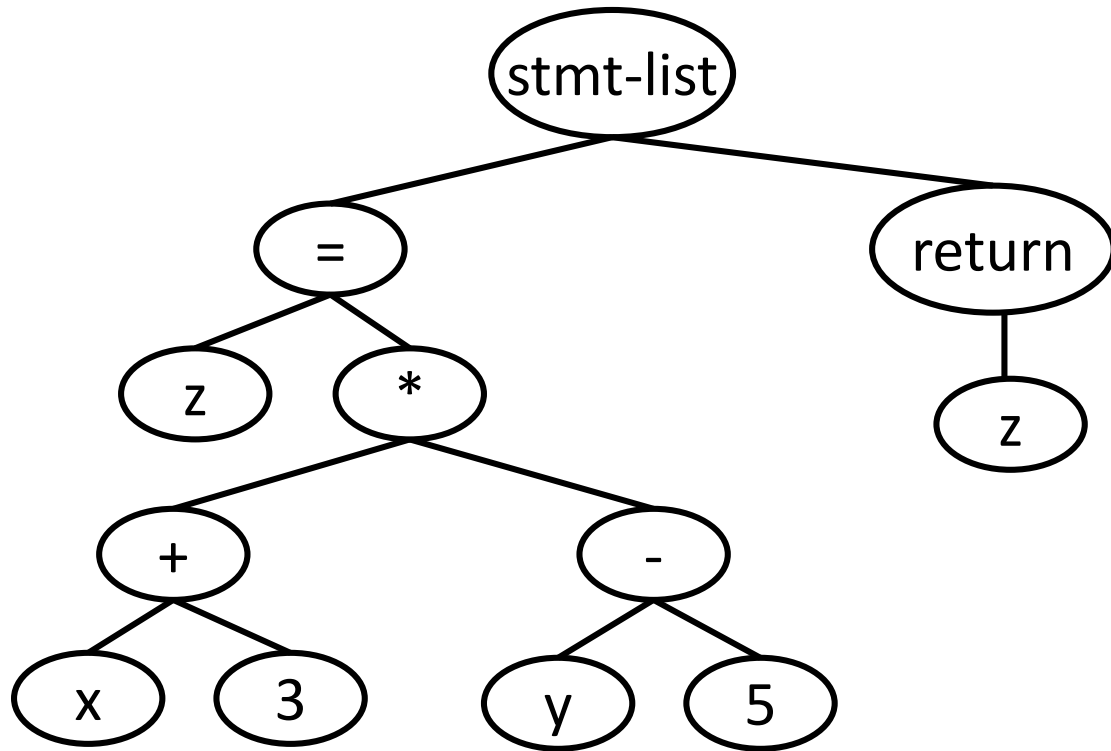
- recursively match tree
- At each step, pick “best” tile
- need to indicate what destinations are
 - choose either to supply destination
 - or generate a destination

codegen

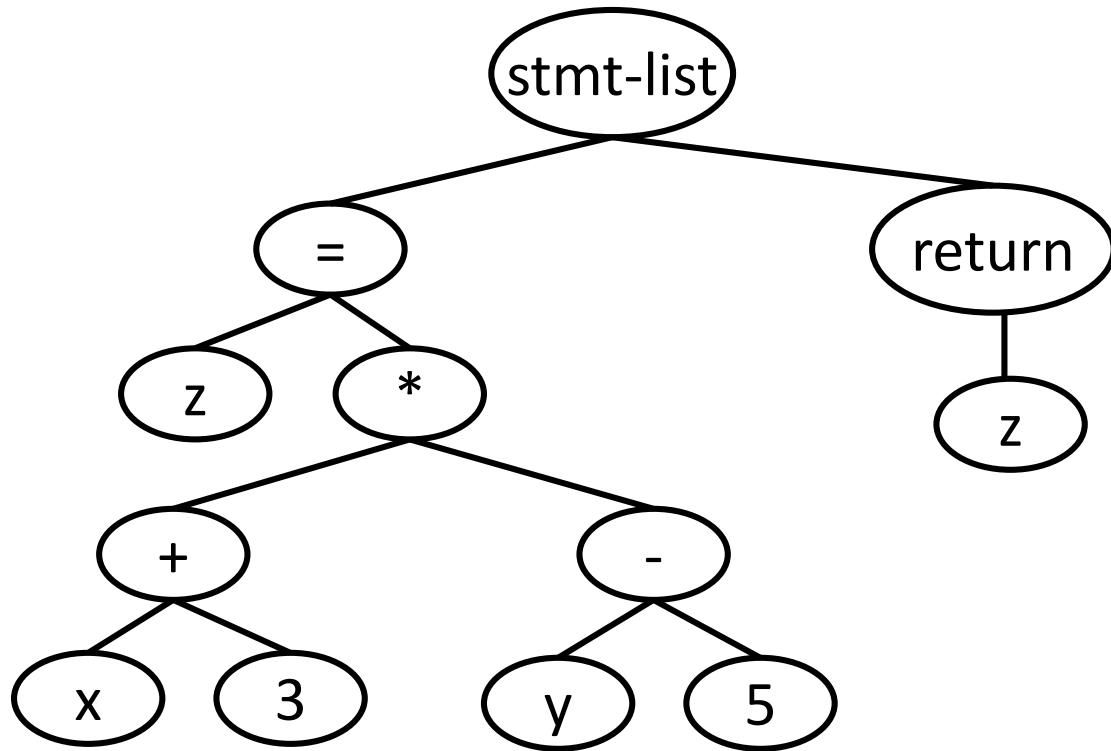
e	codegen(d, e)
c	$d \leftarrow c$
v	$d \leftarrow x$
$e_1 \oplus e_2$	codegen(t_1 , e_1) codegen(t_2 , e_2) $d \leftarrow t_1 \oplus t_2$

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

Example



Result



$t_3 \leftarrow x$

$t_4 \leftarrow 3$

$t_1 \leftarrow t_3 + t_4$

$t_5 \leftarrow y$

$t_6 \leftarrow 5$

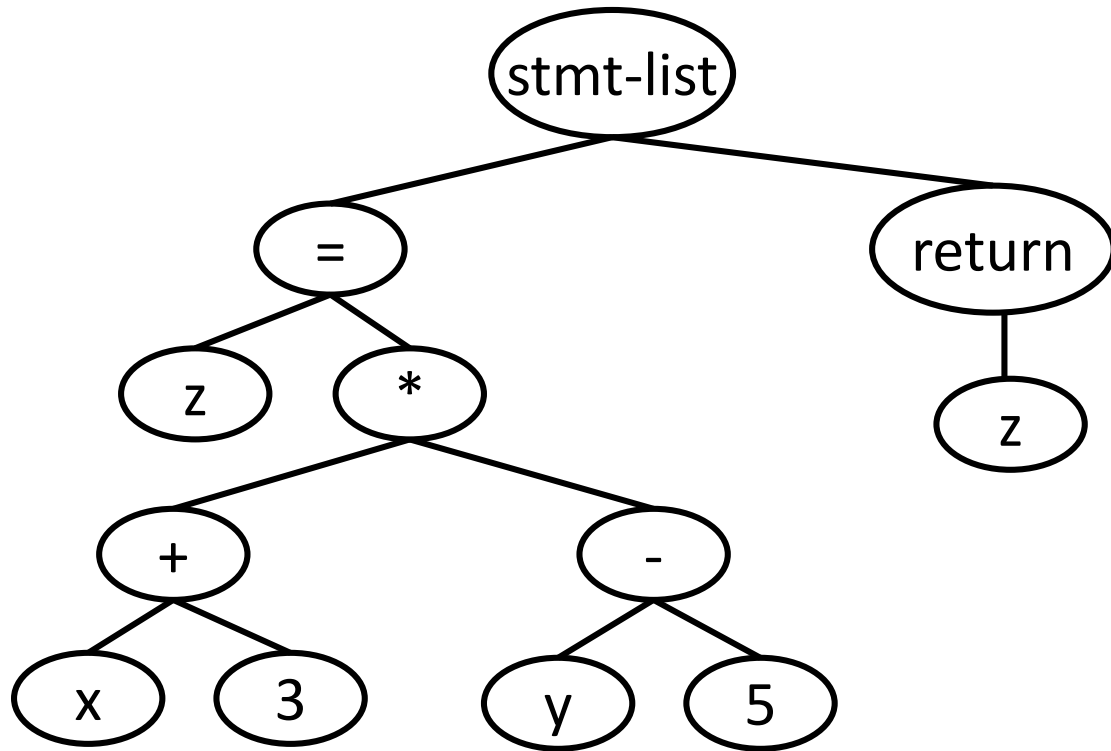
$t_2 \leftarrow t_5 * t_6$

$z \leftarrow t_1 * t_2$

$rax \leftarrow z$

ret

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

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) $d \leftarrow t_1 \oplus t_2$

Generally not recommended

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 Propagation

$t_3 \leftarrow x$

~~$t_4 \leftarrow 3$~~

$t_1 \leftarrow t_3 + t_4 \quad 3$

$t_5 \leftarrow y$

~~$t_6 \leftarrow 5$~~

$t_2 \leftarrow t_5 * t_6 \quad 5$

$z \leftarrow t_1 * t_2$

$rax \leftarrow z$

ret

Copy Propagation

~~$t_3 \leftarrow x$~~

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

~~$t_5 \leftarrow y$~~

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

$z \leftarrow t_1 * t_2$

$rax \leftarrow z$

ret

Have to be careful

- Constant propagation:

$x \leftarrow 5$

$y \leftarrow x - 4$

$x \leftarrow y + 7$

$z \leftarrow x$

- 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

x ← 5

y ← x - 4

x ← y + 7

z ← x

- Copy Propagation:

x ← y

y ← u - 4

z ← x + 7

Have to be careful

- Constant propagation:

- Can't just replace
all x's with 5

- Stop if x is redefined

x ← 5

y ← x - 4

x ← y + 7

z ← x

- Copy Propagation:

- Can't just replace
all x's with y's

- Stop if x or y is redefined

x ← y

y ← u - 4

z ← x + 7

Today

- Context
- Abstract Assembly
- $AST \rightarrow IR$
- 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 propagation.
- 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 propagation.
- 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.

$x \leftarrow 5$

$y \leftarrow x - 4$

$x \leftarrow y + 7$

$z \leftarrow x$

$x_0 \leftarrow 5$

$y \leftarrow x - 4$

$x \leftarrow y + 7$

$z \leftarrow x$

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.

$x \leftarrow 5$

$y \leftarrow x - 4$

$x \leftarrow y + 7$

$z \leftarrow x$

$x_0 \leftarrow 5$

$y \leftarrow x_0 - 4$

$x \leftarrow y + 7$

$z \leftarrow x$

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.

$x \leftarrow 5$

$y \leftarrow x - 4$

$x \leftarrow y + 7$

$z \leftarrow x$

$x_0 \leftarrow 5$

$y_0 \leftarrow x_0 - 4$

$x \leftarrow y + 7$

$z \leftarrow x$

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.

$x \leftarrow 5$

$y \leftarrow x - 4$

$x \leftarrow y + 7$

$z \leftarrow x$

$x_0 \leftarrow 5$

$y_0 \leftarrow x_0 - 4$

$x_1 \leftarrow y_0 + 7$

$z \leftarrow x$

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.

$x \leftarrow 5$

$y \leftarrow x - 4$

$x \leftarrow y + 7$

$z \leftarrow x$

$x_0 \leftarrow 5$

$y_0 \leftarrow x_0 - 4$

$x_1 \leftarrow y_0 + 7$

$z_0 \leftarrow x_1$

Now easy

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

$$x_0 \leftarrow 5$$

$$y_0 \leftarrow x_0 - 4$$

$$x_1 \leftarrow y_0 + 7$$

$$z_0 \leftarrow x_1$$

- Copy Propagation:
 - Can replace all x_0 with y_0

$$x_0 \leftarrow y_0$$

$$y_1 \leftarrow u_0 - 4$$

$$z_0 \leftarrow x_0 + 7$$

Today

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

How to real assembly on x86

- x86 doesn't have 3 address instructions!

$$d \leftarrow s_1 + s_2$$

How to real assembly on x86

- x86 doesn't have 3 address instructions!

$$d \leftarrow s_1 + s_2$$

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

How to 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.