Check-In

Review: Abstract Interpretation

Draw the CFG for the following 3AC procedure. Indicate the IN and OUT sets for each basic block on a, b, c for a constant propagation analysis

Assume c is global and all other vars are local

```
fun_foo:    enter foo
     L1:    getarg 1 [a]
     L2:    [b] := 2
     L3:    [c] := 2
     L4:    [tmp0] := [a] LT64 3
     L5:    IFZ [tmp0] GOTO L11
     L6:    [tmp1] := [b] ADD64 7
     L7:    [b] := [tmp1]
     L8:    call bar
     L9:    [tmp3] := [c] ADD64 7
     L10:   [c] := [tmp3]
     L11:   setret [b]
     L12:    leave foo
```

Announcements

Administrivia

- Quiz 4 Friday
- Review Session Wednesday, 7:15 – 9:15 (I’ll try to show up at 7:00)

Drew Davidson | University of Kansas

EECS 665 COMPILER CONSTRUCTION

SSA

Previously…

Abstract Interpretation

Rounding out dataflow analysis concepts

- Some more examples
- Considering more complex code
- Dataflow Framework

Abstract Interpretation

- Concepts
- Examples

You should know

- The saturation approach to dataflow
- Handling loops, globals, large domains

Optimization

Today's Lecture Outline

SSA

Static Single Assignment

- Motivation
- Concept
- Importance
- Implementation

Optimization

Recall Data Allocation

SSA – Motivation

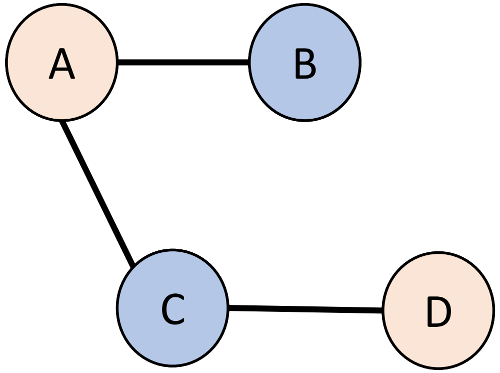
Simplistic Interference Graph:

- Nodes are “variables”
- Edges indicate interference

2-colorable

```
1. [A] := 1
2. [B] := 2
3. output [B]
4. [C] := 3
5. output [A]
6. [D] := 4
7. output [D]
8. output [C]
```

A live: (1, 5]
B live: (2, 3]
C live: (4, 8]
D live: (6, 7]



Recall Data Allocation

SSA – Motivation

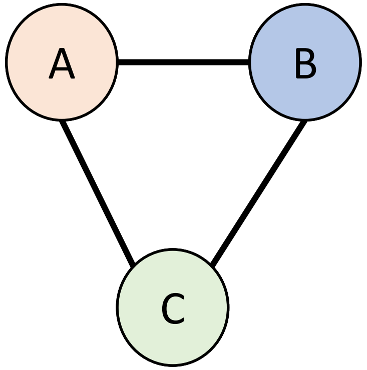
3-colorable

```
1. [A] := 1
2. [B] := 2
3. output [B]
4. [C] := 3
5. output [A]
6. [B] := 4
7. output [B]
8. output [C]
```

A live: (1, 5]

B live: (2, 3] and (6,7]

C live: (4, 8]



Breaking out B into *more* variables uses *fewer* resources!

2-colorable

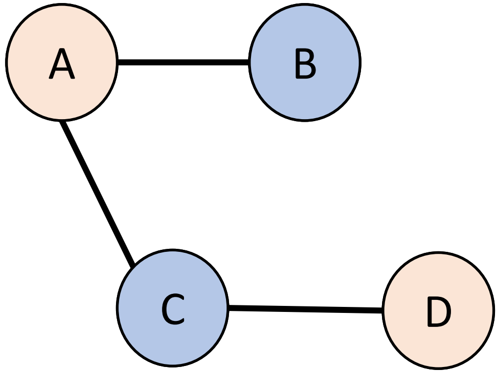
```
1. [A] := 1
2. [B] := 2
3. output [B]
4. [C] := 3
5. output [A]
6. [D] := 4
7. output [D]
8. output [C]
```

A live: (1, 5]

B live: (2, 3]

C live: (4, 8]

D live: (6, 7]



The Static Single Assignment Concept

SSA

An additional restriction on the IR:

- Every variable is assigned a value in *at most one* program point

We can say 3AC is (or isn't) in *SSA form*

Why does that matter?

Disentangles value use

Simplifies other analyses

```
a := 1
b := a
c := a + b
```

✓

```
a := 1
b := a
a := b * 2
c := a + b
```

✗

```
L1: b := 7
 goto L1
```

✓

Ok! statically defined only once (doesn't matter that it's dynamically assigned > 1)

Transformation to SSA Form

SSA

Basic Idea

- Break noncompliant variables into multiple "versions"
- Preserve semantics!

Obvious within a BBL

- Each definition rewritten to a new variable version
- Each use rewritten to the most recently defined variable version

Before **(not SSA form)**	**After** **(is SSA form)**
[a] := 1	[a_1] := 1
[b] := [a]	[b] := [a_1]
[a] := [b] * 2	[a_2] := [b] * 2
[c] := [a] + [b]	[c] := [a_2] + [b]

quick note on notation:
Ok to leave off the subscript
if there's only one "version"

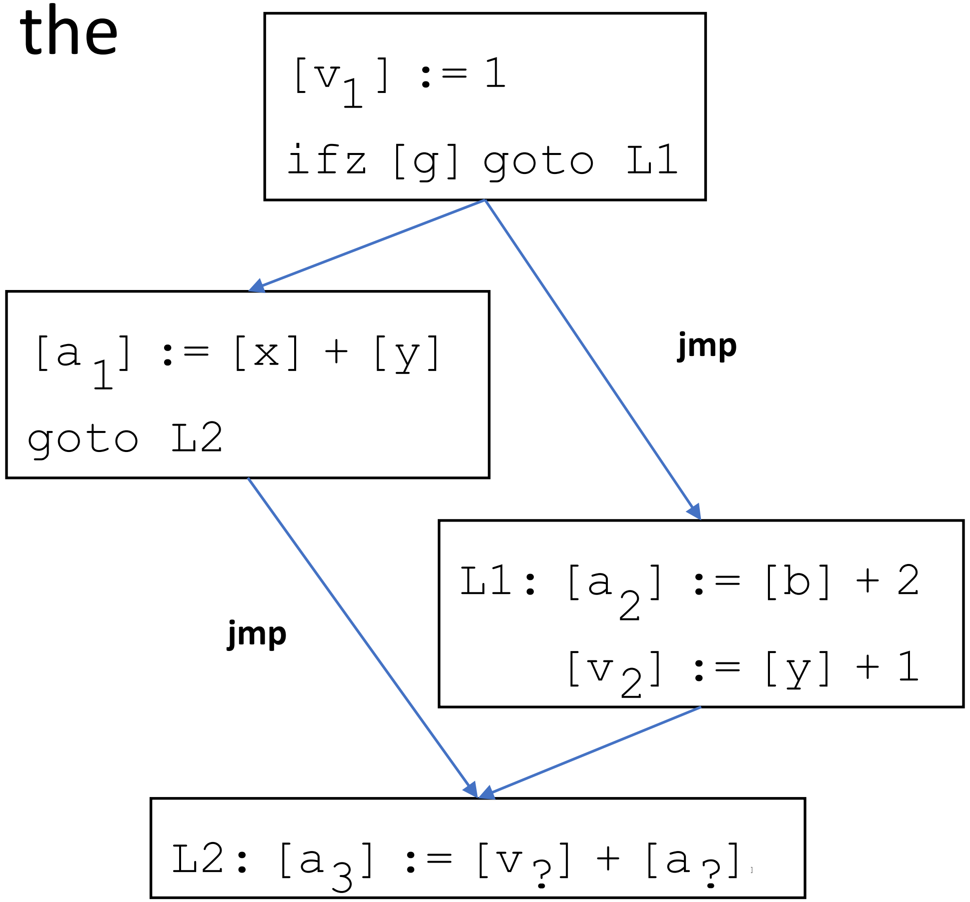
Transformation to SSA Form

SSA

Non-Obvious between BBLs

- Don’t know (statically) the most recently defined variable version

```
    [v]  :=  1
    ifz  [g]  goto L1
    [a]  :=  [x]  +  [y]
    goto  L2
L1: [a]  :=  [b]  +  2
    [v]  :=  [y]  +  1
L2: [a]  :=  [v]  +  [a]
```



[v_1] := 1
ifz [g] goto L1

[a_1] := [x] + [y]
goto L2

L1: [a_2] := [b] + 2
[v_2] := [y] + 1

L2: [a_3] := [$v_?$] + [$a_?$]

ϕ Functions – Notational Placeholders

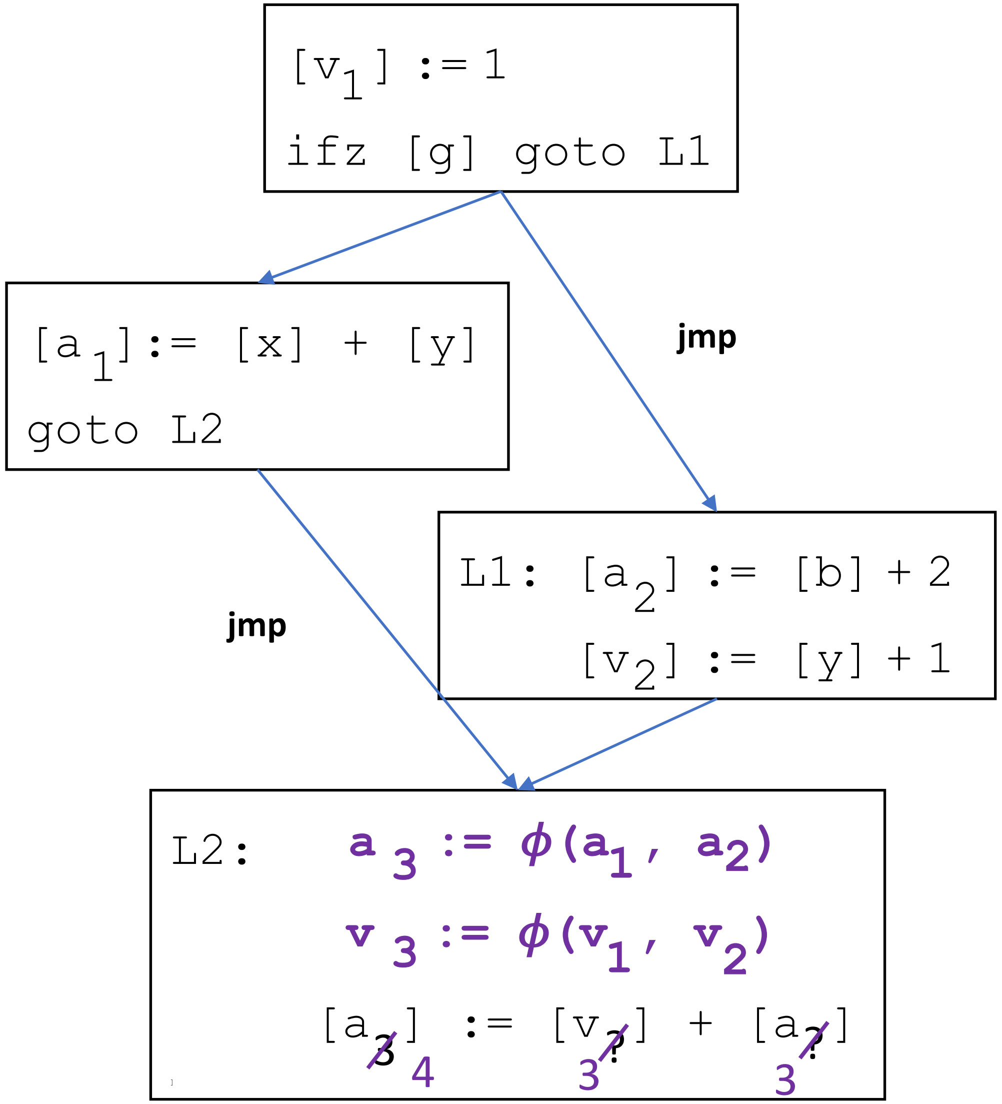
Encapsulated the uncertainty of which version to use

$$\boxed{a_4 := \phi(a_1,\ a_2,\ a_3\ )}$$

means that a_4 will hold whichever version of **a** was defined most recently

ϕ Functions – Resolving "Conflicts"

SSA – ϕ Functions



$[v_1] := 1$

ifz [g] goto L1

$[a_1] :=$ [x] + [y]

goto L2

jmp

L1: $[a_2] :=$ [b] + 2

$[v_2] :=$ [y] + 1

jmp

L2: $a_3 := \phi(a_1, a_2)$

$v_3 := \phi(v_1, v_2)$

$[a_{\cancel{3}\,4}] := [v_{\cancel{?}\,3}] + [a_{\cancel{?}\,3}]$

Example Time – Transform to SSA Form

SSA – ϕ Functions

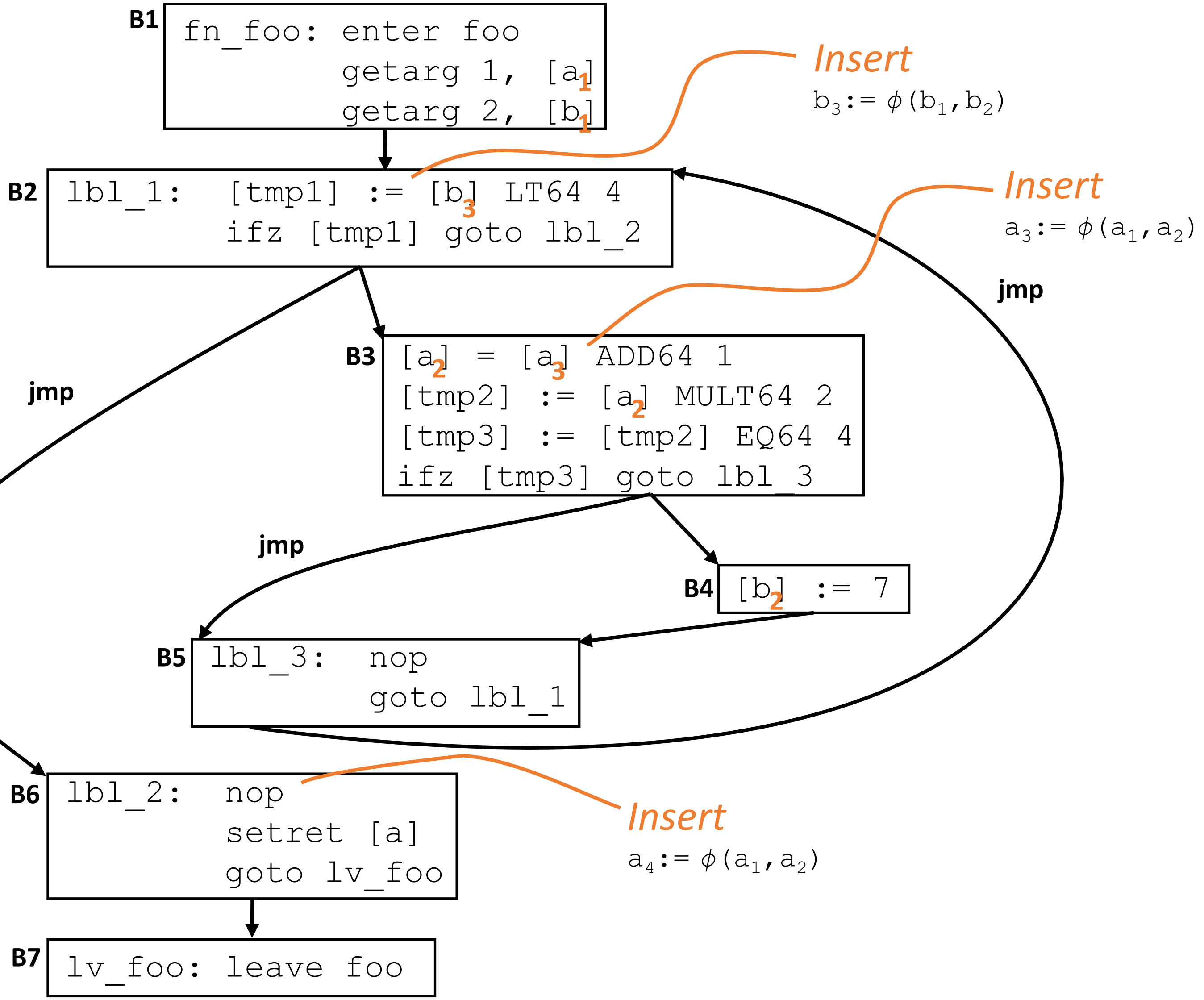
```
int foo(int a, int b){
  while(b < 4){
    a += 1;
    if ( a * 2 == 4){
      b = 7;
    }
  }
  return a;
}
```

```
B1  fn_foo: enter foo
            getarg 1, [a]
            getarg 2, [b]
B2  lbl_1:  [tmp1] := [b] LT64 4
            ifz [tmp1] goto lbl_2
B3          [a] = [a] ADD64 1
            [tmp2] := [a] MULT64 2
            [tmp3] := [tmp2] EQ64 4
            ifz [tmp3] goto lbl_3
B4          [b] := 7
B5  lbl_3:  nop
            goto lbl_1
B6  lbl_2:  nop
            setret [a]
            goto lv_foo
B7  lv_foo: leave foo
```

Example Time – Transform to SSA Form

SSA – ϕ Functions

```
int foo(int a, int b){
  while(b < 4){
    a += 1;
    if ( a * 2 == 4){
      b = 7;
    }
  }
  return a;
}
```



B1
```
fn_foo: enter foo
        getarg 1, [a1]
        getarg 2, [b1]
```

B2
```
lbl_1:  [tmp1] := [b3] LT64 4
        ifz [tmp1] goto lbl_2
```

B3
```
[a2] = [a3] ADD64 1
[tmp2] := [a2] MULT64 2
[tmp3] := [tmp2] EQ64 4
ifz [tmp3] goto lbl_3
```

B4
```
[b2] := 7
```

B5
```
lbl_3:  nop
        goto lbl_1
```

B6
```
lbl_2:  nop
        setret [a]
        goto lv_foo
```

B7
```
lv_foo: leave foo
```

Insert
$b_3 := \phi(b_1, b_2)$

Insert
$a_3 := \phi(a_1, a_2)$

Insert
$a_4 := \phi(a_1, a_2)$

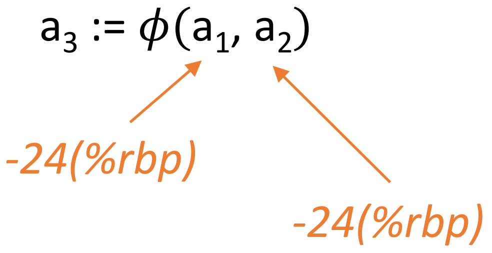
ϕ Functions – A "Magical" Placeholder

Why rely on a function we cannot compute?

We can remove the ϕs later

- Easy solution: make sure that all arguments to the ϕ share a common memory location

Image Credit: Avyst e-forms wizard



$a_3 := \phi(a_1, a_2)$

-24(%rbp)

-24(%rbp)

ϕ Functions are Costly!

SSA – Placing ϕs

Rolls back our sub-variable resource goals

- Consider a naïve algorithm to place ϕs:
  - Place ϕ for every defined version of the variable

What Points Actually Require ϕ?

SSA – Placing ϕs

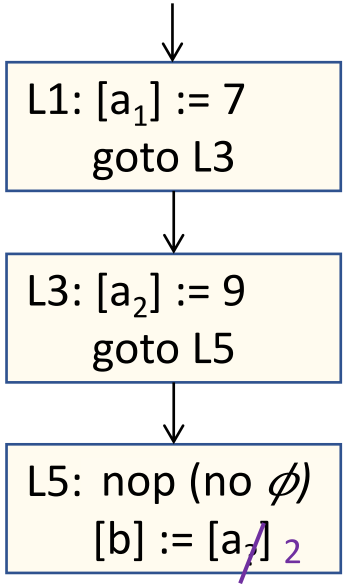
One sufficient condition for Avoiding ϕ nodes:

(wlog, assume Block A defines x and Block B uses x)

- Block B has an *unambiguous variable definition* if you're guaranteed to go through block A on any path to B

There's a name for this constraint…

Possible CFG Snippet 1

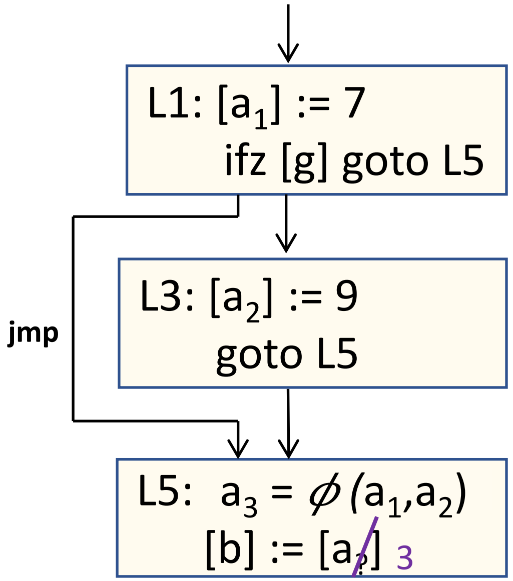


L1: $[a_1]$:= 7
goto L3

L3: $[a_2]$:= 9
goto L5

L5: nop (no ϕ)
[b] := $[a_{\cancel{?}}]$ 2

Possible CFG Snippet 2



L1: $[a_1]$:= 7
ifz [g] goto L5

jmp

L3: $[a_2]$:= 9
goto L5

L5: $a_3 = \phi(a_1,a_2)$
[b] := $[a_{\cancel{?}}]$ 3

Domination

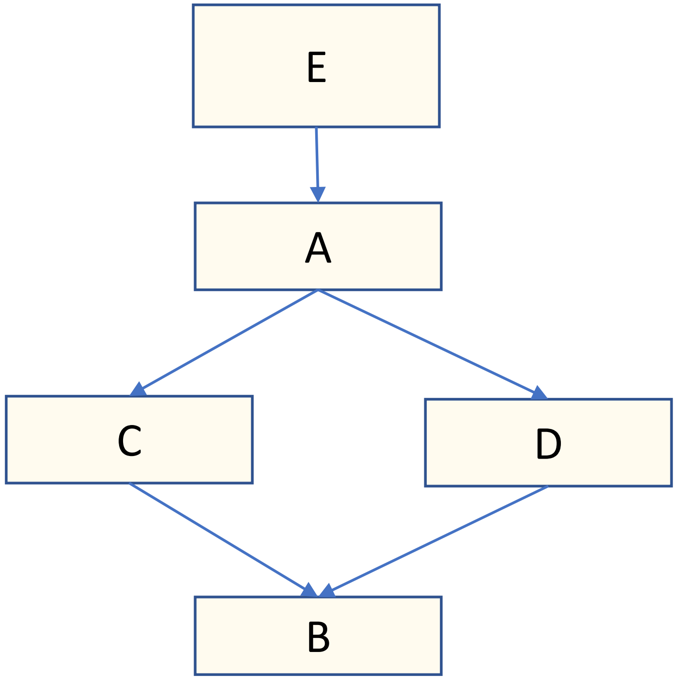
Domination Examples

SSA – Placing ϕs

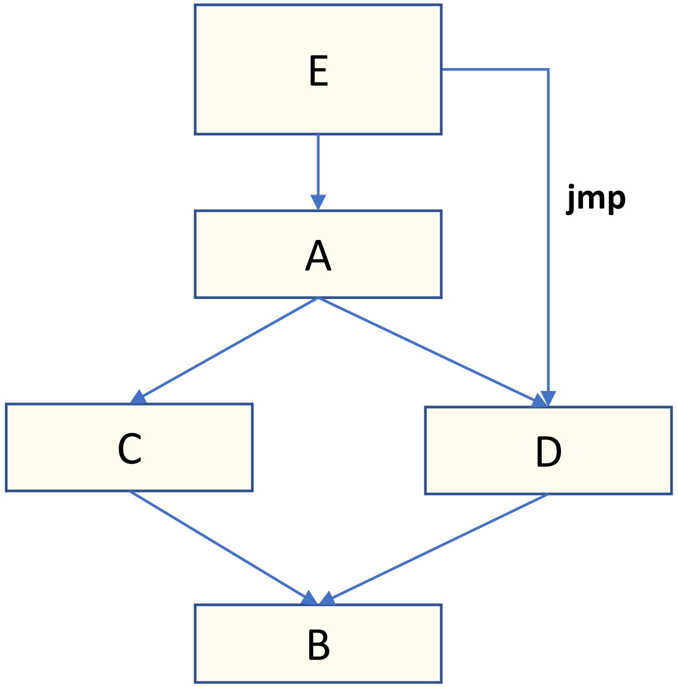
Block X **dominates** block Y if all paths to Y must pass through X

Examples *(what does A dominate?)*

A **dominates** A, D, C, B



A **dominates** A and C only



Domination Vocabulary

SSA – Placing ϕs

X DOM Y – *X dominates Y*

- All paths to Y go through X
- (Reflexive - X DOM X)

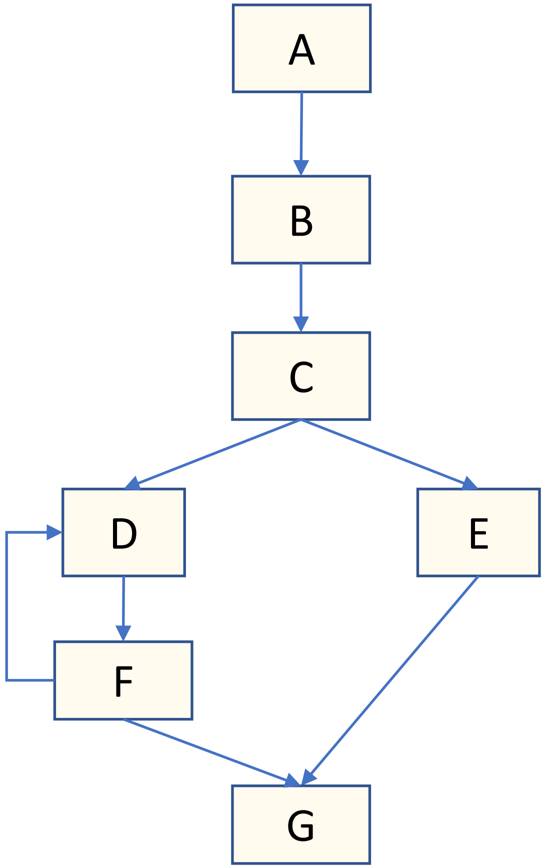
X SDOM Y – *X strictly dominates Y*

- Non-reflexive domination
- Formally: X DOM Y and X != Y

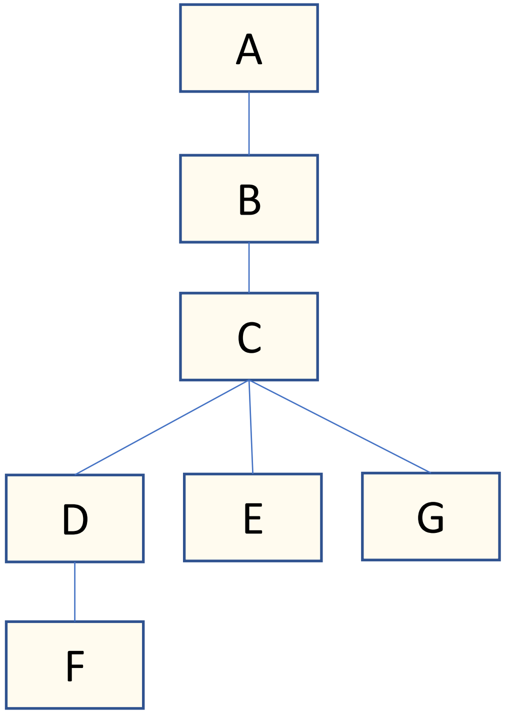
X IDOM Y – *X immediately dominates Y*

- “Closest” strict dominator
- Formally: X SDOM Y and Z SDOM Y ⇒ Z = X

Control-Flow Graph



Dominator Tree



What Good is Domination?

SSA – Placing ϕs

Provides guarantees about execution (sorta-kinda like a looser version of statements being in the same basic block)

- A given block can rely on statements in a dominator to always have happened before the block is executed
- Similarly, a given block cannot rely on statements in non-dominators to always have happened before the block is executed

The boundary has interesting properties for SSA

Wdetour: Using Dominators for ϕs

SSA – Placing ϕs

Domination Vocabulary

SSA – Placing ϕs

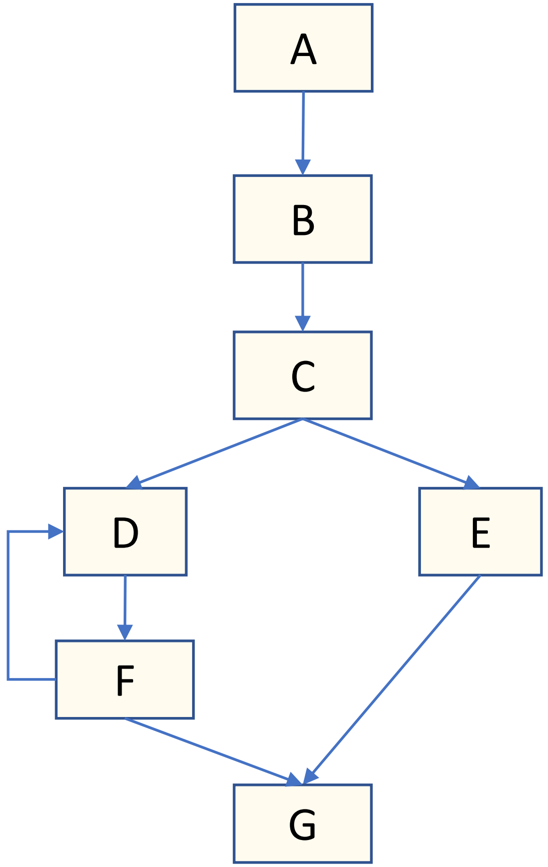
DETOUR

Dominator Frontier of X:

The set of nodes k_i

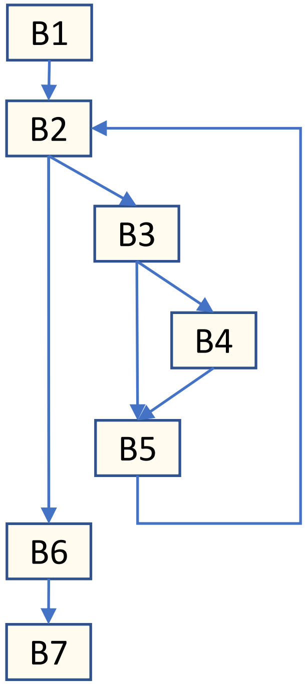
that X does not strictly dominate,

but X dominates an immediate predecessor of k_i



Example Time – Compute Dom Frontier

SSA – ϕ Functions



B1 What does B1 dominate? B1 B2 B3 B4 B5 B6 B7

What do these precede? ~~B2~~ ~~B3~~ ~~B6~~ ~~B4~~ ~~B5~~ ~~B2~~ ~~B7~~

Disqualify if B1 SDOMs

Dominator Frontier of X:

The set of nodes k_i

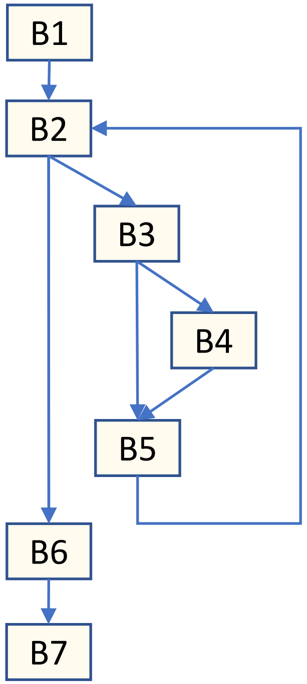
! X SDOM k_i

X DOM Y and Y IPRED k_i

BBL	IPRED	DOM	SDOM	DF
B1	B2	(all)	B2,B3,B4,B5,B6,B7	{}
B2	B3, B6	B2,B3,B4,B5,B6,B7	B3,B4,B5,B6,B7	
B3	B4,B5	B3, B4,B5	B4,B5	
B4	B5	B4	{}	
B5	B2	B5	{}	
B6	B7	B6,B7	B7	
B7	{}	B7	{}	

Example Time – Compute Dom Frontier

SSA – ϕ Functions



Dominator Frontier of X:

The set of nodes k_i

! X SDOM k_i

X DOM Y and Y IPRED k_i

B1 What does B1 dominate? B1 B2 B3 B4 B5 B6 B7

What do these precede? ~~B2~~ ~~B3~~ ~~B6~~ ~~B4~~ ~~B5~~ ~~B2~~ ~~B7~~

Disqualify if B1 SDOMs

B2 What does B2 dominate? B2 B3 B4 B5 B6 B7

What do these precede? ~~B3~~ ~~B6~~ ~~B4~~ ~~B5~~ ~~B5~~ B2 ~~B7~~

Disqualify if B2 SDOMs

B3 What does B3 dominate? B3 B4 B5

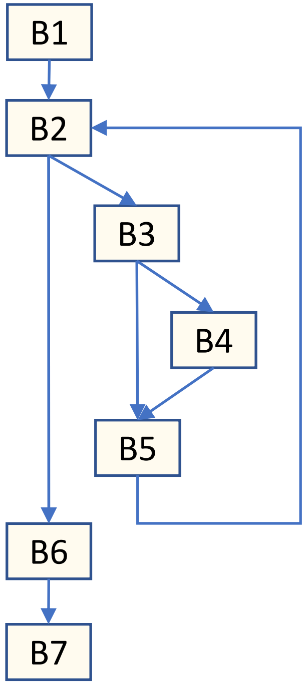
What do these precede? ~~B4~~ ~~B5~~ ~~B5~~ B2

Disqualify if B3 SDOMs

BBL	IPRED	DOM	SDOM	DF
B1	B2	(all)	B2,B3,B4,B5,B6,B7	{}
B2	B3, B6	B2,B3,B4,B5,B6,B7	B3,B4,B5,B6,B7	B2
B3	B4,B5	B3, B4,B5	B4,B5	B2
B4	B5	B4	{}	
B5	B2	B5	{}	
B6	B7	B6,B7	B7	
B7	{}	B7	{}	

Example Time – Compute Dom Frontier

SSA – ϕ Functions



Dominator Frontier of X:

The set of nodes k_i

! X SDOM k_i

X DOM Y and Y IPRED k_i

BBL	IPRED	DOM	SDOM	DF
B1	B2	(all)	B2,B3,B4,B5,B6,B7	{}
B2	B3, B6	B2,B3,B4,B5,B6,B7	B3,B4,B5,B6,B7	B2
B3	B4,B5	B3, B4,B5	B4,B5	B2
B4	B5	B4	{}	B5
B5	B2	B5	{}	B2
B6	B7	B6,B7	B7	
B7	{}	B7	{}	

B1 What does B1 dominate? B1 B2 B3 B4 B5 B6 B7
What do these precede? ~~B2~~ ~~B3~~ ~~B6~~ ~~B4~~ ~~B5~~ ~~B2~~ ~~B7~~
Disqualify if B1 SDOMs

B2 What does B2 dominate? B2 B3 B4 B5 B6 B7
What do these precede? ~~B3~~ ~~B6~~ ~~B4~~ ~~B5~~ ~~B5~~ B2 ~~B7~~
Disqualify if B2 SDOMs

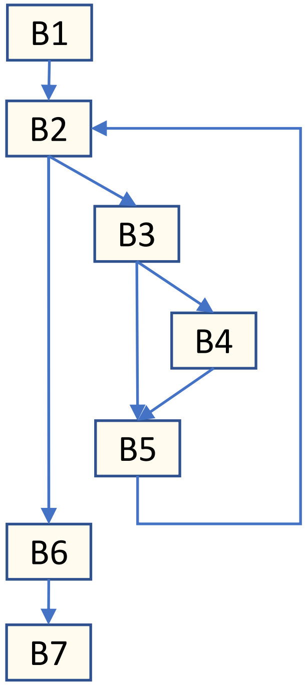
B3 What does B3 dominate? B3 B4 B5
What do these precede? ~~B4~~ ~~B5~~ ~~B5~~ B2
Disqualify if B3 SDOMs

B4 What does B4 dominate? B4
What do these precede? B5
Disqualify if B4 SDOMs

B5 What does B5 dominate? B5
What do these precede? B2
Disqualify if B5 SDOMs

Example Time – Compute Dom Frontier

SSA – ϕ Functions



Dominator Frontier of X:

The set of nodes k_i

! X SDOM k_i

X DOM Y and Y IPRED k_i

BBL	IPRED	DOM	SDOM	DF
B1	B2	(all)	B2,B3,B4,B5,B6,B7	{}
B2	B3, B6	B2,B3,B4,B5,B6,B7	B3,B4,B5,B6,B7	B2
B3	B4,B5	B3, B4,B5	B4,B5	B2
B4	B5	B4	{}	B5
B5	B2	B5	{}	B2
B6	B7	B6,B7	B7	{}
B7	{}	B7	{}	{}

B1 What does B1 dominate? B1 B2 B3 B4 B5 B6 B7
What do these precede? ~~B2~~ ~~B3~~ ~~B6~~ ~~B4~~ ~~B5~~ ~~B2~~ ~~B7~~
Disqualify if B1 SDOMs

B2 What does B2 dominate? B2 B3 B4 B5 B6 B7
What do these precede? ~~B3~~ ~~B6~~ ~~B4~~ ~~B5~~ ~~B5~~ B2 ~~B7~~
Disqualify if B2 SDOMs

B3 What does B3 dominate? B3 B4 B5
What do these precede? ~~B4~~ ~~B5~~ ~~B5~~ B2
Disqualify if B3 SDOMs

B4 What does B4 dominate? B4
What do these precede? B5
Disqualify if B4 SDOMs

B5 What does B5 dominate? B5
What do these precede? B2
Disqualify if B5 SDOMs

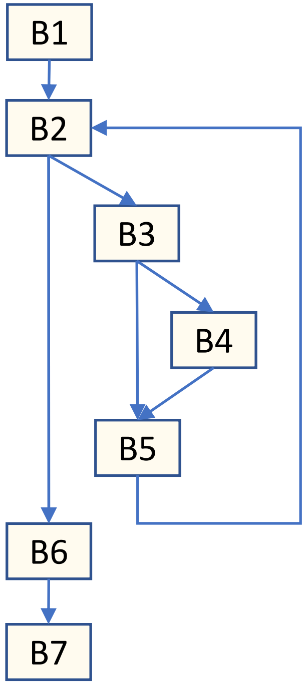
B6 What does B6 dominate? B6 B7
What do these precede? ~~B7~~
Disqualify if B6 SDOMs

B7 What does B7 dominate? B7
What do these precede? {}

Example Time – Compute Dom Frontier

SSA – ϕ Functions

DETOUR



Dominator Frontier of X:

The set of nodes k_i

! X SDOM k_i

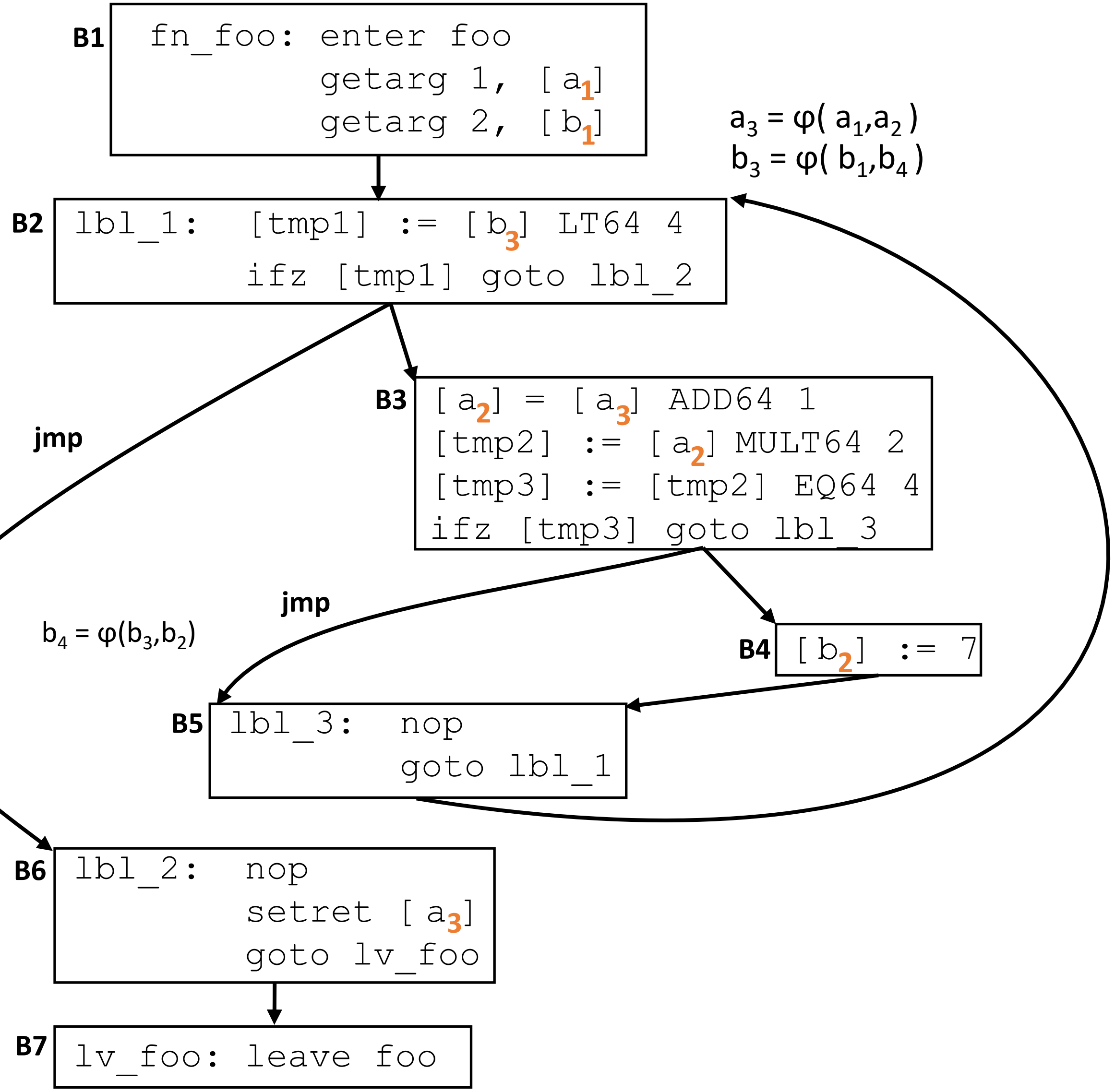
X DOM Y and Y IPRED k_i

```
for v in vars:
   for d in DefBBLs[v]:
     for block in DF[d]:
       Add a ϕ-node to block,
         unless we have done so already.
       Add block to DefBBLs[v]
         unless it's already in there.
```

BBL	IPRED	DOM	SDOM	DF
B1	B2	(all)	B2,B3,B4,B5,B6,B7	{}
B2	B3, B6	B2,B3,B4,B5,B6,B7	B3,B4,B5,B6,B7	B2
B3	B4,B5	B3, B4,B5	B4,B5	B2
B4	B5	B4	{}	B5
B5	B2	B5	{}	B2
B6	B7	B6,B7	B7	{}
B7	{}	B7	{}	{}

Example Time – Compute Dom Frontier

SSA – ϕ Functions



B1 fn_foo: enter foo
getarg 1, [a_1]
getarg 2, [b_1]

$a_3 = \varphi(a_1,a_2)$
$b_3 = \varphi(b_1,b_4)$

B2 lbl_1: [tmp1] := [b_3] LT64 4
ifz [tmp1] goto lbl_2

B3 [a_2] = [a_3] ADD64 1
[tmp2] := [a_2] MULT64 2
[tmp3] := [tmp2] EQ64 4
ifz [tmp3] goto lbl_3

B4 [b_2] := 7

B5 lbl_3: nop
goto lbl_1

B6 lbl_2: nop
setret [a_3]
goto lv_foo

B7 lv_foo: leave foo

```
for v in vars:
   for d in DefBBLs[v]:
     for block in DF[d]:
       Add a ϕ-node to block,
         unless we have done so already.
       Add block to DefBBLs[v]
         unless it's already in there.
```

var	DefBBLs	Φ Blocks
a	B1 B3 B2	B2
b	B1 B4 B5 B2	B5 B2

BBL	IPRED	DOM	SDOM	DF
B1	B2	(all)	B2,B3,B4,B5,B6,B7	{}
B2	B3, B6	B3,B4,B5,B6,B7	B3,B4,B5,B6,B7	B2
B3	B4,B5	B3, B4,B5	B4,B5	B2
B4	B5	B4	{}	B5
B5	B2	B5	{}	B2
B6	B7	B6,B7	B7	{}
B7	{}	B7	{}	{}

End Detour: Using Dominators for ϕs

SSA – Placing ϕs

Dominance: Summary

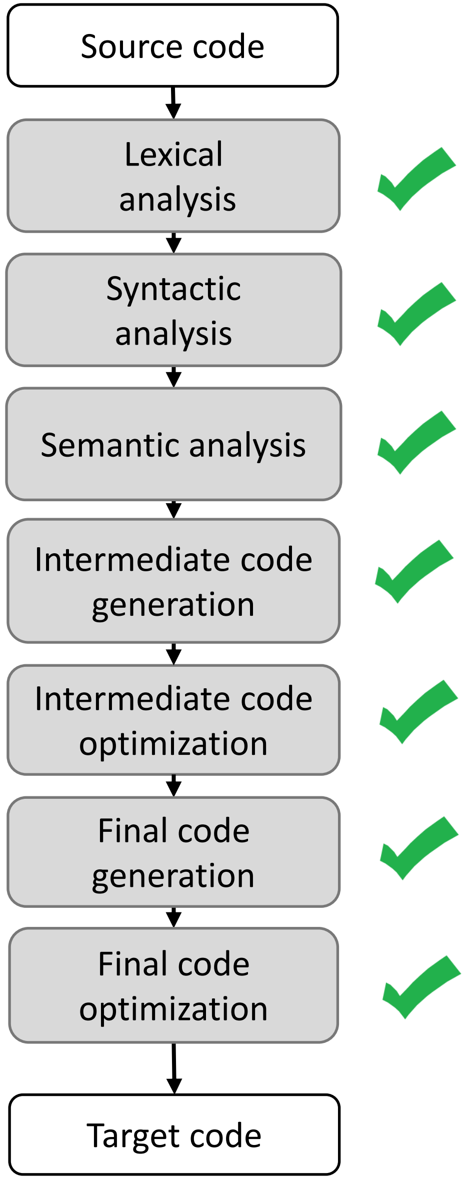
SSA – Placing ϕs

Summary:

- Dominators can be computed efficiently
- Dominance can be used to aid in efficient SSA
- SSA aids in efficient program optimization and future analysis

Oh Hey, We Built a Compiler!

Underview



COMPILER

LAND
A MILTON BRADLEY GAME
MOLASSES SWAMP
GLOPPY
CANDY CASTLE
KING KANDY
QUEEN FROSTINE
ICE CREAM SEA
GRAMMA NUTT
STUCK IN MOLASSES SWAMP LOSE ONE TURN
LOST IN LOLLIPOP WOODS LOSE ONE TURN
PRINCESS LOLLY
STUCK ON A GOOEY GUMDROP LOSE ONE TURN
LICORICE CASTLE
LORD LICORICE
GUMDROP PASS
JOLLY
RAINBOW TRAIL
PLUMPY
MR. MINT

Regular Languages

Lexical Analysis

Parsing

Syntactic Definition

Syntax-Dir Translation

Semantic Analysis

Intermediate Representation

Code Generation

Optimization

Machine Codegen

What Next?

Underview

Practical Applications

Why does this class matter?

- “So you can do compilers”: Practical skills for language implementation / reasoning
- “What you do with compilers is useful outside doing compilers”