EECS 483: Compiler Construction

Lecture 16:

Register Allocation Part 1: Liveness Analysis

March 17
Winter Semester 2025

Announcements

- Exam tomorrow evening, March 18 6-8pm.
- Location: 3 rooms in DOW. Which room is determined by your uniqname:
  - DOW1010 if your uniqname starts with A-JO
  - DOW1017 if your uniqname starts with JS-R
  - DOW1018 if your uniqname starts with S-Z
- Bring your own pen/pencil.
- 1 page of notes ("cheat sheet") allowed.
  - Standard letter size
  - Typed or hand-written ok

Assignment 4 released on Wednesday.

Where Were We?

We've discussed so far how to compile many features **correctly** (functional correctness) only worrying about preserving **asymptotic complexity**.

But our generated code has high **constant factors** in its complexity:

- Use stack-allocation for all local variables.

- Many redundant dynamic type checks

- Simple arithmetic is preserved even if we can evaluate it at compile time

...

Live code example

Where Were We?

We've discussed so far how to compile many features **correctly** (functional correctness) only worrying about preserving **asymptotic complexity**.

But our generated code has high **constant factors** in its complexity:

- Use stack-allocation for all local variables.

- Many redundant dynamic type checks

- Simple arithmetic is preserved even if we can evaluate it at compile time

...

Where Were We?

We've discussed so far how to compile many features **correctly** (functional correctness) only worrying about preserving **asymptotic complexity**.

But our generated code has high **constant factors** in its complexity:

- Use stack-allocation for all local variables.

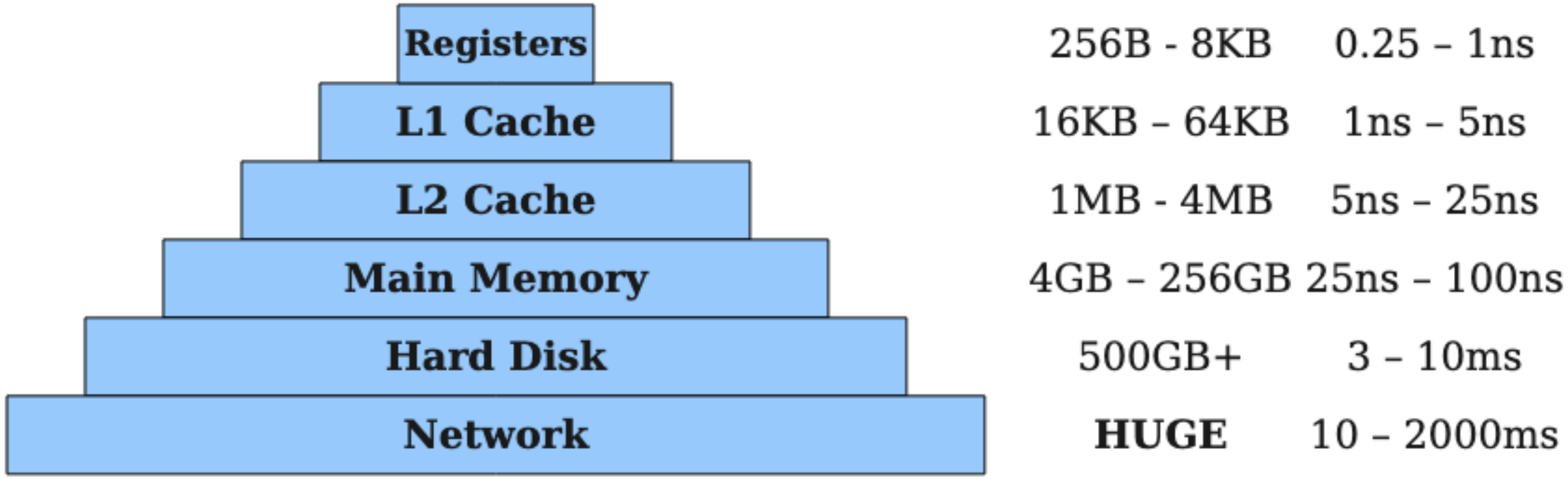
- Many redundant dynamic type checks

- Simple arithmetic is preserved even if we can evaluate it at compile time

...

Memory Hierarchy

Systems view of memory:



Snake/SSA view of memory

variables, heap allocated objects

Memory Allocation for Locals

For code generation, we need to map our SSA local variables to memory locations where they are stored.

Current strategy:

- Allocate all variables onto the stack, based on nesting of the current scope.
- Not completely naive: we do re-use some stack space in nested sub-blocks

Big Performance hit: need to move values in and out of registers frequently.

Register Allocation

For code generation, we need to map our SSA local variables to memory locations where they are stored.

Goal:

Store variable's values in registers whenever possible.

Only use stack space if we run out of registers.

Performance gains:

- 3-10x+ faster variable accesses (by far the **most important** optimization for a compiler)
- Space gain: smaller stack frames

High computational complexity: often the slowest part of the compiler

Register Allocation Examples

Register Allocation Examples

```
f(a):
  x = a * 2
  y = x + 7
  ret y
```

Register Allocation Examples

```
f(a):
  x = a * 2
  y = x + 7
  ret y
```

```
Currently:
a: stack [rsp – 8]
x: stack [rsp – 16]
y: stack [rsp – 32]
```

Register Allocation Examples

```
f(a):
  x = a * 2
  y = x + 7
  ret y
```

```
With register alloc:
a: rax
x: rax
y: rax
```

Register Allocation Examples

```
f(a):
  x = a * 2
  y = x + 7
  ret y

... assertInt
sar rax, 1
imul rax, 4
... assertInt
add rax, 14
ret
```

```
With register alloc:
a: rax
x: rax
y: rax
```

Register Allocation Examples

```
f(a):
  x = a * 2
  y = x + 7
  z = x * y
  ret z
```

```
With register alloc:
a: rax
x: rax
y: rcx
z: rax
```

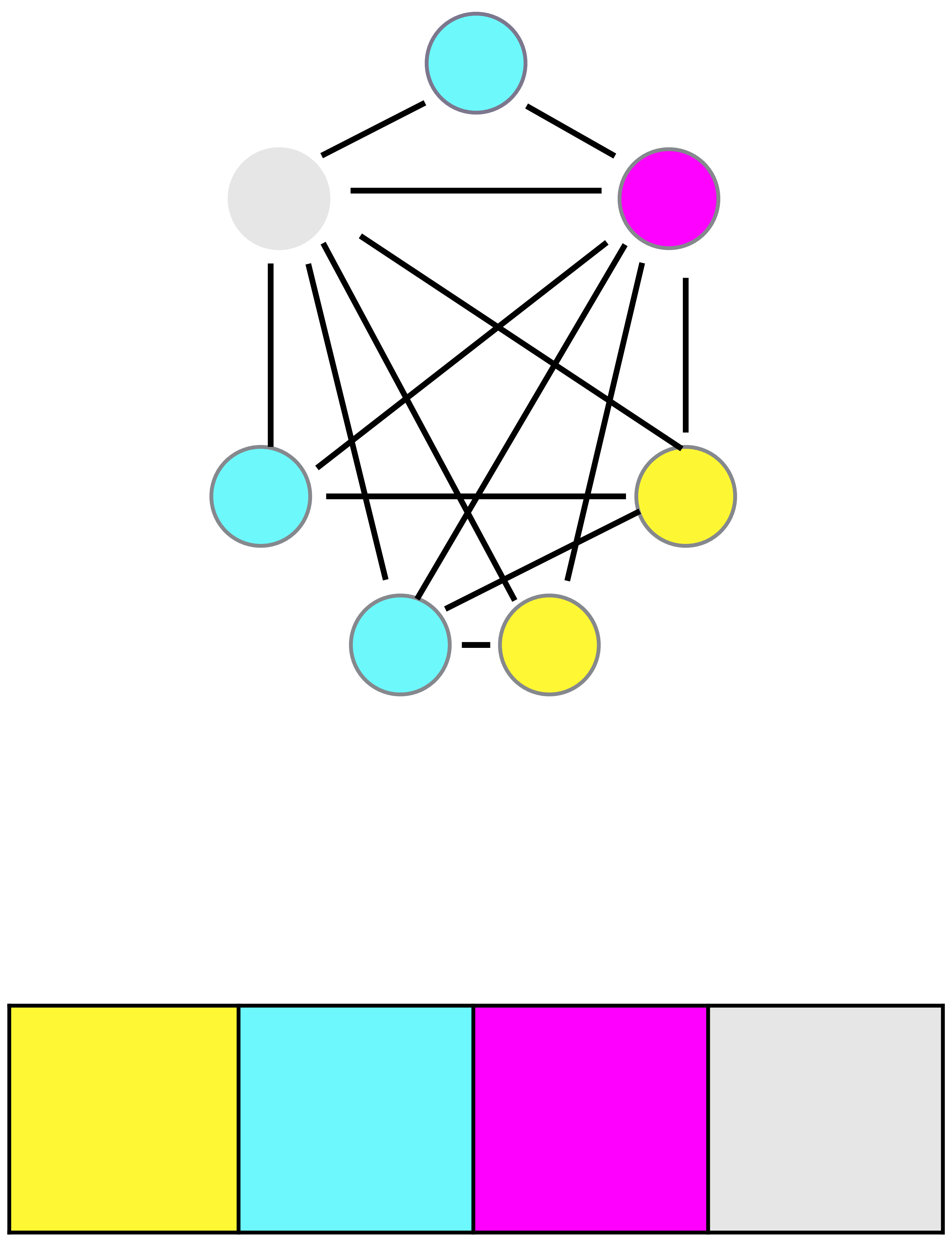
Can't put x and y in the same register
Say they are **in conflict** or **interfering**

Register Allocation: Graph Coloring Approach

The best register allocation algorithms (in terms of quality of output, not efficiency) use **graph coloring**

Graph coloring problem:

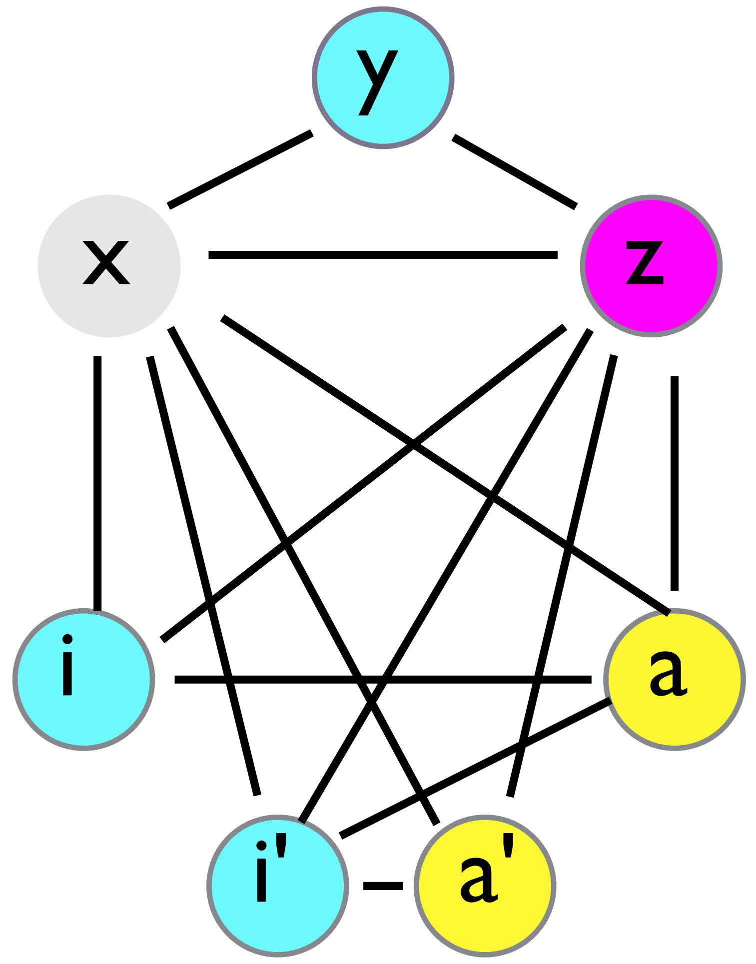
Given a graph (V, E) and set of colors K assign each variable a color so that adjacent nodes all have **different** colors.



Register Allocation: Graph Coloring Approach

Graph Coloring register allocation:

- Make an **interference graph**: vertices are variables, edges are **interference** relationships
- Colors are the different registers
- A solution is a valid register assignment



rax	rsi	rdx	rdi

Register Allocation Overview

Break down into three tasks:

1. **Liveness analysis:** determine which values are needed at every program point

2. **Conflict analysis**: use liveness info to construct interference graph

3. **Graph coloring**: attempt to color the interference graph, **spilling** variables onto the stack if no solution can be found

More complicated in practice because registers are not all treated the same (argument/return registers, caller/callee-save, shift instructions)

Liveness Analysis

One of the most fundamental analyses a compiler performs is **liveness analysis**.

Used to determine at each program point which variables are **live,** i.e., which variables' values need to be available at runtime.

Example uses:

- Lambda lifting: the arguments that **need** to be added in lambda lifting are exactly the **live** ones

- Register allocation: only need to store all of the **live** variables in registers/the stack. Means we can re-use space when a variable is no longer live.

- Function calls: only need to save the values of caller-save registers if the value is **live**

Liveness Analysis

Semantic definition:

a variable **x** is live in a block **b** (or expression, operation, etc) if the observable behavior of **b** depends on the value of **x.**

Can be done for ASTs or SSA blocks

- ASTs: determining what values are captured for lambda lifting/ closure conversion

- SSA: determining interference for register allocation

Liveness Examples

is x live?

Liveness Examples

is x live?

x

Yes

Liveness Example

is x live?

```
x * y
```

Yes

Liveness Example

is x live?

```
if b: x else: y
```

if b is ever true, yes

otherwise no

Liveness Example

is x live?

```
let b = true in
if b: x else: y
```

yes

Liveness Example

is x live?

```
let b = false in
if b: x else: y
```

no

Liveness Example

is x live?

```
let b = read_input() in
if b: x else: y
```

yes

Liveness Example

is x live?

```
let b = complex_fn() in
if b: x else: y
```

if complex_fn ever returns true: yes, otherwise: no

Limitation: Computability

Determining correct liveness information requires determining the possible values produced by arbitrary functions...

Rice's Theorem:

Any non-trivial semantic property of programs in a Turing-complete language is undecidable

Determining liveness of variables is undecidable!

Limitation: Computability

Determining **correct** liveness information can be arbitrarily complicated...

What if we determined **incorrect** liveness information sometimes?

- false positives: sometimes we say a variable is live when it's not
- false negatives: sometimes we say a variable is not live when it is

False positives are ok: we will just use more registers/space than necessary

Limitation: Computability

Goal: **Overapproximate**

The output of our liveness analysis should include every variable that is live, but possibly some that are not live.

Approach so far in class: Use scope as our liveness analysis

- This is an overapproximation: a variable can't be live if it's not in scope

We can do much better

- Only consider variables live if they actually get used
- But consider all execution paths (i.e. branches) to be possible

Liveness Analysis: Specification

Define a function LIVE : Expression -> Set(Variable)

- LIVE(x) = { x }
- LIVE(n) = { }
- LIVE(Prim(op, [imm1,...])) = LIVE(imm1) U ...
- LIVE(call(f; [imm1,...])) = LIVE(imm1) U ...
- LIVE(x = op in b) = (LIVE(b) - x) U LIVE(op)
- LIVE(br f(imm,...)) = (LIVE(f.body) - f.args) U LIVE(imm1) U ...
- LIVE(cbr imm: f else: g) = LIVE(imm) U LIVE(f.body) U LIVE(g.body)
- LIVE(ret imm) = LIVE(imm)

Liveness Analysis: Specification

Our definition of liveness is **recursive** because our blocks are recursive:

- if a block f includes a branch to itself, the live variables in f will depend on the live variables in f...

- if multiple blocks mutually recursively branch to each other, we have the same issue.

This means our specification is not well-formed.

Solution: we want the **minimal** solution to our recursive equations.

To implement this, we **initialize** all blocks to have 0 live variables, and **iteratively** improve this information, using the previous round's information each time.

An example of a general process called **dataflow analysis**, more on this later

Liveness Analysis: Example

```
f(x,y,z):
  loop(i,a):
    thn():
      r = a * z
      ret r
    els():
      i' = i - 1
  B   a' = a + x
      br loop(i', a')
    b = i == 0
    cbr b thn() els()
  br loop(y, 0)
```

In the sub-expression B, which variables are

In scope:

Syntactically occurring:

Live:

Liveness Analysis: Example

```
f(x,y,z):
  loop(i,a):
    thn():
      r = a * z
      ret r
    els():
      i' = i - 1
B     a' = a + x
      br loop(i', a')
    b = i == 0
    cbr b thn() els()
  br loop(y, 0)
```

In the sub-expression B, which variables are

In scope: `x,y,z,i,a,i'`

Syntactically occurring: `x,a,a',i'`

Live: `x,`**`z`**`,a,i'`

Liveness Analysis: Example

```
f(x,y,z):
  loop(i,a):
    thn():
      r = a * z
      ret r
    els():
      i' = i - 1
B     a' = a + x
      br loop(i', a')
    b = i == 0
    cbr b thn() els()
  br loop(y, 0)
```

In the sub-expression B, which variables are

In scope: x,y,z,i,a,i'

Syntactically occurring: x,a,a',i'

Live: x,**z**,a,i'

Liveness Analysis: Example

```
f(x,y,z):
 1loop(i,a):
   5thn():
     6r = a * z
     7ret r
   8els():
     9i' = i - 1
    10a' = a + x
    11br loop(i', a')
   3b = i == 0
   4cbr b thn() els()
 2br loop(y, 0)
```

	Round 0	Round 1
1:	{ }	?
2:	{ }	?
3:	{ }	?
4:	{ }	?
5/6:	{ }	?
7:	{ }	?
8/9:	{ }	?
10:	{ }	?
11:	{ }	?

Liveness Analysis: Example

```
f(x,y,z):
 1loop(i,a):
   5thn():
     6r = a * z
     7ret r
   8els():
     9i' = i - 1
    10a' = a + x
     11br loop(i', a')
   3b = i == 0
   4cbr b thn() els()
 2br loop(y, 0)
```

	Round 0	Round 1
1:	{ }	{x,z}
2:	{ }	{y}
3:	{ }	{a,i,x,z}
4:	{ }	{a,b,i,x,z}
5/6:	{ }	{a,z}
7:	{ }	{r}
8/9:	{ }	{a,i,x}
10:	{ }	{a,i',x}
11:	{ }	{a',i'}

Liveness Analysis: Example

```
f(x,y,z):
 1 loop(i,a):
    5 thn():
       6 r = a * z
       7 ret r
    8 els():
       9 i' = i - 1
      10 a' = a + x
      11 br loop(i', a')
    3 b = i == 0
    4 cbr b thn() els()
 2 br loop(y, 0)
```

	Round 1	Round 2
1:	{x,z}	?
2:	{y}	?
3:	{a,i,x,z}	?
4:	{a,b,i,x,z}	?
5/6:	{a,z}	?
7:	{r}	?
8/9:	{a,i,x}	?
10:	{a,i',x}	?
11:	{a',i'}	?

Liveness Analysis: Example

```
f(x,y,z):
 1loop(i,a):
   5thn():
     6r = a * z
     7ret r
   8els():
     9i' = i - 1
    10a' = a + x
     11br loop(i', a')
   3b = i == 0
   4cbr b thn() els()
 2br loop(y, 0)
```

	Round 1	Round 2
1:	{x,z}	{x,z}
2:	{y}	{x,y,z}
3:	{a,i,x,z}	{a,i,x,z}
4:	{a,b,i,x,z}	{a,b,i,x,z}
5/6:	{a,z}	{a,z}
7:	{r}	{r}
8/9:	{a,i,x}	{a,i,x,z}
10:	{a,i',x}	{a,i',x,z}
11:	{a',i'}	{a',i',x,z}

Liveness Analysis: Example

```
f(x,y,z):
 1loop(i,a):
   5thn():
     6r = a * z
     7ret r
   8els():
     9i' = i - 1
     10a' = a + x
     11br loop(i', a')
   3b = i == 0
   4cbr b thn() els()
 2br loop(y, 0)
```

	Round 2	Round 3
1:	{x,z}	?
2:	{x,y,z}	?
3:	{a,i,x,z}	?
4:	{a,b,i,x,z}	?
5/6:	{a,z}	?
7:	{r}	?
8/9:	{a,i,x,z}	?
10:	{a,i',x,z}	?
11:	{a',i',x,z}	?

Liveness Analysis: Example

```
f(x,y,z):
 1 loop(i,a):
   5 thn():
     6 r = a * z
     7 ret r
   8 els():
     9 i' = i - 1
     10 a' = a + x
     11 br loop(i', a')
   3 b = i == 0
   4 cbr b thn() els()
 2 br loop(y, 0)
```

	Round 2	Round 3
1:	{x,z}	{x,z}
2:	{x,y,z}	{x,y,z}
3:	{a,i,x,z}	{a,i,x,z}
4:	{a,b,i,x,z}	{a,b,i,x,z}
5/6:	{a,z}	{a,z}
7:	{r}	{r}
8/9:	{a,i,x,z}	{a,i,x,z}
10:	{a,i',x,z}	{a,i',x,z}
11:	{a',i',x,z}	{a',i',x,z}

Liveness Analysis: Example

```
f(x,y,z):
 1loop(i,a):
   5thn():
     6r = a * z
     7ret r
   8els():
     9i' = i - 1
 B  10a' = a + x
    11br loop(i', a')
   3b = i == 0
   4cbr b thn() els()
 2br loop(y, 0)
```

1:	{x,z}
2:	{x,y,z}
3:	{a,i,x,z}
4:	{a,b,i,x,z}
5/6:	{a,z}
7:	{r}
8/9:	{a,i,x,z}
10:	{a,i',x,z}
11:	{a',i',x,z}

In the sub-expression B, which variables are

In scope: x,y,z,i,a,i'

Syntactically occurring: x,a,a',i'

Live: x,**z**,a,i'

Implementation Concerns

How to store live sets?

- Add annotation metadata to the SSA AST
  - init_liveness(e: BB<T>) -> BB<HashSet<String>>
  - update_liveness(e: BB<HashSet<String>>) -> BB<HashSet<String>>
- iterate until you reach a fixed point
  - update_liveness(b) == b

Conflict Analysis

Once we know when we need the value of each variable, we determine which variables cannot be assigned the same register

2 variables truly conflict when

- They are live at the same time
- with different values

Err on the side of *too many* conflicts.

Conflict Analysis

Simple approach:

- Initialize the graph with all variables in the program
- Add a clique of edges for every live set

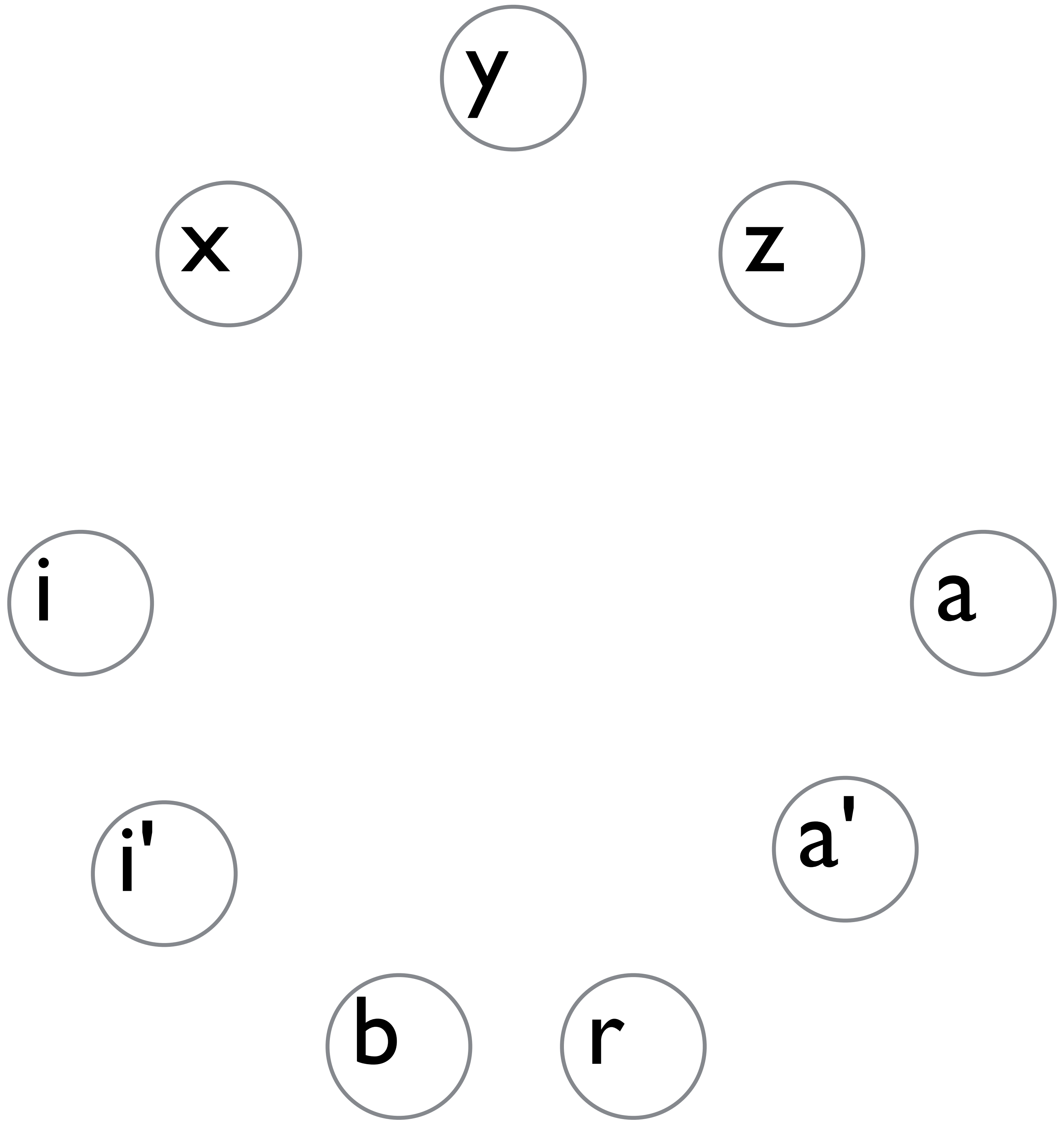
This is an overapproximation to true conflicts

```
f(y):
  y = x
  z = x + y
  ret z
```

Conflict Analysis

```
f(x,y,z):
 1loop(i,a):
   5thn():
     6r = a * z
     7ret r
   8els():
     9i' = i - 1
    10a' = a + x
     11br loop(i', a')
   3b = i == 0
   4cbr b thn() els()
 2br loop(y, 0)
```

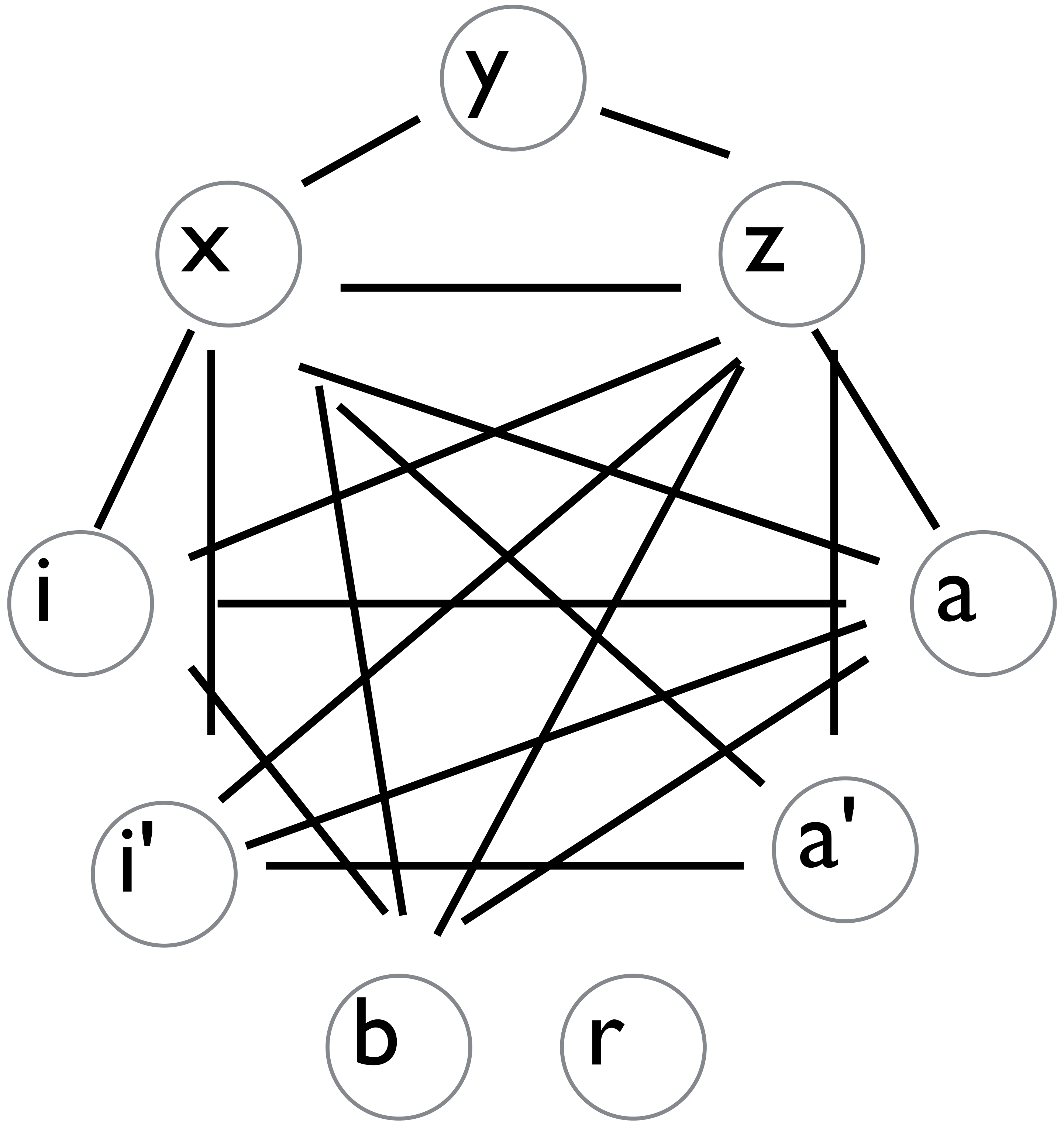
1:	{x,z}
2:	{x,y,z}
3:	{a,i,x,z}
4:	{a,b,i,x,z}
5/6:	{a,z}
7:	{r}
8/9:	{a,i,x,z}
10:	{a,i',x,z}
11:	{a',i',x,z}



Conflict Analysis

```
f(x,y,z):
 1loop(i,a):
   5thn():
     6r = a * z
     7ret r
   8els():
     9i' = i - 1
    10a' = a + x
     11br loop(i', a')
   3b = i == 0
   4cbr b thn() els()
 2br loop(y, 0)
```

1:	{x,z}
2:	{x,y,z}
3:	{a,i,x,z}
4:	{a,b,i,x,z}
5/6:	{a,z}
7:	{r}
8/9:	{a,i,x,z}
10:	{a,i',x,z}
11:	{a',i',x,z}



Summary so Far

For each top level function in the program

1. Liveness Analysis determines at every program point what variables are live
2. Conflict Analysis produces a conflict graph whose nodes are variables and edges are conflicts (the variables cannot share a register)
3. Next time: Use this conflict graph to assign registers to variables, and generate more efficient code