What about memory operations (globals, heap values) ?
- Need to model dependences, changes in values.
- SSA doesn’t help directly.

Dependences (order of operations):

\[
\begin{align*}
g.a &= 0; \\
g.b &= 1; \\
g.a &= g.b + 2; \\
g.b &= g.a + 3;
\end{align*}
\]

Last two statements must be done in order due to:
- flow dependence (write-before-read) on g.a
- anti-dependence (read-before-write) on g.b

First and third statements must be done in order due to:
- output dependence (write-before-write) on g.a
**Using Alias Information**

When can we do common subexpression elimination to save loads?

```java
  g.a = 10;
  h.a = 20;
  if (g.a == 10) // can only avoid if g and h don’t alias.
      ...
  g.b = 20;
  if (g.a == 10) // can always avoid load here given type info
      ...
```
Good example of complete system based on SSA.

To cope with memory operations, they add explicit "threading" store variables.

```c
int method (int a[], int b[]) {
    arr_store(a,0,10);
    arr_store(b,0,20);
    return arr_fetch(a,0);}
```

becomes

```c
(int,Store) method (int a[], int b[], Store S0) {
    S1 = arr_store(a,0,10,S0);
    S2 = arr_store(b,0,20,S1);
    return (arr_fetch(a,0,S2),S2);}
```

where \(S0, S1, S2\) are pseudo-values representing the global store.

Can now continue to use congruence testing to detect redundant computations.
Helps improve understanding of dependence between memory operations.

In last example, a and b might be the same array, e.g., called as method(c,c).

Simplest form of alias analysis just uses types:

```c
int method (int a[], short b[]) {
    arr_store(a,0,10);
    arr_store(b,0,20);
    return arr_fetch(a,0); }
```

Now know a and b cannot be aliased to the same array.
A more sophisticated analysis (requiring dataflow analysis) tracks creation points:

```java
int method() {
    int a[] = new a[10];
    int b[] = new b[10];
    arr_store(a,0,10);
    arr_store(b,0,20);
    return arr_fetch(a,0); }
```

Once again, `a` and `b` cannot be aliased to the same array, even though they have the same type.
Can represent the results of this analysis by changing the store argument dependencies:

\[
\begin{align*}
S1 &= \text{arr\_store}(a,0,10,S0); \\
S2 &= \text{arr\_store}(b,0,20,S0); \quad // \text{not } S1! \\
S3 &= \text{phi}(S1,S2); \\
\text{return } (\text{arr\_fetch}(a,0,S1), \quad // \text{not } S2! \\
&\quad S3); \\
\end{align*}
\]

But this doesn’t scale well.
Aliasing problems arise:

- in heap for Java
- more broadly in call-by-reference languages
- everywhere in C!

Divide memory pointers into alias classes that are guaranteed not to alias with each other.

Can use:
- types
- field names
- known objects

Alias analysis interacts with:
- class analysis (enhance type analysis to use knowledge about Java class hierarchy)
- escape analysis (determine which values can out-live the function that created them)