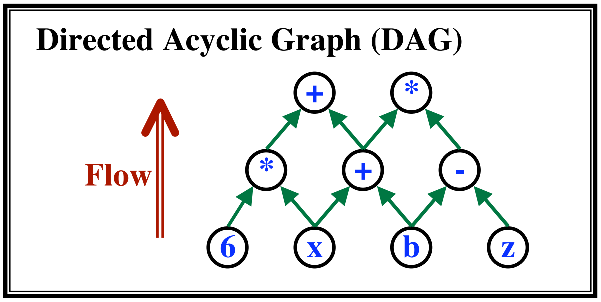
DAG-Based Optimization

of IR Code in a Basic Block



Look at one Basic Block at a time

```
<x,v,w> := f(x,b,z)
```

Construct a DAG from the IR.

Generate code from the DAG.

- Generate IR Code
- Generate Target Code

Leaves

Represent initial values on entry to the block

- Variables
- Constants

Interior Nodes

Labelled by operators

Also:

Each interior node may have an attached list of variable names

Example

Source Code:

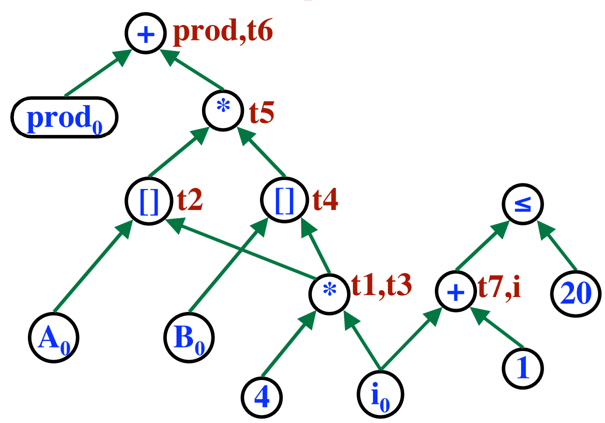
```
repeat
  prod := prod + A[i] * B[i];
  i := i + 1;
until i > 20;
```

Assume each array element is 4 bytes

IR:

```
t1  := 4 * i
t2  := A[t1]
t3  := 4 * i
t4  := B[t4]
t5  := t2 * t4
t6  := prod + t5
prod := t6
t7  := i + 1
i  := t7
if i <= 20 goto BB9
```

- Go through IR instructions.
- For each operation construct a new node.
- Label each node.
- Re-use existing nodes, when possible.



Mappings

Functions:

- Domain
- Range

Supply an element from the domain...

The function returns an element from the range.

Mappings

Functions:

- Domain
- Range

Supply an element from the domain...

The function returns an element from the range.

Definition: A "Mapping"
A data structure that implements a function.
Can be updated.

Mappings

Functions:

- Domain
- Range

Supply an element from the domain...

The function returns an element from the range.

Definition: A "Mapping"
A data structure that implements a function.
Can be updated.

Examples:

A mapping from Strings to Integers. (e.g., a phone book)
A mapping from Variables to VarDecls (e.g., a symbol table)

Mappings

Functions:

- Domain
- Range

Supply an element from the domain...
The function returns an element from the range.

Definition: A "Mapping"
A data structure that implements a function.
Can be updated.

Examples:

A mapping from Strings to Integers. (e.g., a phone book)
A mapping from Variables to VarDecls (e.g., a symbol table)

Basic Operations:

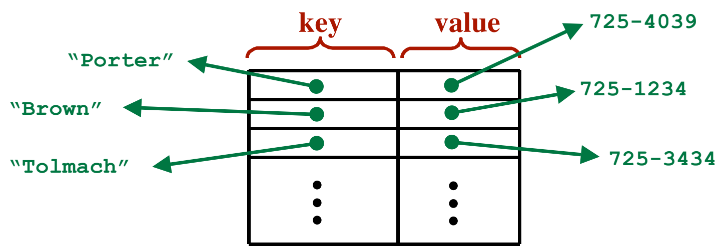
Lookup (key) → value
AddEntry (key, value)
DeleteEntry (key)
...etc...

Visual Representations

key	value
"Porter"	725-4039
"Brown"	725-1234
"Tolmach"	725-3434
"Fant"	725-7654
"Antoy"	725-4050
"Mocas"	725-8899

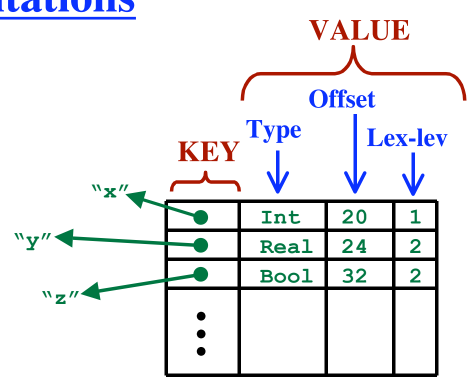
Visual Representations

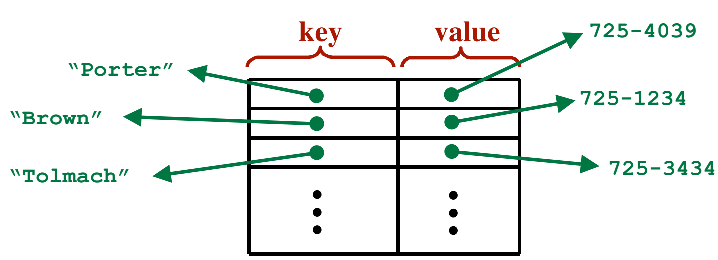
key	value
"Porter"	725-4039
"Brown"	725-1234
"Tolmach"	725-3434
"Fant"	725-7654
"Antoy"	725-4050
"Mocas"	725-8899



Visual Representations

key	value
"Porter"	725-4039
"Brown"	725-1234
"Tolmach"	725-3434
"Fant"	725-7654
"Antoy"	725-4050
"Mocas"	725-8899





Implementation

A Mapping from small Integers to ...

Use an Array

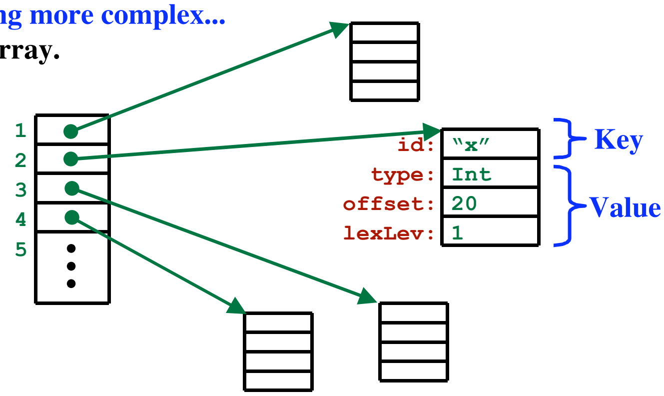
Implementation

A Mapping from small Integers to ...

Use an Array

If the key is something more complex...

Can still use an array.



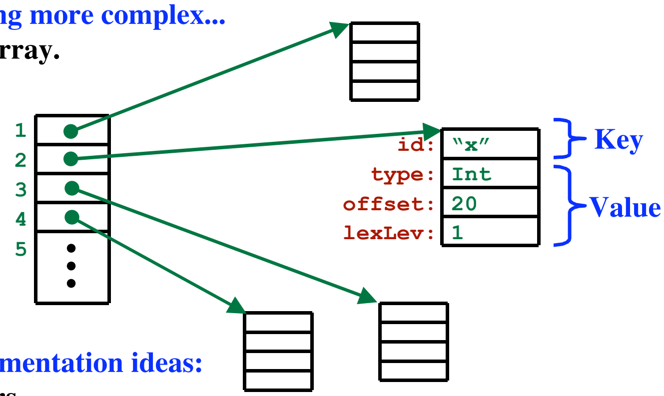
Implementation

A Mapping from small Integers to ...

Use an Array

If the key is something more complex...

Can still use an array.



More complex implementation ideas:

- Objects, Pointers
- Linked Lists
- Arrays
- Binary Trees
- Hash Tables

Basic Operations:

Lookup (key) → value
AddEntry (key, value)
DeleteEntry (key)
...etc...

Building the DAG

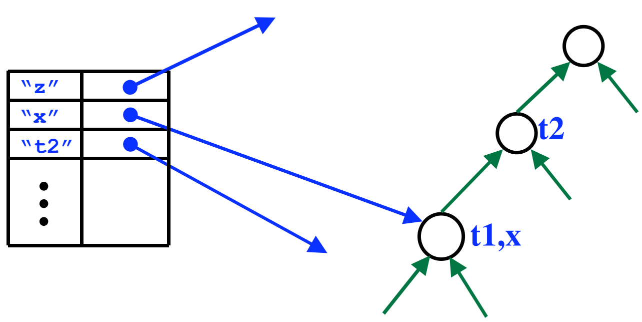
Need a mapping

Call it "CurrentNode"

FROM: Variable Names

TO: Nodes in the DAG

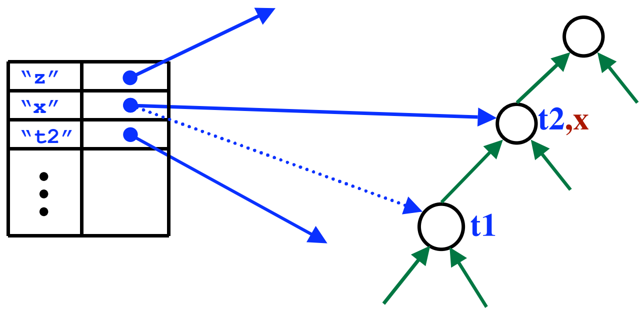
CurrentNode (x) points to the node currently labelled with "x".



Building the DAG

Need a mapping
Call it "CurrentNode"
FROM: Variable Names
TO: Nodes in the DAG

CurrentNode (x) points to the node currently labelled with "x".



Algorithm to Construct the DAG

Go through the Basic Block (in order)
For each IR in the block...

Add to the growing DAG...
Assume we have a binary IR instruction, such as

x := y $\oplus$ z

If CurrentNode(y) is undefined...
Create a leaf named "y_0".
Set CurrentNode(y) to point to it.
If CurrentNode(z) is undefined...
<same>
Look for a node labelled "$\oplus$"
with left child = CurrentNode(y)
and right child = CurrentNode(z)
(If none found, then create one.)
Call this node N.
Delete x from the list of ID's attached
to CurrentNode(x).
Add x to the list of ID's attached to N.
Set CurrentNode(x) to point to N.

Algorithm to Construct the DAG

If we have a unary operation, such as

```
    x := -y

If CurrentNode(y) is undefined...
    Create a leaf named "y0".
    Set CurrentNode(y) to point to it.
Look for a node labelled "-"
    with child = CurrentNode(y)
         (If none found, then create one.)
Call this node N.
Delete x from the list of ID's attached
            to CurrentNode(x).
Add x to the list of ID's attached to N.
Set CurrentNode(x) to point to N.
```


Algorithm to Construct the DAG

If we have a copy operation

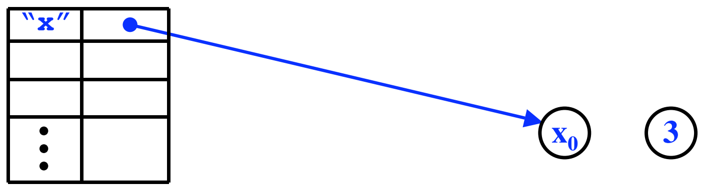
```
    x := y

If CurrentNode(y) is undefined...
    Create a leaf named "y0".
    Set CurrentNode(y) to point to it.
Let N = CurrentNode(y)
Delete x from the list of ID's attached
            to CurrentNode(x).
Add x to the list of ID's attached to N.
Set CurrentNode(x) to point to N.
```

Example

IR Code:

```
x := x * 3
y := y + x
x := y - z
y := x
```

IR Code:

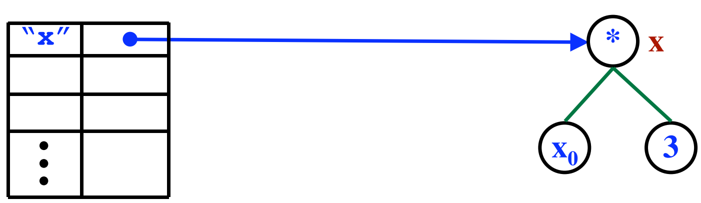


Example

IR Code:

```
x := x * 3
y := y + x
x := y - z
y := x
```

IR Code:

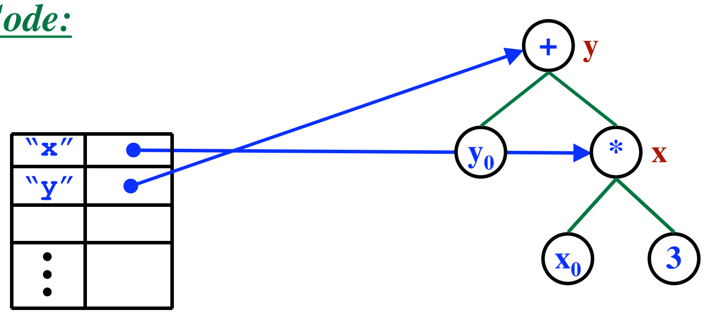


Example

IR Code:

```
x := x * 3
y := y + x
x := y - z
y := x
```

IR Code:

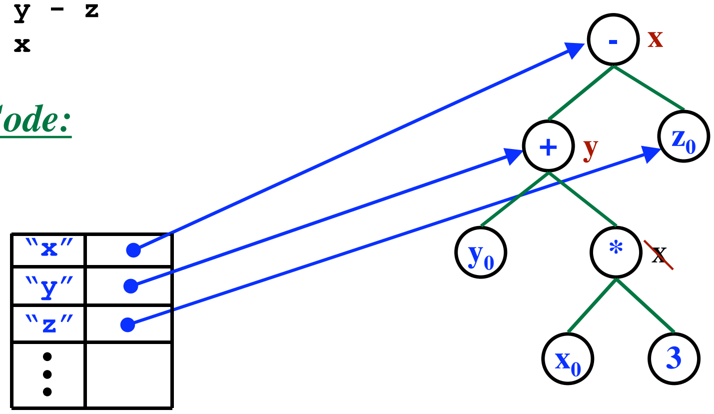


Example

IR Code:

```
x := x * 3
y := y + x
x := y - z
y := x
```

IR Code:

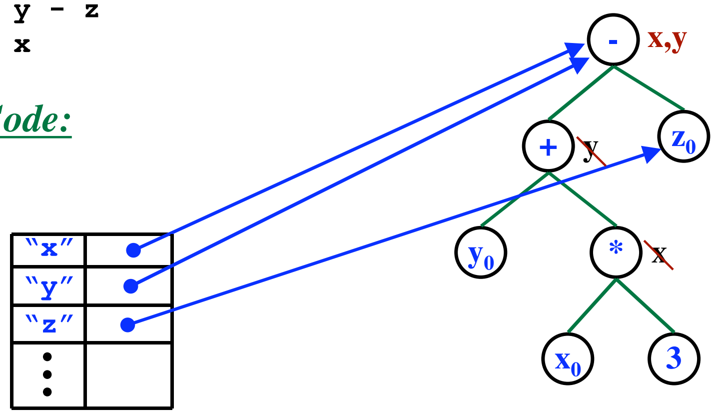


Example

IR Code:

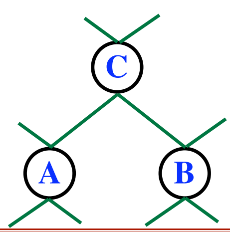
```
x := x * 3
y := y + x
x := y - z
y := x
```

IR Code:



Topological Sort

An ordering of the nodes of the DAG.
Each node must be listed after all its children.



...B...A...C...

Idea:

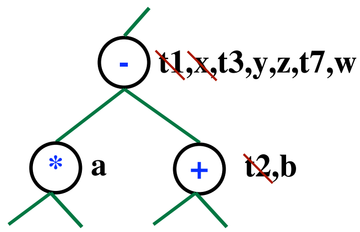
Find a topological order of nodes.
Evaluate a node after all its children have been evaluated.
...and before it is needed by its parents!

Summary:

- Build DAG
- Find topological order
- Regenerate IR instructions.

To Regenerate the IR Code

Look at each node, in topological order...

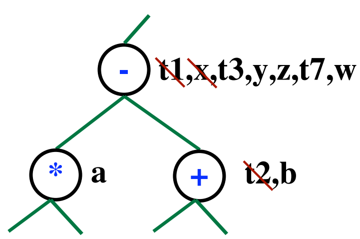



```
a := ...
b := ...
```

Some of the labels have been removed from the list.

To Regenerate the IR Code

Look at each node, in topological order...




```
a := ...
b := ...
```

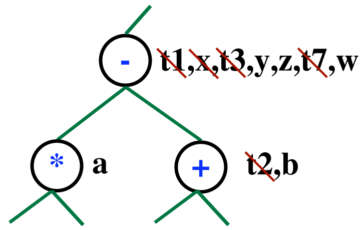
Some of the labels have been removed from the list.

Of the remaining labels
see which are LIVE at the end of the Basic Block.

Ignore the DEAD variables; select a live variable.
(If no LIVE variables, create a temp variable.)

To Regenerate the IR Code

Look at each node, in topological order...




```
a := ...
b := ...
```

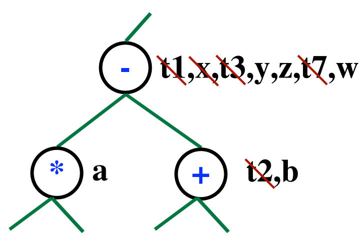
Some of the labels have been removed from the list.

Of the remaining labels
see which are LIVE at the end of the Basic Block.

Ignore the DEAD variables; select a live variable.
(If no LIVE variables, create a temp variable.)

To Regenerate the IR Code

Look at each node, in topological order...




```
a := ...
b := ...
y := a - b
```

Some of the labels have been removed from the list.

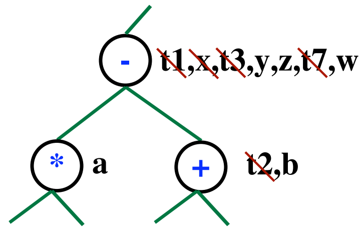
Of the remaining labels
see which are LIVE at the end of the Basic Block.

Ignore the DEAD variables; select a live variable.
(If no LIVE variables, create a temp variable.)

Generate an IR instruction for the operation.

To Regenerate the IR Code

Look at each node, in topological order...




```
a := ...
b := ...
y := a - b
z := y
w := y
```

Some of the labels have been removed from the list.

Of the remaining labels
see which are LIVE at the end of the Basic Block.

Ignore the DEAD variables; select a live variable.
(If no LIVE variables, create a temp variable.)

Generate an IR instruction for the operation.

Generate copies for any additional LIVE variables.

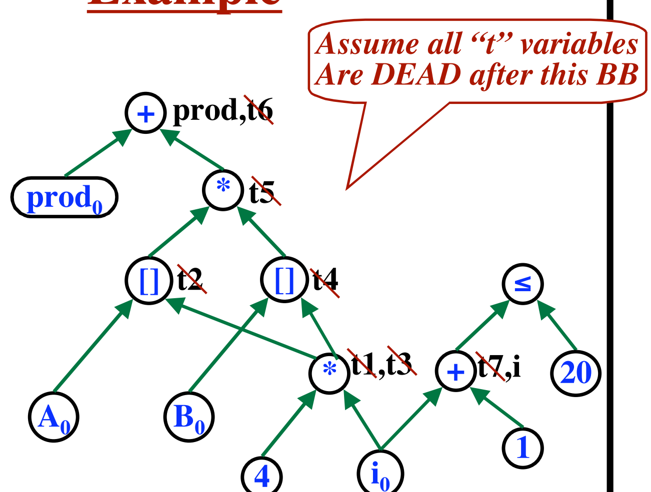
Example

Before:

```
t1 := 4 * i
t2 := A[t1]
t3 := 4 * i
t4 := B[t4]
t5 := t2 * t4
t6 := prod + t5
prod := t6
t7 := i + 1
i := t7
if i <= 20 goto BB9
```

Now:

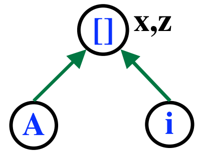
```
t1 := 4 * i
t2 := A[t1]
t4 := B[t1]
t5 := t2 * t4
prod := prod + t5
i := i + 1
if i <= 20 goto BB9
```



Problems

Assignments to Arrays

```
x := A[i]
A[j] := 43
z := A[i]
```



Will "x" and "z" be set to the same value? Possibly not!!!

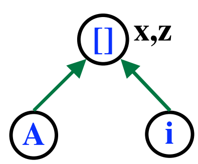
Problems

Assignments to Arrays

```
x := A[i]
A[j] := 43
z := A[i]
```

The Optimized Code:

```
x := A[i]
z := x
A[j] := 43
```



Will "x" and "z" be set to the same value? Possibly not!!!

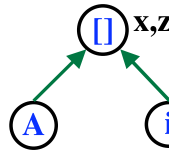
Problems

Assignments to Arrays

```
x := A[i]
A[j] := 43
z := A[i]
```

The Optimized Code:

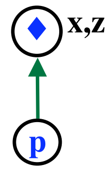
```
x := A[i]
z := x
A[j] := 43
```



Will "x" and "z" be set to the same value? Possibly not!!!

Indirect Assignments (through pointers)

```
x := *p
*q := z
z := *p
```



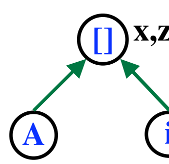
Problems

Assignments to Arrays

```
x := A[i]
A[j] := 43
z := A[i]
```

The Optimized Code:

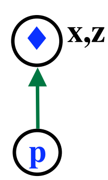
```
x := A[i]
z := x
A[j] := 43
```



Will "x" and "z" be set to the same value? Possibly not!!!

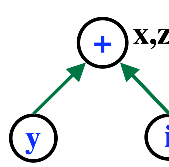
Indirect Assignments (through pointers)

```
x := *p
*q := z
z := *p
```



"Equivalenced" Names

```
x := y + i
w := 43
z := y + i
```



What of "w" is another Name for "y"???

Solution #1

Put things like

A[..] := ...

*p := ...

call ...

into their own blocks.

Solution #2

When building the DAG...

We try to re-use nodes

Look for a node labelled “+” with operands “x” and “y”...

If found, use that node.

Else, create a new node.

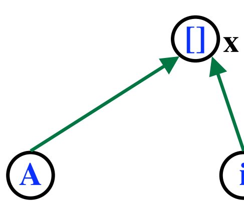
Array Accesses -- always do the fetch from the array

Pointer Indirection -- always do the fetch from memory

Also, we need to impose some order constraints.

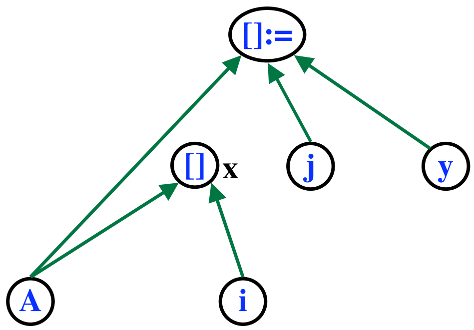
Order Restrictions

```
x := A[i]

A[j] := y

z := A[i]
```



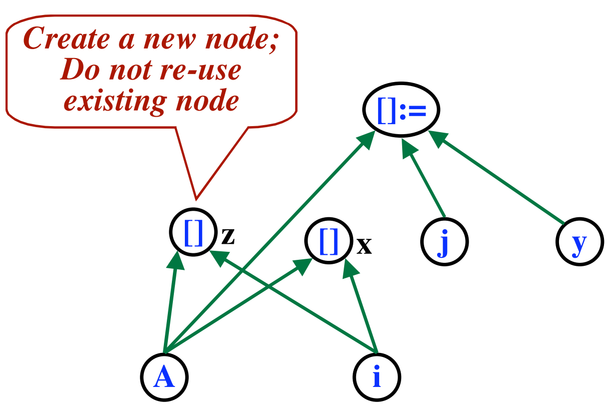
Order Restrictions

```
x := A[i]

A[j] := y

z := A[i]
```



Order Restrictions

```
x := A[i]

A[j] := y

z := A[i]
```

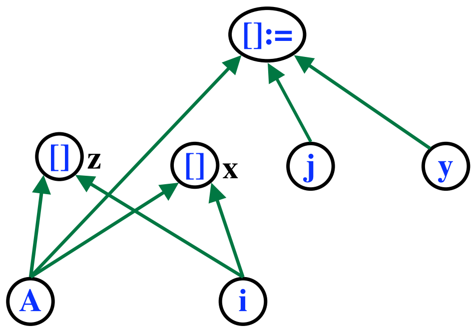


Order Restrictions

```
x := A[i]
A[j] := y
z := A[i]
```

Must follow

Must follow



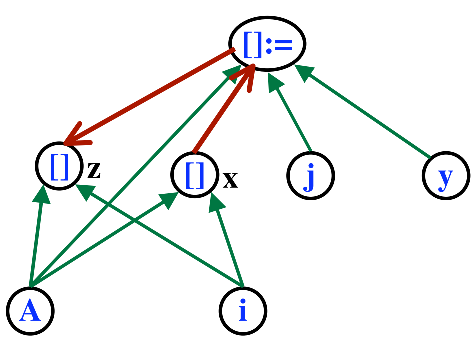
Order Restrictions

```
x := A[i]
A[j] := y
z := A[i]
```

Must follow

Must follow

Add special edges the DAG to show the order restrictions.



Order Restrictions

```
…[…]  := ...
      ↓
...  := …[…]
```

```
*p := ...
      ↓
... x ...
```

```
call …
      ↓
... x ...
```

```
...  := …[…]
      ↓
…[…]  := ...
```

```
... x ...
      ↓
*p := ...
```

```
... x ...
      ↓
call …
```