CS412/CS413

Introduction to Compilers

Tim Teitelbaum

Lecture 31: Instruction Selection

09 Apr 08

Backend Optimizations

- Instruction selection
  - Translate low-level IR to assembly instructions
  - A machine instruction may model multiple IR instructions
  - Especially applicable to CISC architectures

- Register Allocation
  - Place variables into registers
  - Avoid spilling variables on stack

Instruction Selection

- Different sets of instructions in low-level IR and in the target machine
- Instruction selection = translate low-level IR to assembly instructions on the target machine

- Straightforward solution: translate each low-level IR instruction to a sequence of machine instructions
- Example:

```
x = y + z
```

→

```
mov y, r1
mov z, r2
add r2, r1
mov r1, x
```

Instruction Selection

- Problem: straightforward translation is inefficient
  - One machine instruction may perform the computation in multiple low-level IR instructions
  - Excessive memory traffic

- Consider a machine that includes the following instructions:

add r2, r1	$r1 \leftarrow r1+r2$
mulc c, r1	$r1 \leftarrow r1*c$
load r2, r1	$r1 \leftarrow *r2$
store r2, r1	$*r1 \leftarrow r2$
movem r2, r1	$*r1 \leftarrow *r2$
movex r3, r2, r1	$*r1 \leftarrow *(r2+r3)$

Example

- Consider the computation:
  a[i+1] = b[j]
- Assume a,b, i, j are global variables
  register ra holds address of a
  register rb holds address of b
  register ri holds value of i
  register rj holds value of j

Low-level IR:

```
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3
```

Possible Translation

- Address of b[j]: mulc 4, rj
  add rj, rb
- Load value b[j]: load rb, r1
- Address of a[i+1]: add 1, ri
  mulc 4, ri
  add ri, ra
- Store into a[i+1]: store r1, ra

Low-level IR:

t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3

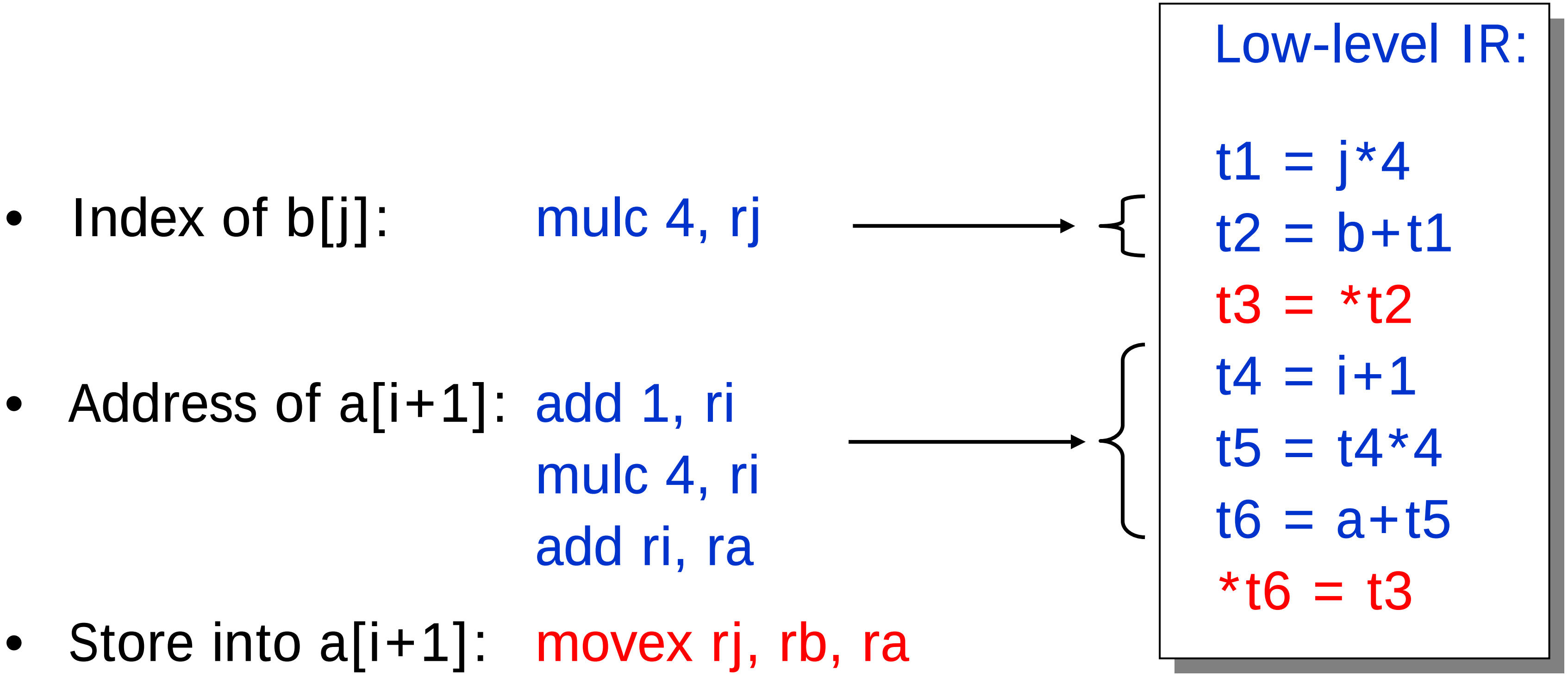
Another Translation

- Address of b[j]: mulc 4, rj
  add rj, rb
- Address of a[i+1]: add 1, ri
  mulc 4, ri
  add ri, ra
- Store into a[i+1]: movem rb, ra

Low-level IR:

```
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3
```

Yet Another Translation



- Index of b[j]: mulc 4, rj
- Address of a[i+1]: add 1, ri
  mulc 4, ri
  add ri, ra
- Store into a[i+1]: movex rj, rb, ra

Low-level IR:

```
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3
```

Issue: Instruction Costs

- Different machine instructions have different costs
  - Time cost: how fast instructions are executed
  - Space cost: how much space instructions take

- Example: cost = number of cycles

add r2, r1	cost=1
mulc c, r1	cost=10
load r2, r1	cost=3
store r2, r1	cost=3
movem r2, r1	cost=4
movex r3, r2, r1	cost=5

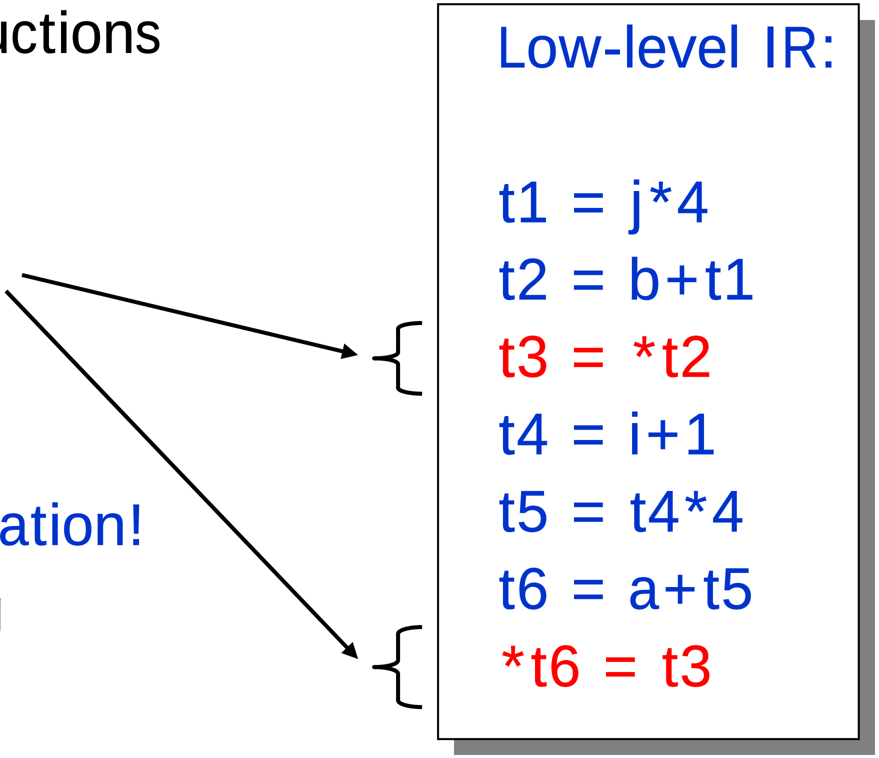
- Goal: find translation with smallest cost

How to Solve the Problem?

- Difficulty: low-level IR instruction matched by a machine instructions may not be adjacent

- Example: movem rb, ra

- Idea: use tree-like representation!
  - Easier to detect matching instructions



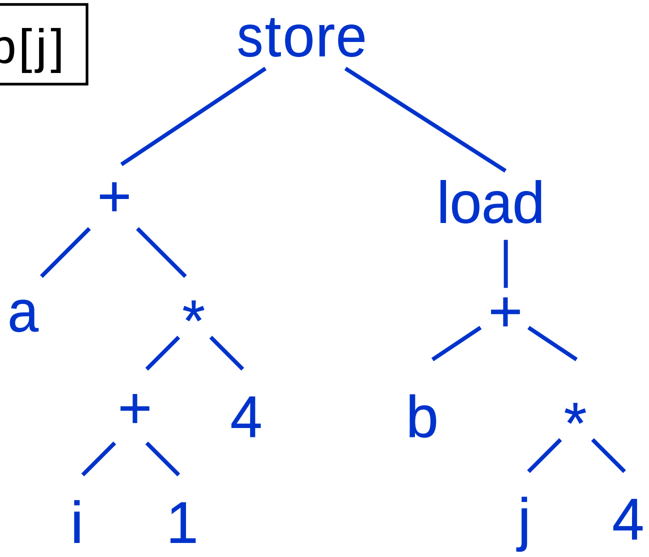
Low-level IR:

```
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3
```

Tree Representation

- Goal: determine parts of the tree that correspond to machine instructions

`a[i+1] = b[j]`



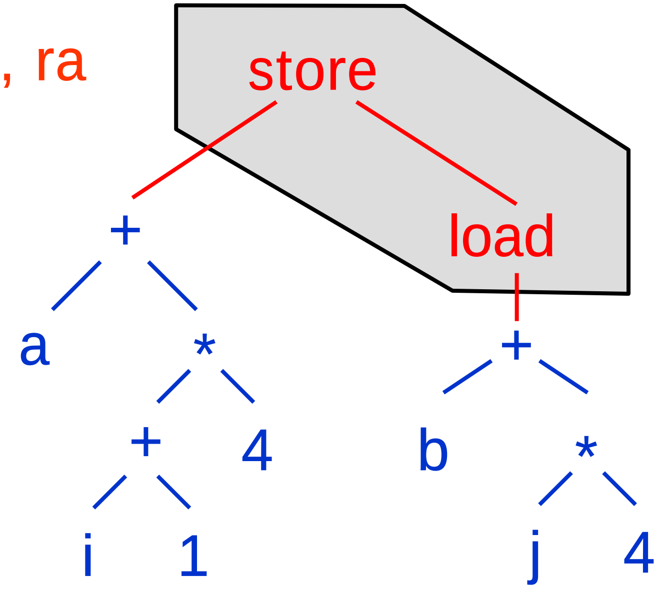
Low-level IR:

```
t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3
```

Tiles

- Tile = tree patterns (subtrees) corresponding to machine instructions

movem rb, ra

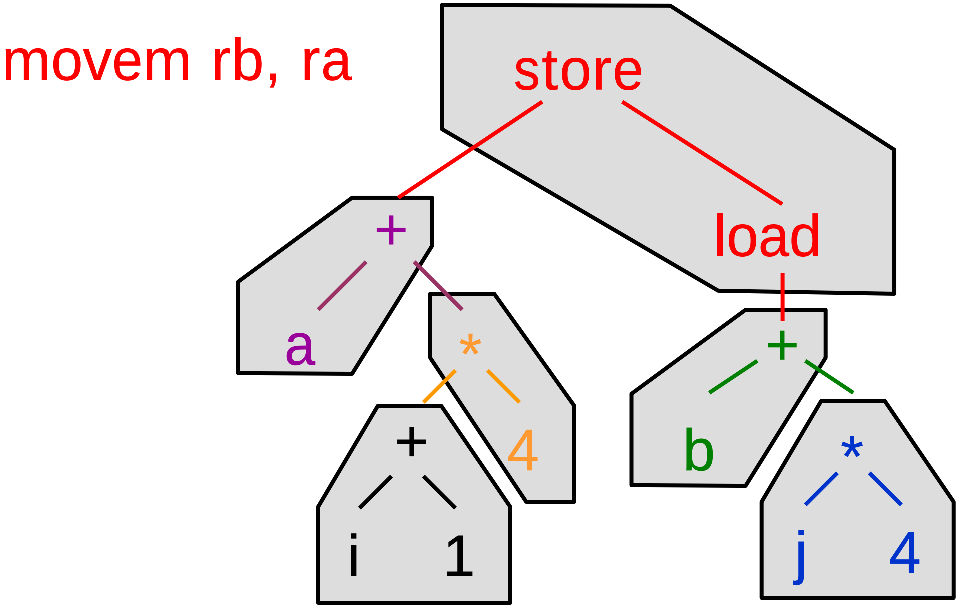


Low-level IR:

t1 = j*4
t2 = b+t1
t3 = *t2
t4 = i+1
t5 = t4*4
t6 = a+t5
*t6 = t3

Tiling

- Tiling = cover the tree with disjoint tiles



Assembly:

mulc 4, rj

add rj, rb

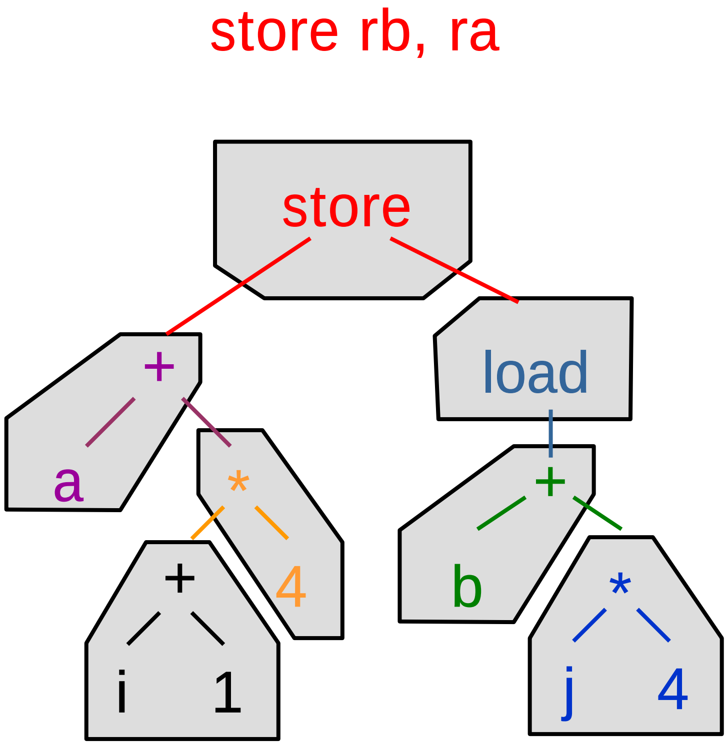
add 1, ri

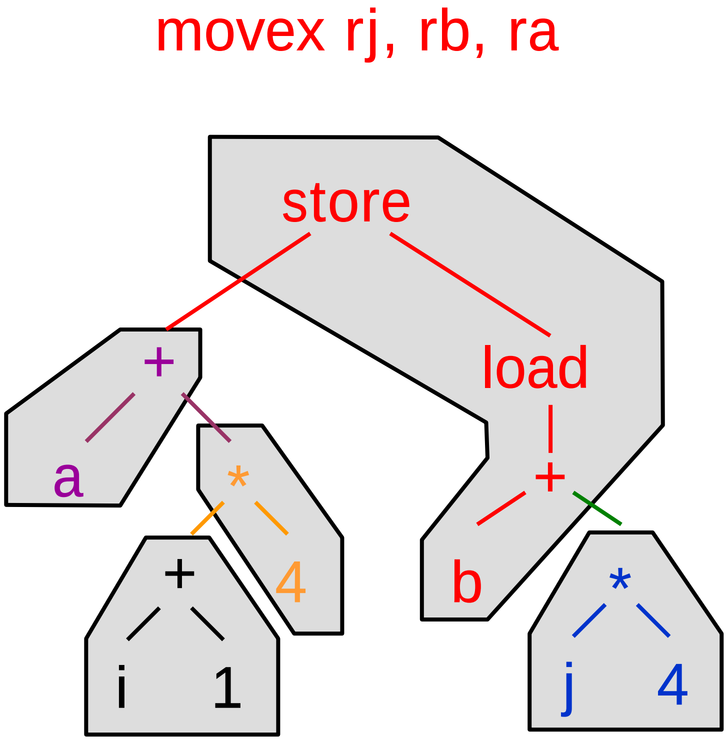
mulc 4, ri

add ri, ra

movem rb,ra

Tiling

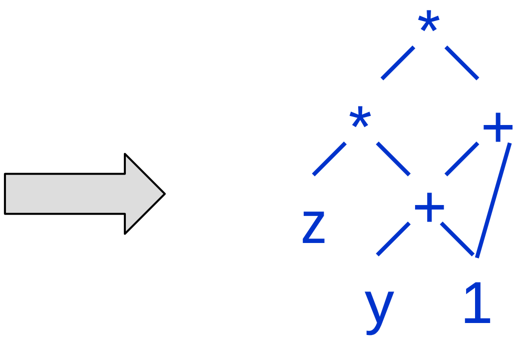

store rb, ra
store
load
+
a
*
4
+
i
1
+
b
*
j
4


movex rj, rb, ra
store
load
+
a
*
4
+
i
1
+
b
*
j
4

Directed Acyclic Graphs

- Tree representation: appropriate for instruction selection
  - Tiles = subtrees → machine instructions
- DAG = more general structure for representing instructions
  - Common sub-expressions represented by the same node
  - Tile the expression DAG
- Example:

t = y+1
y = z*t
t = t+1
z = t*y



Big Picture

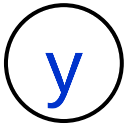
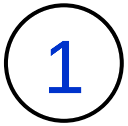
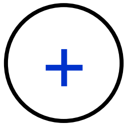
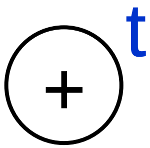
- What the compiler has to do:

  1. Translate low-level IR code into DAG representation

  2. Then find a good tiling of the DAG
     - Maximal munch algorithm
     - Dynamic programming algorithm

DAG Construction

- Input: sequence of low IR instructions in basic block
- Output: expression DAG for the block

- Idea:
  - Each node is labeled with either a variable, constant, or operator, e.g., , , or 
  - Each node is annotated with variables that hold the value, e.g., 
  - Build DAG bottom-up

DAG Construction Algorithm

```
for each instruction I in basic block in execution order
    if I has form x = y op z;
    – Find a dag node annotated y, or create one; call it $n_y$
    – Find a dag node annotated z, or create one; call it $n_z$
    – Find a dag node labeled op with operands $n_y$ and $n_z$, or create a
      one; call it $n_x$
    – Remove annotation x from any node on which it appears.
    – Add x to list of annotations for node $n_x$
    else if I has form x = y;
    – Find a dag node annotated y, or create one; call it $n_y$
    – Add x to list of annotations of node $n_y$
    else ...
```

DAG Construction Example

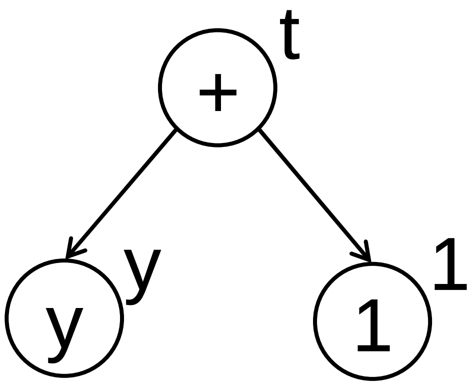
Basic block

```
t = y+1
w= y+1
y = z*t
t = t+1
z = t*y
w = z
```

DAG Construction Example

Basic block

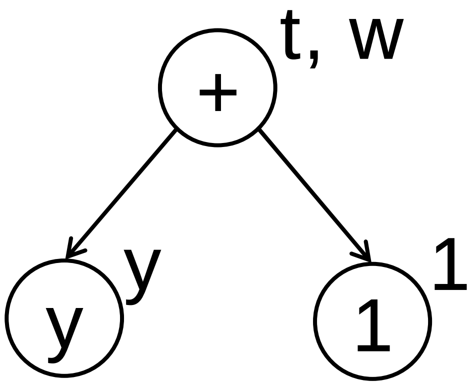
t = y+1
w= y+1
y = z*t
t = t+1
z = t*y
w = z



DAG Construction Example

Basic block

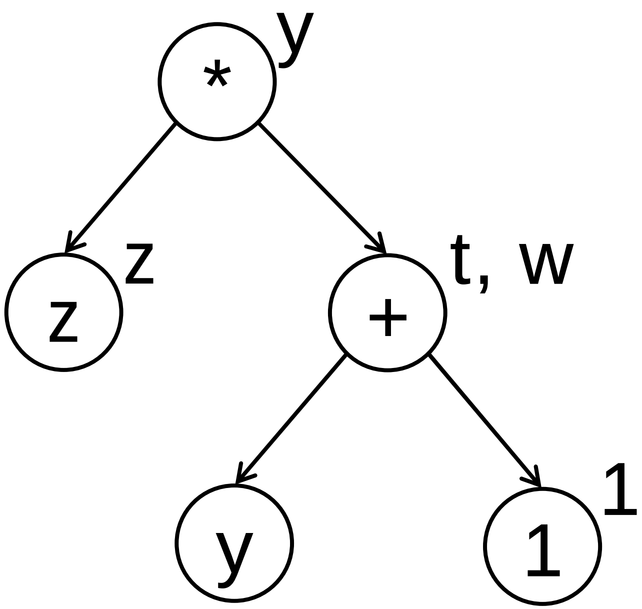
```
t = y+1
w= y+1
y = z*t
t = t+1
z = t*y
w = z
```



DAG Construction Example

Basic block

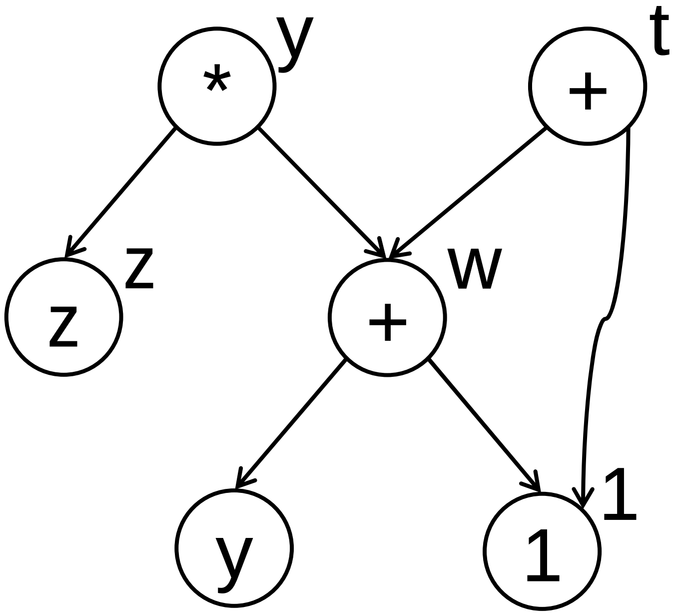
```
t = y+1
w= y+1
y = z*t
t = t+1
z = t*y
w = z
```



DAG Construction Example

Basic block

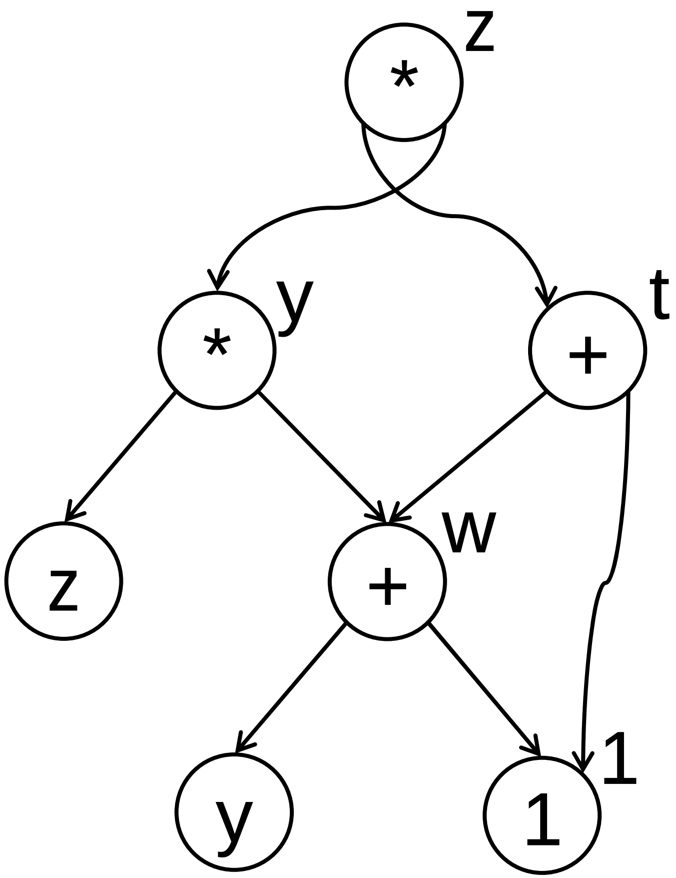
t = y+1
w= y+1
y = z*t
t = t+1
z = t*y
w = z



DAG Construction Example

Basic block

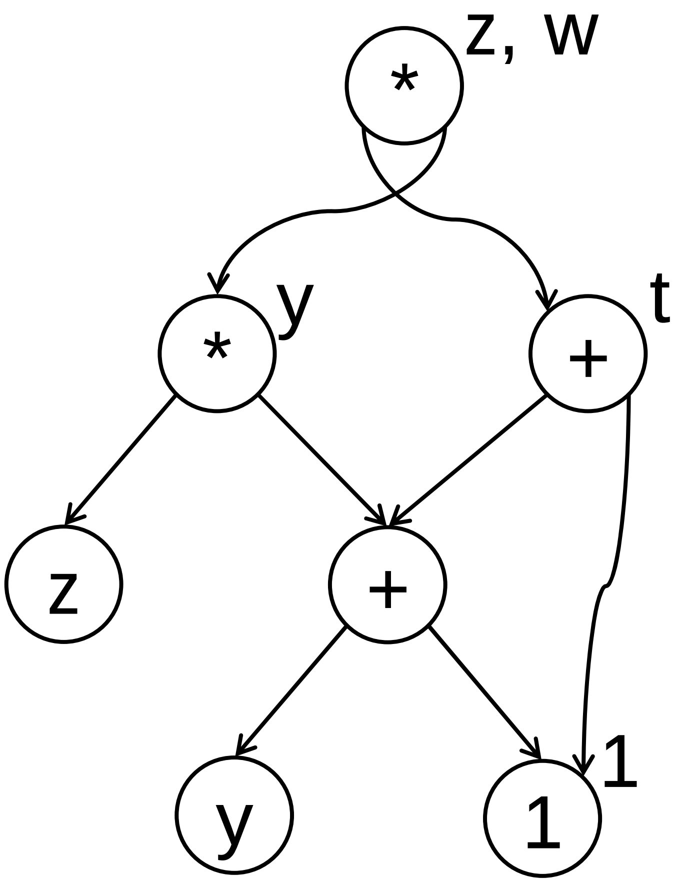
t = y+1
w= y+1
y = z*t
t = t+1
z = t*y
w = z



DAG Construction Example

Basic block

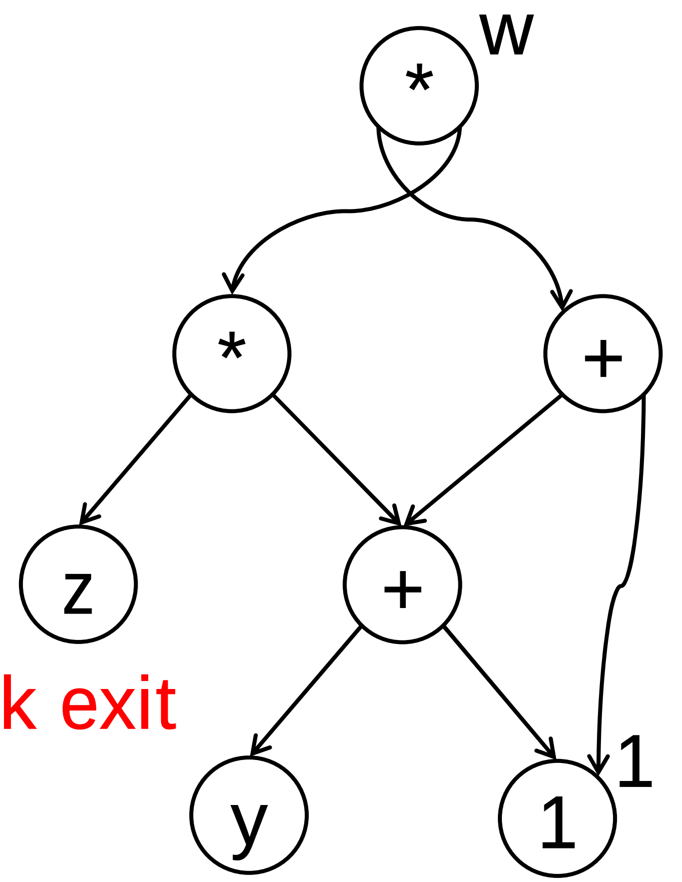
t = y+1
w= y+1
y = z*t
t = t+1
z = t*y
w = z



DAG Construction Example

Basic block

```
t = y+1
w= y+1
y = z*t
t = t+1
z = t*y
w = z
```



If only w is live at block exit