Intermediate Representations

Front End & Back End

- The portion of the compiler that does scanning, parsing and static semantic analysis is called the front-end.
- The translation and code generation portion of it is called the back-end.
- The front-end depends mainly on the source language and the back-end depends on the target architecture.

Intermediate Representation

- A compiler transforms the source program to an intermediate form that is mostly independent of the source language and the machine architecture.
- This approach isolates the front-end and the back-end[a].

[a] Every source language has its front end and every target language has its back end.

Note

- More than one intermediate representations may be used for different levels of code improvement.
- A high level intermediate form preserves source language structure. Code improvements on loop can be done on it.
- A low level intermediate form is closer to target architecture.

Tree Representations

- Parse tree is a representation of complete derivation of the input.
- It has intermediate nodes labeled with non-terminals of derivation.
- This is used (often implicitly) for parsing and attribute synthesis.

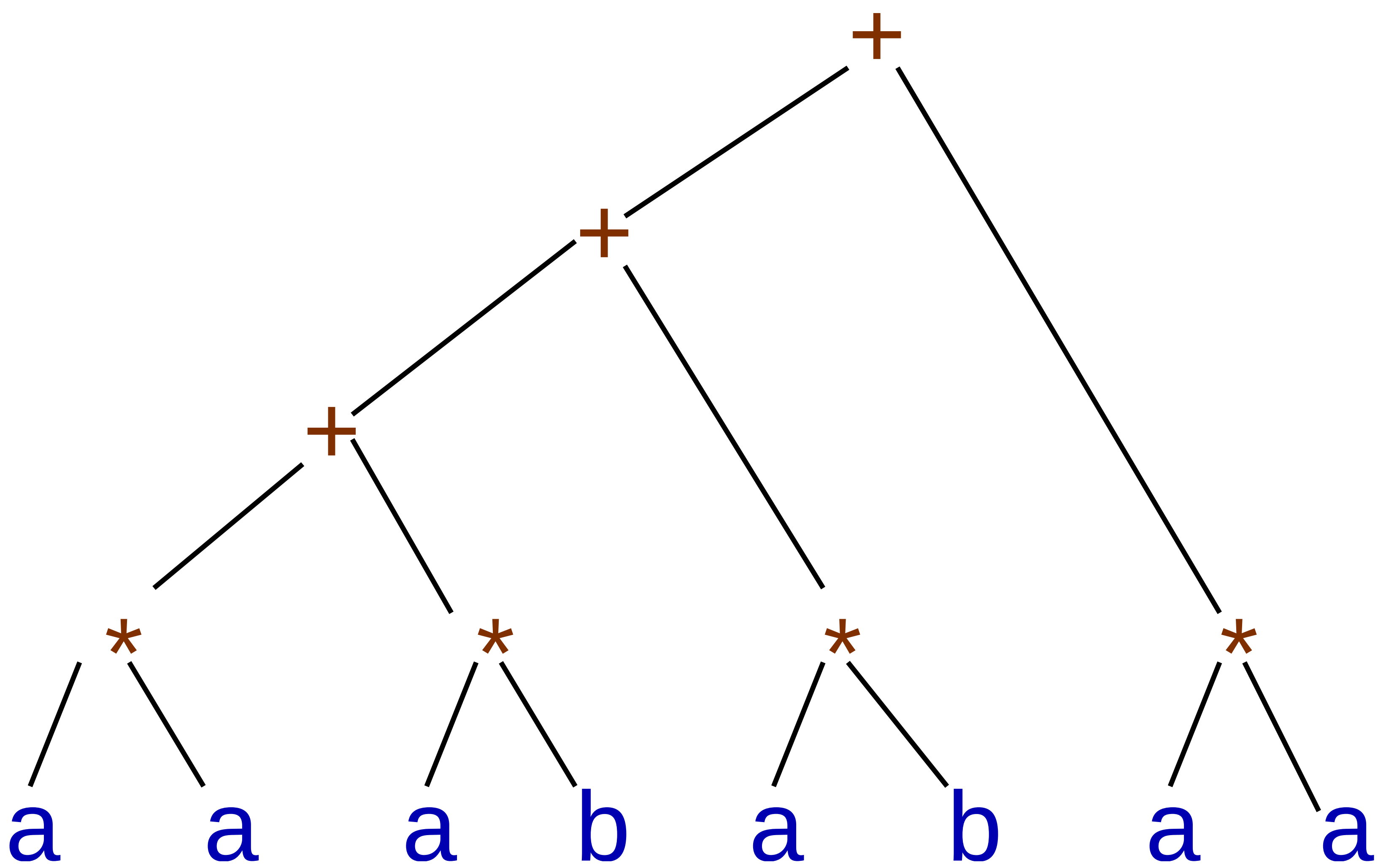
Tree Representations

- A syntax tree is very similar to a parse tree where extraneous nodes are removed.
- It is a good representation that is close to the source-language as it preserves the structure of source constructs.
- It is very useful in applications like source-to-source translation, or syntax-directed editor etc.

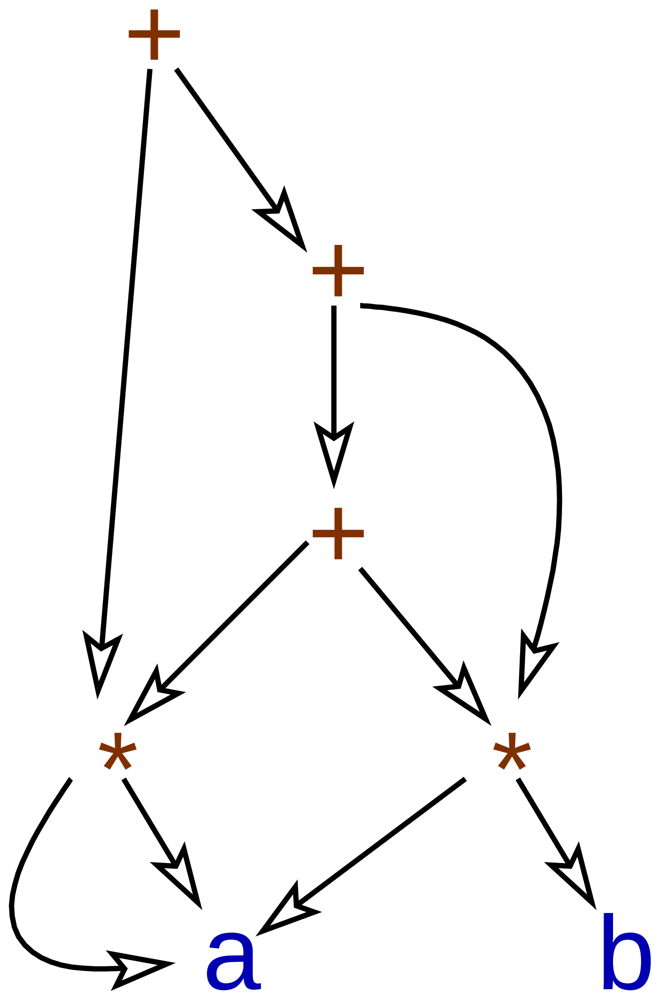
Directed Acyclic Graph (DAG)

- A directed acyclic graph (DAG) is an improvement over a syntax tree, where duplications of subtrees such as common subexpressions are identified and shared.
- This helps to identify common sub-expressions, so that the cost of evaluation can be reduced.

Syntax Tree: `a*a+a*b+a*b+a*a`



DAG: `a*a+a*b+a*b+a*a`



Note

- There are six occurrences of '`a`' and two occurrences of '`b`' in the expression.
- In the DAG '`a`' has two parents to indicate two occurrences of it in two different sub-expressions.

Note

- Similarly, '`b`' has one parent to indicate its occurrence in one sub-expression.
- The internal nodes representing '`a*a`' and '`a*b`' also has two parents each indicating their two occurrences.

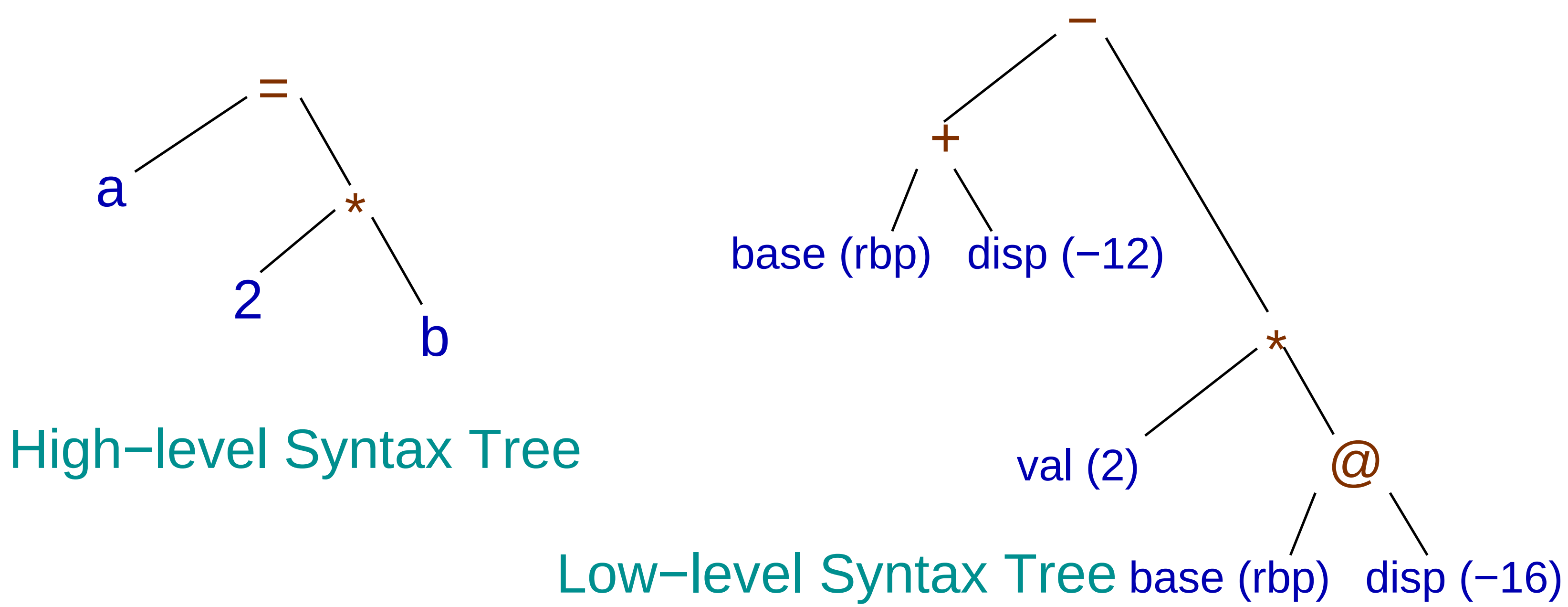
Low-Level Tree

- The tree and DAG we have discussed so far are closer to the source code.
- But they do not have the low-level details of different variables e.g. their locations, types, addressing modes, initial values etc.
- A low-level tree may contain these information for code generation and improvement.

Low-Level Tree

- Location of a variable may be specified by a memory address stored in a register and a displacement.
- There may be one or more levels of address indirection.
- An occurrence of a variable may refer to l-value or r-value.

Trees: `a = 2 * b`


=
a
*
2
b
High−level Syntax Tree
−
+
base (rbp)
disp (−12)
*
val (2)
@
base (rbp)
disp (−16)
Low−level Syntax Tree

Graph Representations

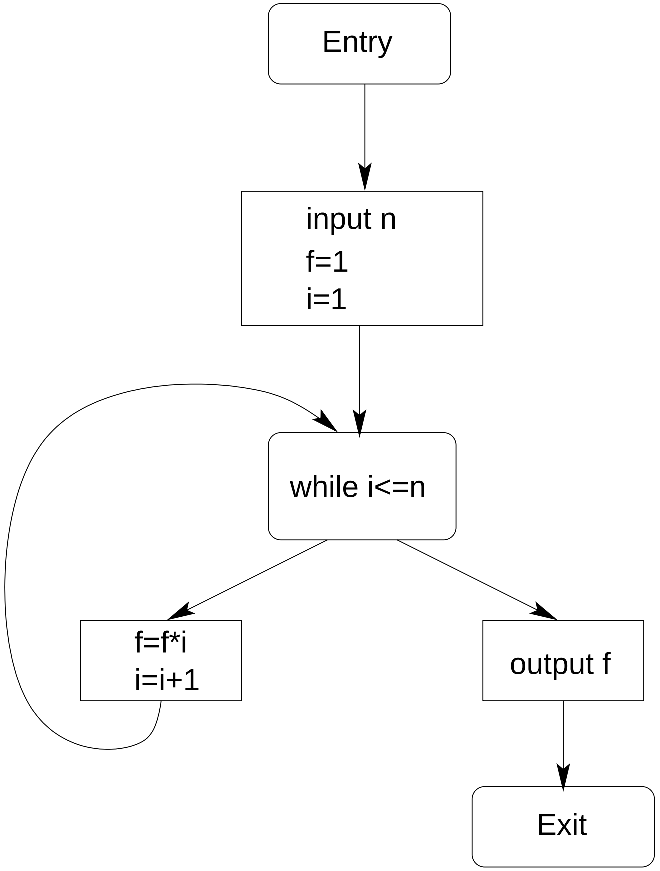
- There are different types of graph representations used to represent and analyze properties of a program.
- A control-flow graph[a] models the flow of control between the basic blocks[b].

[a]Afterward we shall define them formally.

[b]Maximal length sequence of single entry-point branch-free code.

Control-Flow Graph

```
0 input n
1 f=1
2 i=1
3 while i <= n
4       f=f*i
5       i=i+1
6 output f
```

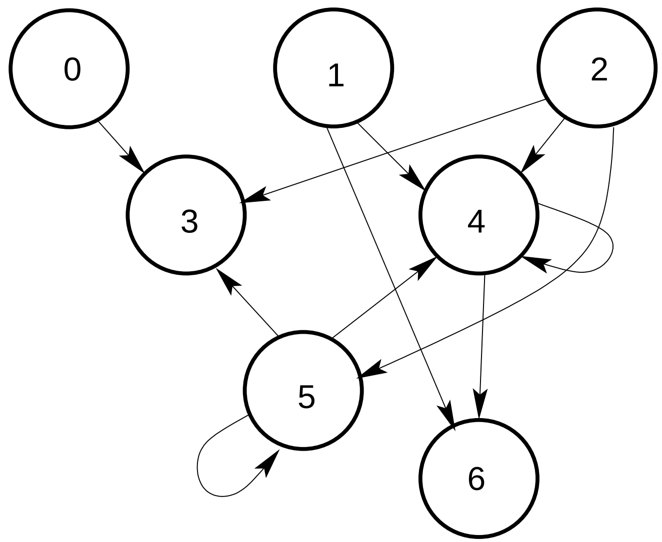


Graph Representations

- A data-dependence graph captures the definition or creation of a new data and its usage. There is are edges from the definition of a data to different points of its use.
- Call graph is used for inter-procedural analysis of code. There is an edge from each instance of call to the procedure.

Data-Dependence Graph

```
0 input n
1 f=1
2 i=1
3 while i <= n
4       f=f*i
5       i=i+1
6 output f
```



SDT for Tree and DAG

- Following are syntax directed translations to construct expression tree and DAG from the classic expression grammar G.
- We are not considering the error handling where the variable is undefined.

SDT for Tree

$F \rightarrow \texttt{id}$

```
{
 index = searchInsertSymTab(id.name) ;
 F.node = mkLeaf(index);
}
```

$E \rightarrow E_1 + T$

```
{ E.node = mkNode('+', E1.node, T.node);}
```

SDT for DAG

$F \rightarrow$ `id`

```
{
 (index, new) = searchInsertSymTab(id.name) ;
 if(new == NEW) {
    F.node = mkLeaf(index);
    symTab[index].leaf = F.node;
 }
 else F.node = symTab[index].leaf;
}
```

SDT for DAG

$E \rightarrow E_1 + T$

```
{
  node = searchNode('+',E1.node,T.node);
  if(node == NULL)
     E.node = mkNode('+',E1.node,T.node);
  else E.node = node;
}
```

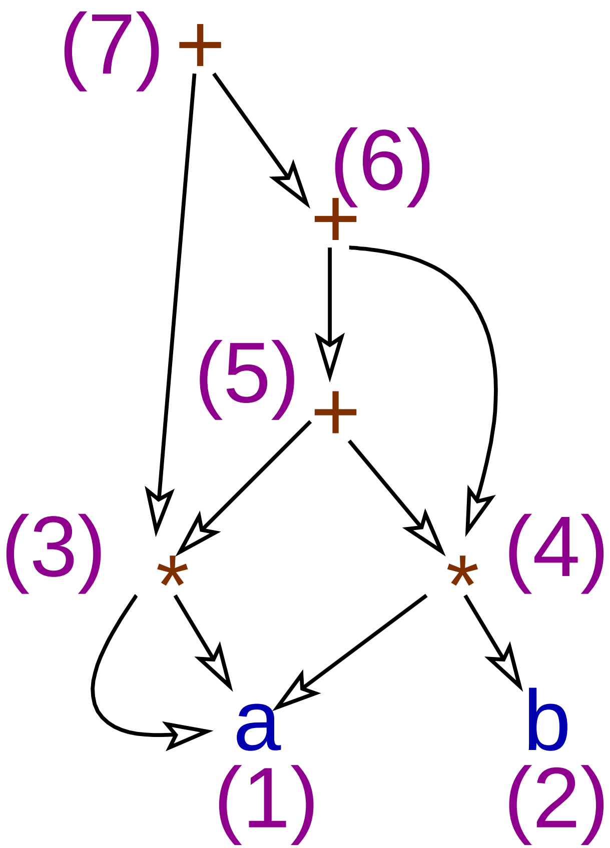
Nodes

- Nodes are organized in such a way that they can be searched efficiently and shared.
- Often nodes are stored in an array of records with a few fields.
- The first field corresponds to a token or an operator.

Nodes

- Other fields correspond to attributes for a leaf node, or indices of its children in case of internal node.
- The index of a node is known as its value number.

DAG and Its Nodes



1	ID(a)		
2	ID(b)		
3	*	1	1
4	*	1	2
5	+	3	4
6	+	5	4
7	+	3	6

SymTab

Note

Searching for a node in a flat array is not efficient so nodes may be arranged as a hash table.

Linear Intermediate Representation

- Both the high-level source code and the target assembly codes are linear in their text.
- The intermediate representation may also be linear sequence of codes. with conditional branches and jumps to control the flow of computation.

Linear Intermediate Representation

- A linear intermediate code may have one operand address[a], two-address[b], or three-address like RISC architectures.
- In fact it may also be zero-address[c]. But we shall only talk about the three-address codes.

[a] Suitable for an accumulator architecture.
[b] Suitable for a register architecture with limited number of registers.
[c] Like a stack machine.

Three-Address Instruction/Code

It is a sequence of instructions of following forms:

1. `a = b` # copy
2. `a = b op c` # binary operation
3. `a[i] = b` # array write
4. `a = b[i]` # array read
5. `goto L` # jump
6. `if a==true goto L` # branch
7. `if a==false goto L`

Three-Address Instruction/Code

8. `a = op b` # unary operation
9. `if a relop b goto L` # relOp and branch
10. `param a` # parameter passing
11. `call p, n` # function call
12. `a = call p, n` # function returns a value
13. `*a = b` # indirect assignment

There may be a few more.

Three-Address Instruction/Code

1. '`a`' corresponds to a source program variable or compiler defined temporary, and '`b`' corresponds to either a variable, or a temporary, or a constant.
2. '`a`' is similar; `b, c` are similar to '`b`' in 1. op is a binary operator.
3. '`a`' is the array name and '`i`' is the byte offset. '`b`' is similar.

Three-Address Instruction/Code

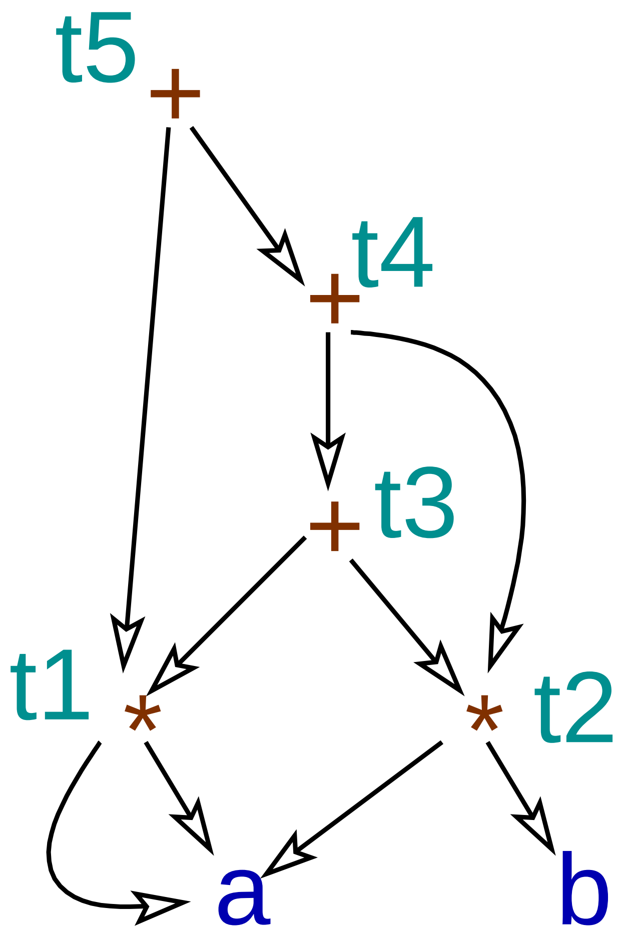
4. Similar.
5. `L` is a label
6. If '`a`' is true, jump to label `L`.
7. If '`a`' is false, jump to label `L`.
8. op is a unary operator.
9. relop is a relational operator.

Three-Address Instruction/Code

10. Passing the parameter '`a`'.
11. Calling the function '`p`', that takes `n` parameters.
12. The return value is stored in '`a`'.
13. Indirection.

Three-Address Code: an Example

t1 = a * a
t2 = a * b
t3 = t1 + t2
t4 = t3 + t2
t5 = t1 + t4



GCC Intermediate Codes

The GCC compiler uses three intermediate representations:

1. GENERIC - it is a language independent tree representation of the entire function.
2. GIMPLE - is a three-address representation generated from GENERIC.
3. RTL - a low-level representation known as register transfer language.

A Example

Consider the following C function.

```
double CtoF(double cel) {
       return cel * 9 / 5.0 + 32 ;
}
```

Readable GIMPLE Code

```
$ cc -Wall -fdump-tree-gimple -S CtoF.c

CtoF (double cel)
{
  double D.1914;

  _1 = cel * 9.0e+0;
  _2 = _1 / 5.0e+0;
  D.1914 = _2 + 3.2e+1;
  return D.1914;
}
```

Raw GIMPLE Code

```
$ cc -Wall -fdump-tree-gimple-raw -S CtoF.c

CtoF (double cel)
gimple_bind <
  double D.1914;

  gimple_assign <mult_expr, _1, cel, 9.0e+0, NULL>
  gimple_assign <rdiv_expr, _2, _1, 5.0e+0, NULL>
  gimple_assign <plus_expr, D.1914, _2, 3.2e+1, NULL>
  gimple_return <D.1914>
>
```

C program with `if`

```
#include <stdio.h>
int main() // code4.c
{
    int l, m ;
    scanf("%d", &l);
    if(l < 10) m = 5*l;
    else m = l + 10;
    printf("l: %d, m: %d\n", l, m);
    return 0;
}
```

Gimple code

`cc -Wall -fdump-tree-gimple -S code4.c`

Output: `code4.c.004t.gimple`

```
main ()
{
  int D.2386;
  {
    int l;
    int m;

    try
      {
        scanf ("%d", &l);
```

```
l.0_1 = l;
if (l.0_1 <= 9) goto <D.2383>; else goto <D.2384>;
<D.2383>:
l.1_2 = l;
m = l.1_2 * 5;
goto <D.2385>;
<D.2384>:
l.2_3 = l;
m = l.2_3 + 10;
<D.2385>:
l.3_4 = l;
printf ("l: %d, m: %d\n", l.3_4, m);
D.2386 = 0;
return D.2386;
```

```
        }
      finally
        {
          l = {CLOBBER};
        }
    }
    D.2386 = 0;
    return D.2386;
}
```

C program with for

```
#include <stdio.h>
int main() // code5.c
{
    int n, i, sum=0 ;
    scanf("%d", &n);
    for(i=1; i<=n; ++i) sum = sum+i;
    printf("sum: %d\n", sum);
    return 0;
}
```

Gimple code

`cc -Wall -fdump-tree-gimple -S code5.c`

Output: `code5.c.004t.gimple`

```
main ()
{
  int D.2387;

  {
    int n;
    int i;
    int sum;

    try
```

```
{
  sum = 0;
  scanf ("%d", &n);
  i = 1;
  goto <D.2384>;
  <D.2383>:
  sum = sum + i;
  i = i + 1;
  <D.2384>:
  n.0_1 = n;
  if (i <= n.0_1) goto <D.2383>; else goto <D.2385>;
  <D.2385>:
  printf ("sum: %d\n", sum);
  D.2387 = 0;
```

```
          return D.2387;
        }
      finally
        {
          n = {CLOBBER};
        }
    }
    D.2387 = 0;
    return D.2387;
}
```

Representation of Three-Address Code

- Any three address code has two essential components: operator and operand.
- There can be at most three operands and one operator.
- The operands are of three types, a name from the source program, a temporary name generated by the compiler or a constant[a].

[a] There are different types of constants used in a programming language.

Representation of Three-Address Code

- There is another category of name, a label in the sequence of three-address codes.
- A three-address code sequence may be represented as a list or array of structures.

Quadruple

- A quadruple is the most obvious first choice[a].
- It has an operator, one or two operands, and the target field.
- Following are a few examples of quadruple representations of three-address codes.

[a]It looks like a RISC instruction at the intermediate level.

Example

Operation	Op_1	Op_2	Target
copy	b		a
add	b	c	a
writeArray	b	i	a
readArray	b	i	a
jmp			L

The variable names are pointers to symbol table.

Example

Operation	Op_1	Op_2	Target
ifTrue	a		L
ifFalse	a		L
minus	b		a
address	b		a
indirCopy	b		a

Example

Operation	Op_1	Op_2	Target
lessEq	a	b	L
param	a		
call	p	n	
copyIndir	b		a

Triple

- A triple is a more compact representation of a three-address code.
- It does not have an explicit target field in the record.
- When a triple u uses the value produced by another triple d, then u refers to the value number (index) of d.
- Following is an example:

Example

```
t1 = a * a

t2 = a * b

t3 = t1 + t2

t4 = t3 + t2

t5 = t1 + t4
```

	Op	Op_1	Op_2
0	mult	a	a
1	mult	a	b
2	add	(0)	(1)
3	add	(2)	(1)
4	add	(0)	(3)

Note

An operand field in a triple can hold a constant, an index of the symbol table or a value number or index of another triple.

Indirect Triple

- It may be necessary to reorder instructions for the improvement of execution.
- Reordering is easy with a quad representation, but is problematic with triple representation as it uses absolute index of a triple.

Indirect Triple

- Indirect triples are used as a solution, where the ordering is maintained by a list of pointers (index) to the array of triples.
- Physically the triples are in their natural translation order.
- But the execution order is maintained by an array of pointers (index) pointing to the array of triples.

Example

Exec.	Order
0	(0)
1	(2)
2	(1)
3	(3)
...	...

	Op	Op_1	Op_2
0	mult	a	b
1	add	(0)	c
2	add	a	b
3	add	(1)	(2)
	...	...	

Static Single-Assignment (SSA) Form

- This representation is similar to three-address code with two main differences.
- Every definition[a] has a distinct name (virtual register).
- Each use of a value refers to a particular definition.

[a] Assignment of value to a variable (user defined or compiler defined) e.g. `t7 = a + t3`.

Static Single-Assignment (SSA) Form

- Each variable is assigned exactly once. If the same user variable is defined on more than one control paths[a], they are renamed using appropriate subscripts.
- When more than one control-flow paths join, a ϕ-function is used to combine them for use.

[a] Conditional statements.

Static Single-Assignment (SSA) Form

- The value of a variable does not change at different points of its use.
- Dataflow path is simple, directly from the definition to the use.

Static Single-Assignment (SSA) Form

- At a join point where more than one control pathe meets the ϕ-function selects its argument[a] depending on the flow of control.
- The ϕ-functions are eliminated before the code generation.

[a] One argument for each incoming edge.

Static Single-Assignment (SSA) Form

- There are variations of SSA forms, but in all of them the value of a variable is independent of the place of use.

```
          x = 1;                x1 = 1;
          a = x + 1             a = x1 + 1
non-SSA:  x = 2        SSA:     x2 = 2
          b = x + 1             b = x2 + 1
```

SSA and Constant Propagation

- Constant Propagation is simple in SSA form.
- If a variable is defined to be a constant (`x = 5`), all its use (`y = x+y`) can be replaced by the constant (`y=2+y`) as there is always a direct dataflow path from definition to use.

SSA and Constant Propagation

- If the use is after a join, the ϕ-function will take care.
- This may reduce the size of definition-use chains.

Example

Consider the following C code:

```
input n
f = 1
i = 1
while i <= n
    f=f*i
    i=i+1
output f
```

The corresponding three-address codes and SSA codes are as follows.

Three-Address & SSA Codes

```
    input n                     input n
    f = 1                       f0 = 1
    i = 1                       i0 = 1
L2: if i>n goto L1    L2:      i1 = φ(i0, i2)
                                f1 = φ(f0, f2)
                                if i1 > n goto L1
    f = f*i                     f2 = f1*i1
    i = i + 1                   i2 = i1 + 1
    goto L2                     goto L2
L1:                   L1:      i3 = φ(i0, i2)
                                f3 = φ(f0, f2)
    output f                    output f3
```

Note

- ϕ-functions selects the value depending on the control-path.
- When the control flows to `L2` from the top, the ϕ-function selects `i0` and `f0`.
- But when the control is transferred from the `goto L2`, it selects `i2` and `f2`.

Note

- We have not talked about the algorithm for insertion of ϕ-functions at the beginning of basic blocks and renaming of incoming variables.
- We also have not talked about how to remove them after the code improvement.

Note

- At the beginning of every basic block all ϕ-functions present are executed concurrently before any other statements.
- New codes are introduced on different control paths.
- `i1` $\leftarrow$ `i0`, `f1` $\leftarrow$ `f0` on control path from top to L2. But `i1` $\leftarrow$ `i2`, `f1` $\leftarrow$ `f2` on control path from `goto L2`.

Note

- Any number of control paths may merge at the beginning of a basic block. A typical example is the join point of a `switch-case` statement.
- So the ϕ-function does not fit in the 3-address code model, and it is necessary to create provision to store arbitrary number of arguments of a ϕ-function.

Basic Block

A basic block is the longest sequence of three-address codes with the following properties.

- The control flows to the block only through the first three-address code[a].
- The control flows out of the block only through the last three-address code[b].

[a] There is no label in the middle of the code.

[b] No three-address code other than the last one can be branch or jump.

Basic Block

- The first instruction of a basic block is called the leader of the block.
- Decomposing a sequence of 3-address codes in a set of basic blocks and construction of control flow graph[a] helps code generation and code improvement.

[a] We shall discuss.

Partitioning into Basic Blocks

The sequence of 3-address codes is partitioned into basic blocks by identifying the leaders.

- The first instruction of the sequence is a leader.
- The target of any jump or branch instruction is a leader.
- An instruction following a jump or branch instruction is a leader.

Example

```
1: L2: v1 = i
2:      v2 = j
3:      if v1>v2 goto L3
4:      v1 = j
5:      v2 = i
6:      v1 = v1 - v2
7:      j = v1
8:      goto L4
9: L3:  v1 = i
10:     v2 = j
11:     v1 = v1 - v2
12:     i = v1
13: L4:v1 = i
14      v2 = j
15      if v1<>v2
          goto L2
```

Leaders in the Example

3-address instructions at index 1, 4, 9, 13 are leaders. The basic blocks are the following.

Basic Block - b_1

```
1: L2: v1 = i
2:     v2 = j
3:     if v1>v2 goto L3
```

Basic Block - b_2

```
4:      v1 = j
5:      v2 = i
6:      v1 = v1 - v2
7:      j = v1
8:      goto L4
```

Basic Block - b_3

```
9: L3: v1 = i
10:     v2 = j
11:     v1 = v1 - v2
12:     i = v1
```

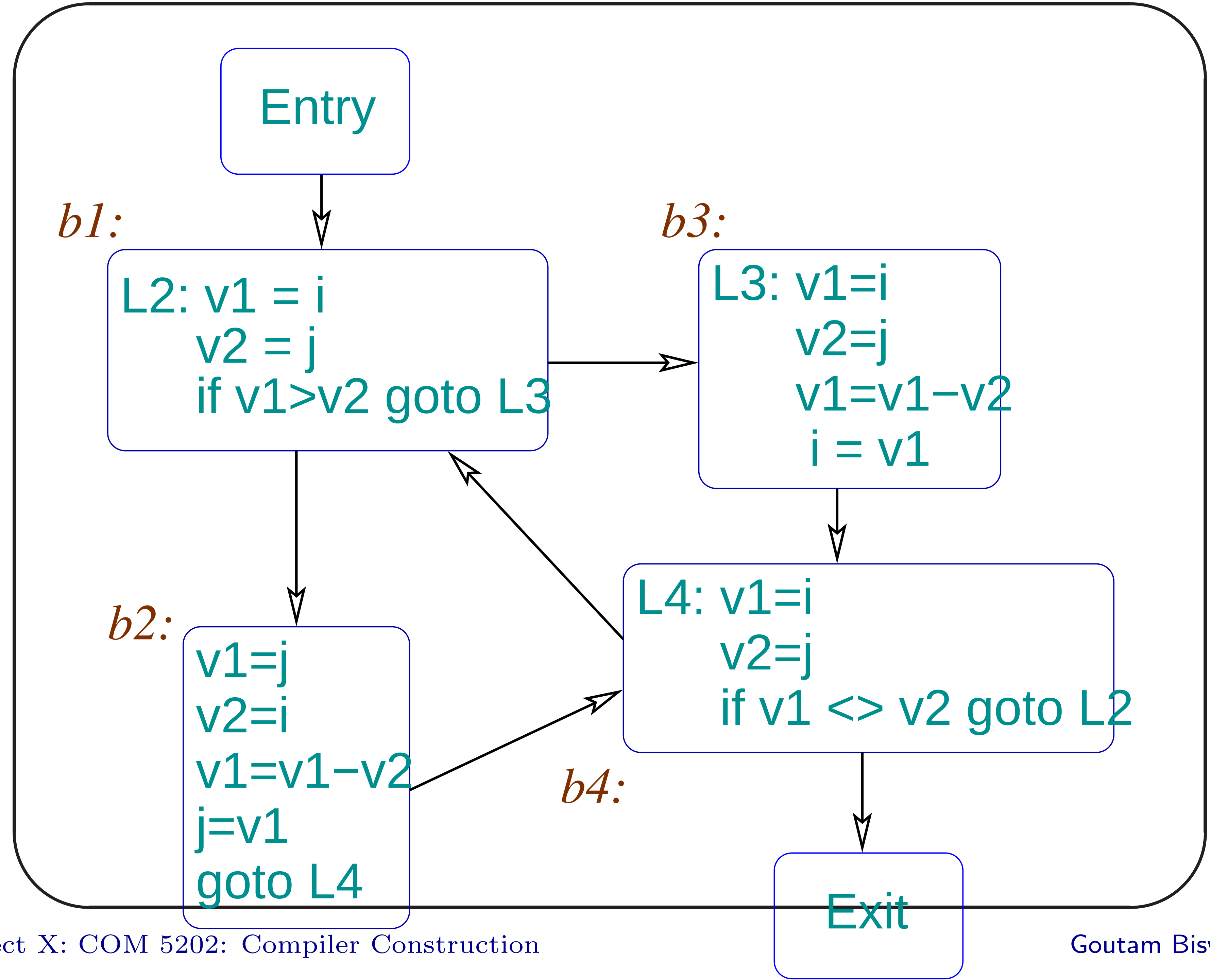
Basic Block - b_4

```
13: L4:v1 = i
14     v2 = j
15     if v1<>v2 goto L2
```

Control-Flow Graph

A control-flow graph is a directed graph $G = (V, E)$, where the nodes are the basic blocks and the edges correspond to the flow of control from one basic block to another. As an example the edge $e_{ij} = (v_i, v_j)$ corresponds to the transfer of flow from the basic block v_i to the basic block v_j.

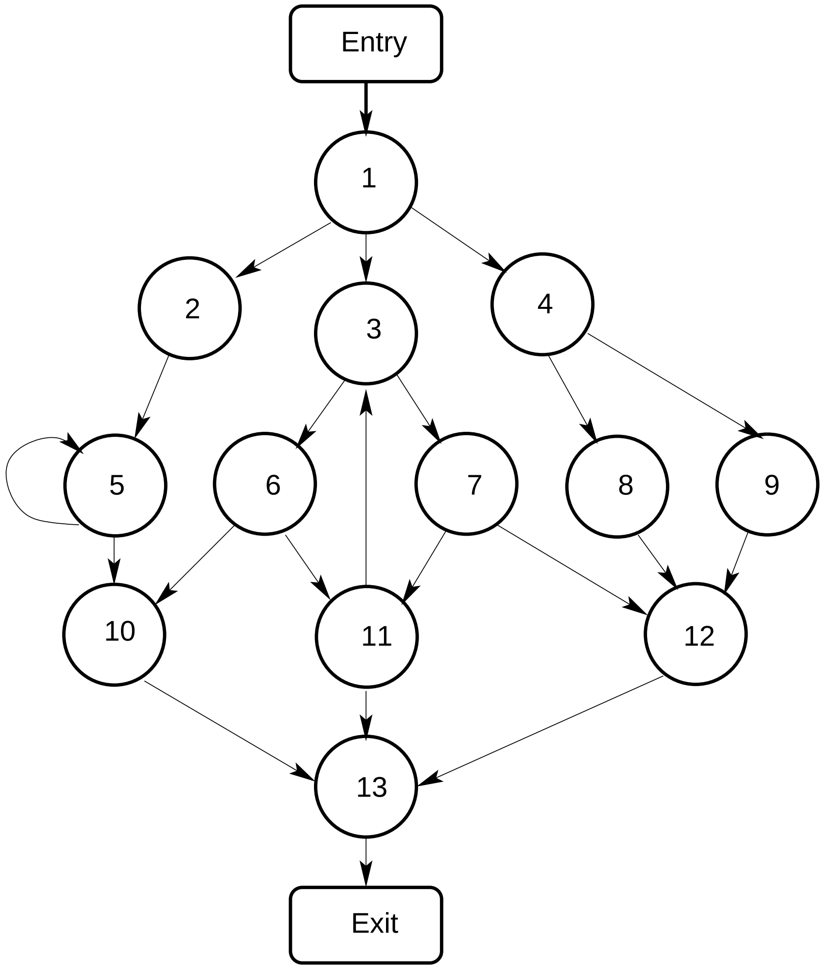
Control-Flow Graph


Entry
b1:
L2: v1 = i
v2 = j
if v1>v2 goto L3
b3:
L3: v1=i
v2=j
v1=v1−v2
i = v1
b2:
v1=j
v2=i
v1=v1−v2
j=v1
goto L4
L4: v1=i
v2=j
if v1 <> v2 goto L2
b4:
Exit

A Few Definitions

- A basic block A of a CFG dominates a basic block B if all paths from the entry node of the CFG to B passes through the block A. We may write A dom B or $A \geq B$. The relation is a partial ordering: $A \geq A$ and transitive.
- A strictly domintes B, A sdom B or $A > B$, if $A \geq B$ but $A \neq B$.

An Example



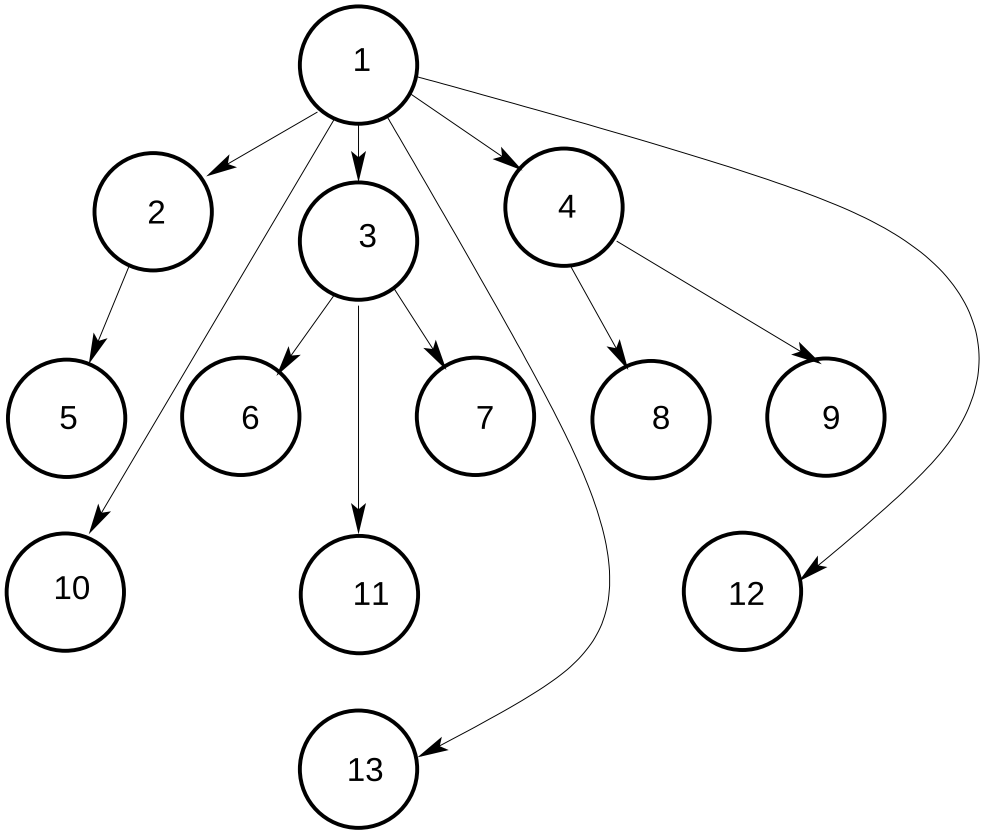
An Example

- The set of nodes dominted by node-3 are $\{3, 6, 7, 11\}$.
- The set of nodes strictly dominted by node-3 are $\{6, 7, 11\}$.
- node 1 dominates every node.

A Few Definitions

- A node I is the immediate dominator of node B. If for all strictly dominator nodes A of B, A strictly dominates I. learly node B cannot be its immediate dominator.
- A dominator tree has every nodes of the CFG. There is an edge from node A to node B, if A is the immediate dominator of B.

Dominator Tree: an Example



A Few Definitions

The dominance frontier of a block A, $DF(A)$, is the set of blocks that are successessors of blocks dominated by A, but are not strictly dominated by A.

$$DF(A) = \{C : B \rightarrow C \ \& \ B \in Dom(A) \ \& \ C \notin SDom(A)\}.$$

Dominance Frontier: an Example

N	1	2	3	4	5	6
$DF(N)$	$\emptyset$	$\{10\}$	$\{3, 10, 12, 13\}$	$\{12\}$	$\{5, 10\}$	$\{10, 11\}$

Note

A basic block is used for improvement of code within the block (local optimization). Our assumption is, once the control enters a basic block, it flows sequentially and eventually reaches the end of the block[a].

[a] This may not be true always. An internal exception e.g. divide-by-zero or unaligned memory access may cause the control to leave the block.

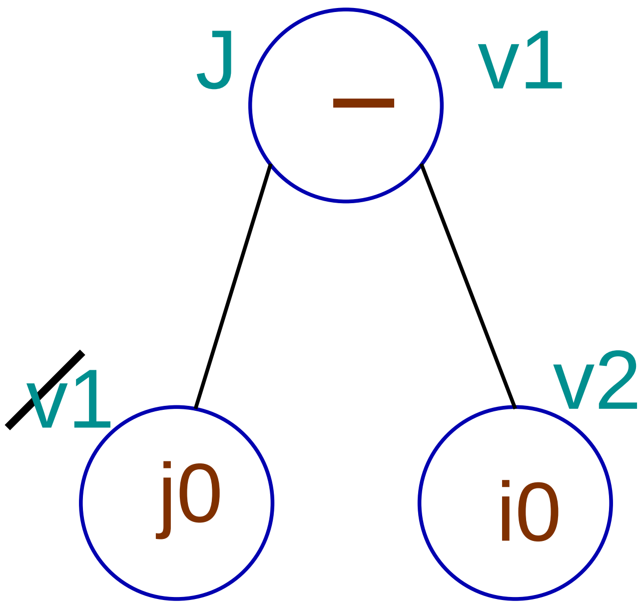
DAG of a Basic Block

- A basic block can be represented by a directed acyclic graph (DAG) which may be useful for some local optimization.
- Each variable entering the basic block with some initial value is represented by a node.
- For each statement in the block we associate a node. There are edges from the statement node to the last definition of its operands.

DAG of a Basic Block

- If N is a node corresponding to the 3-address instruction s, the operator of s should be a label of N.
- If a node N corresponds to the last definition of variables in the block, then these variables are also attached to N.

DAG of b_2



Common Subexpressions

- $V1$ and J stands for the same subexpression when the control comes out of block b_2.
- The variable $V1$ is not live on exit from the block.
- There is no need to keep $V1$.

Exercise

Construct a DAG for the following basic block.

```
a = b + c
b = a - d
c = b + c
d = a - d
```

Show common subexpressions.