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

In a commercial compiler more than one intermediate representations may be used at different levels for code improvement. In a high level intermediate form the language structure is preserved and improvement can be done on it. Whereas a low level intermediate form is suitable for code improvement for the target architecture.

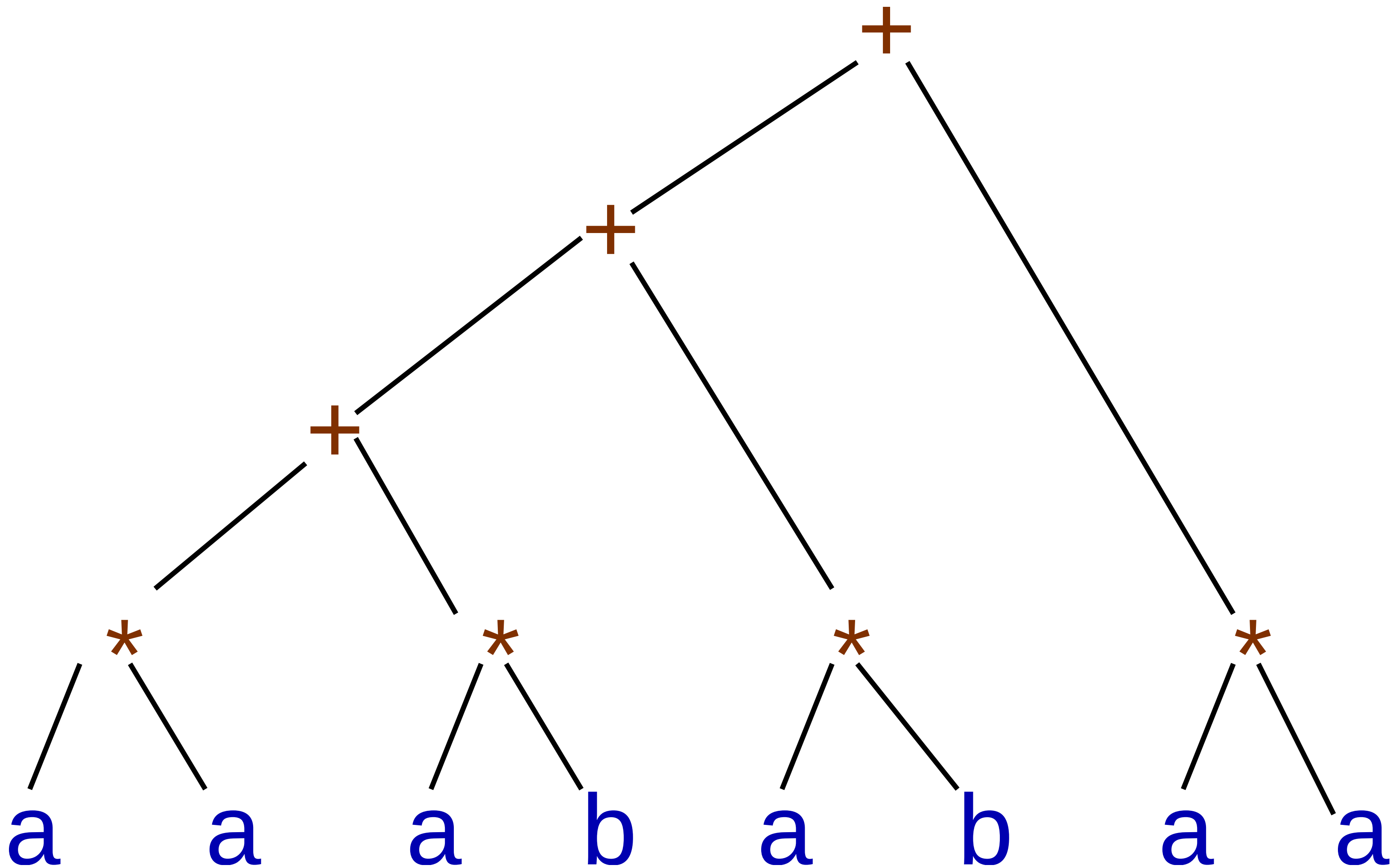
Note

A syntax tree is very similar to a parse tree where extraneous nodes are removed. It is a good representation closer to the source-language and is used in applications like source-to-source translation, syntax-directed editor etc.

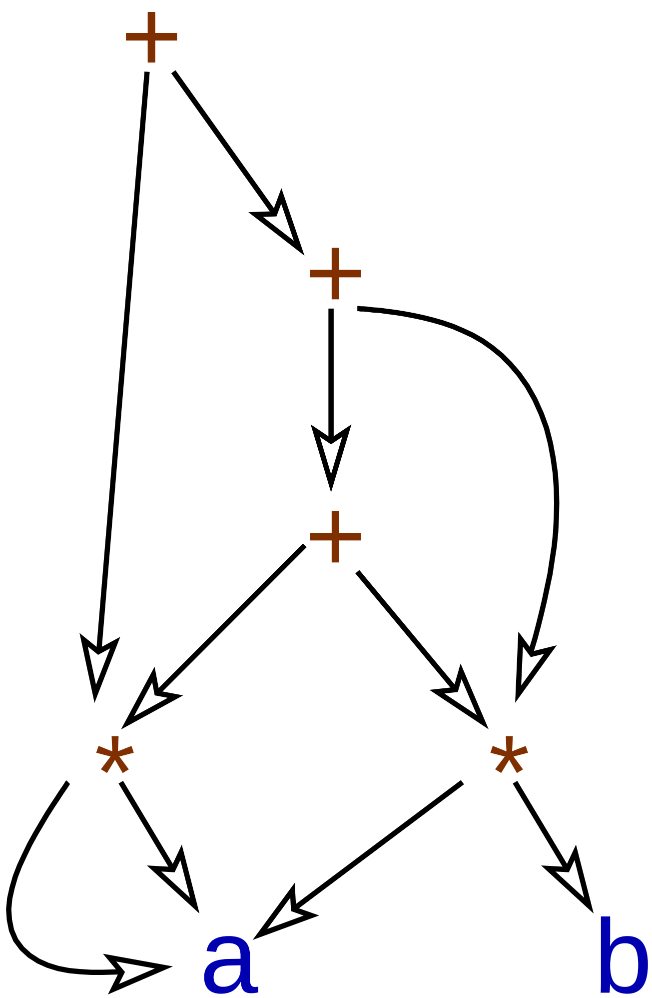
Tree and DAG

A directed acyclic graph (DAG) representation is an improvement over a syntax tree, where subtree duplications such as common subexpressions are identified and shared.

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. 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.

SDT for Tree and DAG

We have the classic expression grammar G. Following are syntax directed translations to construct expression tree and DAG.

SDT for Tree

$F \rightarrow$ **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

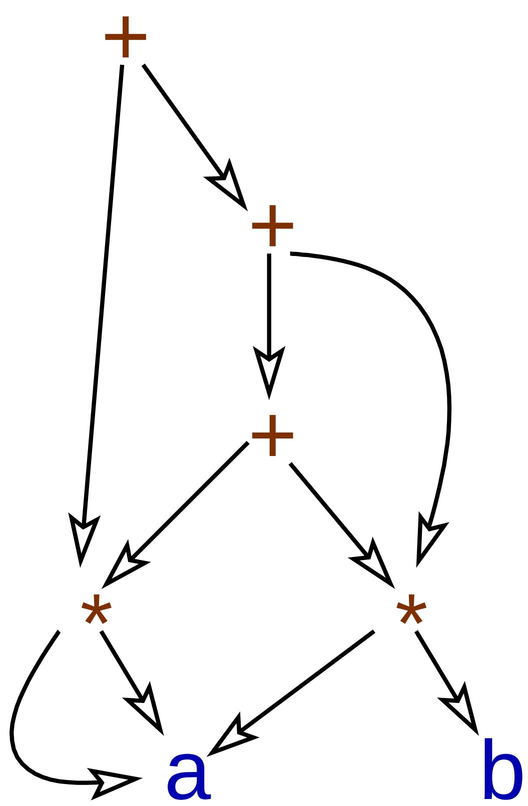
$E \rightarrow E_1 + T$

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

Nodes

It is necessary to organized the nodes 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 corresponding to the node. Other fields correspond to the attributes in case of a leaf node, and indices of its children in case of an internal node.
The index of a node is known as its value number.

DAG and Its Nodes



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

Note

Searching for a node in a flat array is not efficient and the 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 internal representation may also be linear. But then a linear intermediate form should include conditional branches and jumps to control the flow.

Linear Intermediate Representation

Linear intermediate code, like the assembly language code, may be one-address, suitable for an accumulator architecture, two-address, suitable for a register architecture with limited number of registers where one operand is destroyed, or three-address for modern architectures.
In fact it may also be zero-address for a stack machine. We shall only talk about the three-address codes.

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.

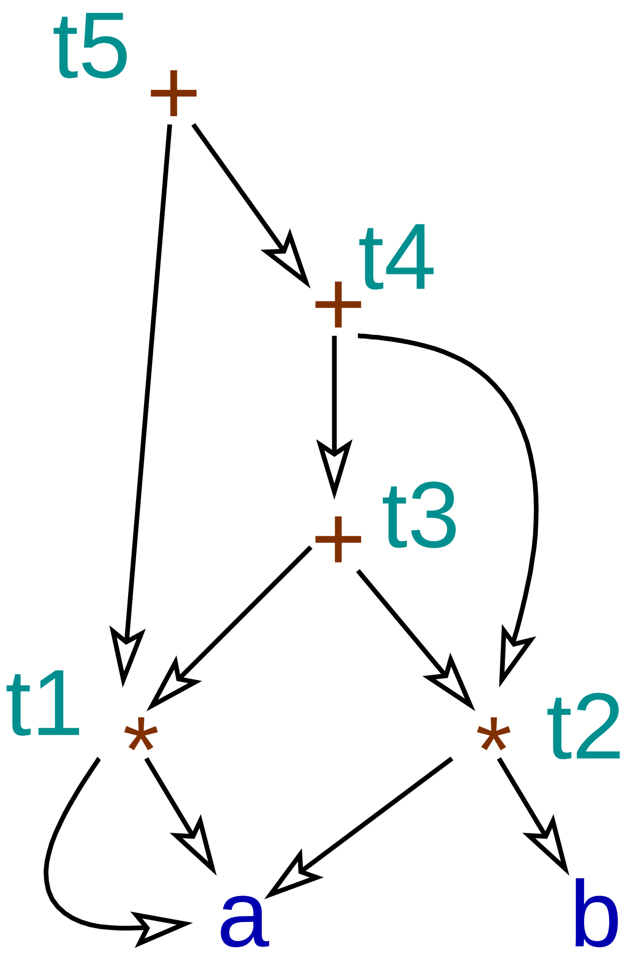
Three-Address Instruction/Code

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.
4. Similar.

Three-Address Instruction/Code

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

Three-Address Code: 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.1248;
  double D.1249;
  double D.1250;

  D.1249 = cel * 9.0e+0;
  D.1250 = D.1249 / 5.0e+0;
  D.1248 = D.1250 + 3.2e+1;
  return D.1248;
}
```

Raw GIMPLE Code

```
$ cc -Wall -fdump-tree-gimple-raw -S ctof.c
CtoF (double cel)
gimple_bind <
  double D.1588;
  double D.1589;
  double D.1590;

  gimple_assign <mult_expr, D.1589, cel, 9.0e+0>
  gimple_assign <rdiv_expr, D.1590, D.1589, 5.0e+0>
  gimple_assign <plus_expr, D.1588, D.1590, 3.2e+1>
  gimple_return <D.1588>
>
```

C program with if

```
#include <stdio.h>
int main() // cCode4.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 cCode4.c
```

Output: `cCode4.c.004t.gimple`

```
main ()
{
  const char * restrict D.2046;
  int l.0;
  int l.1;
  int l.2;
  int l.3;
  const char * restrict D.2054;
  int D.2055;
  int l;
```

```
int m;

D.2046 = (const char * restrict) &"%d"[0];
scanf (D.2046, &l);
l.0 = l;
if (l.0 <= 9) goto <D.2048>; else goto <D.2049>;
<D.2048>:
l.1 = l;
m = l.1 * 5;
goto <D.2051>;
<D.2049>:
l.2 = l;
m = l.2 + 10;
<D.2051>:
```

```
  l.3 = l;
  D.2054 = (const char * restrict) &"l: %d, m: %d\n"[0]
  printf (D.2054, l.3, m);
  D.2055 = 0;
  return D.2055;
}
```

C program with `for`

```
#include <stdio.h>
int main() // cCode5.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 cCode5.c
```

Output: `cCode5.c.004t.gimple`

```
main ()
{
  const char * restrict D.2050;
  int n.0;
  const char * restrict D.2052;
  int D.2053;
  int n;
  int i;
  int sum;
```

```
sum = 0;
D.2050 = (const char * restrict) &"%d"[0];
scanf (D.2050, &n);
i = 1;
goto <D.2047>;
<D.2046>:
sum = sum + i;
i = i + 1;
<D.2047>:
n.0 = n;
if (i <= n.0) goto <D.2046>; else goto <D.2048>;
<D.2048>:
D.2052 = (const char * restrict) &"sum: %d\n"[0];
printf (D.2052, sum);
```

```
  D.2053 = 0;
  return D.2053;
}
```

Representation of Three-Address Code

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

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

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 uses the value produced by another triple, the user-triple refers to the definition-triple by its index. 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 an index of its own.

Indirect Triple

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

Indirect Triple

- As a solution indirect triples are used, where the ordering is maintained by a list of pointers (index) to the array of triples.
- The triples are in their natural translation order and can be accessed by their indexes. But the execution order is maintained by an array of pointers (index) 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 assignment uses different variable (virtual register) name. This helps certain code improvement.

  It tries to encode the definition[a] and use of a name. Each name is defined only once and so it is called static single-assignment.

[a]Here a definition means an assignment of value to a variable.

Static Single-Assignment (SSA) Form

- In a conditional statement if the same user variable is defined on more than one control paths, they are renamed as distinct variables with appropriate subscripts.
- Finally when the paths join, a ϕ-function is used to combine the variables. The ϕ-function selects the value of its arguments depending on the flow-path.

Example

Consider the following C code:

```
for(f=i=1; i<=n; ++i) f = f*i;
```

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

Three-Address & SSA Codes

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

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
```

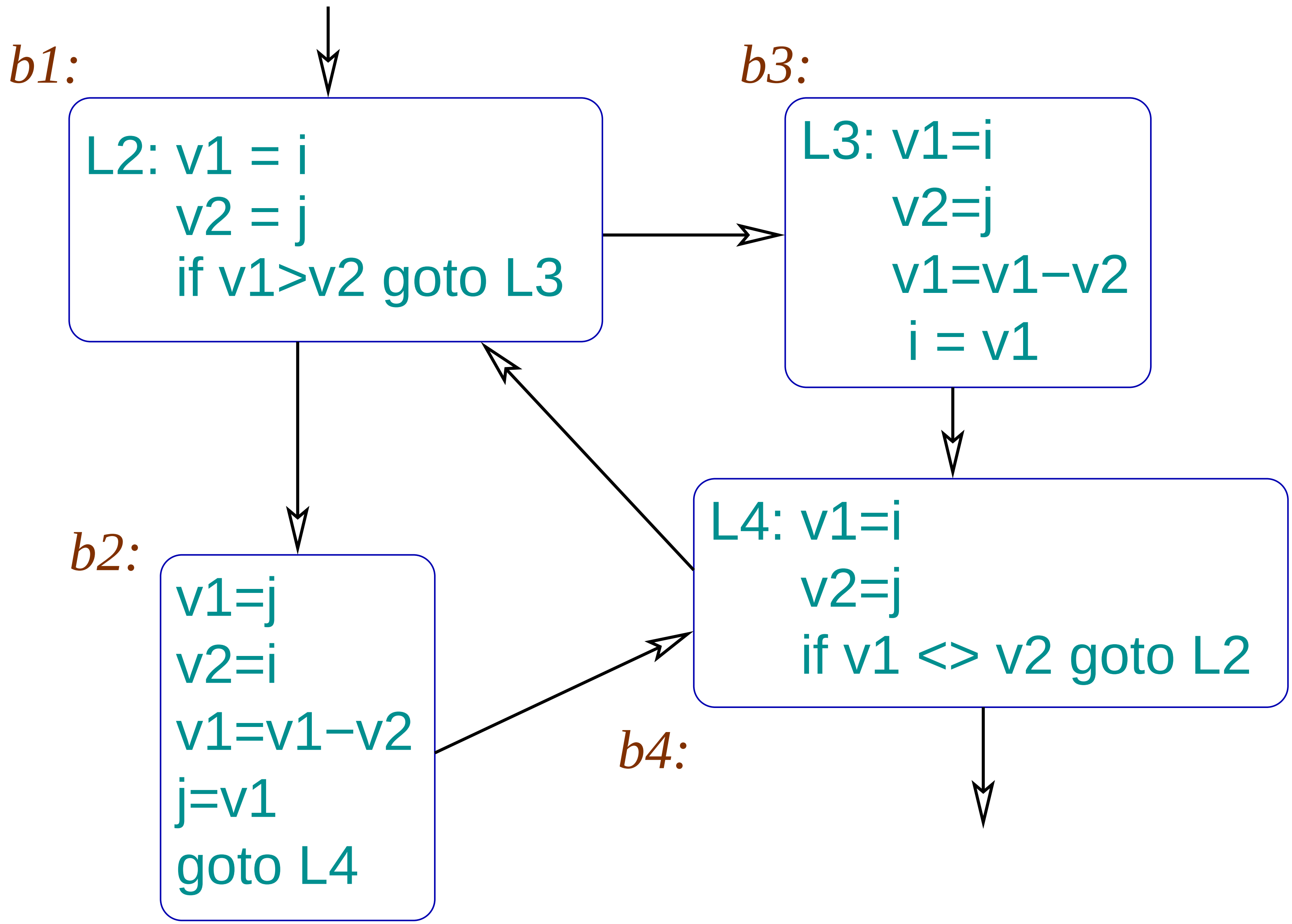
Basic Block - b_4

```
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


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:

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