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.

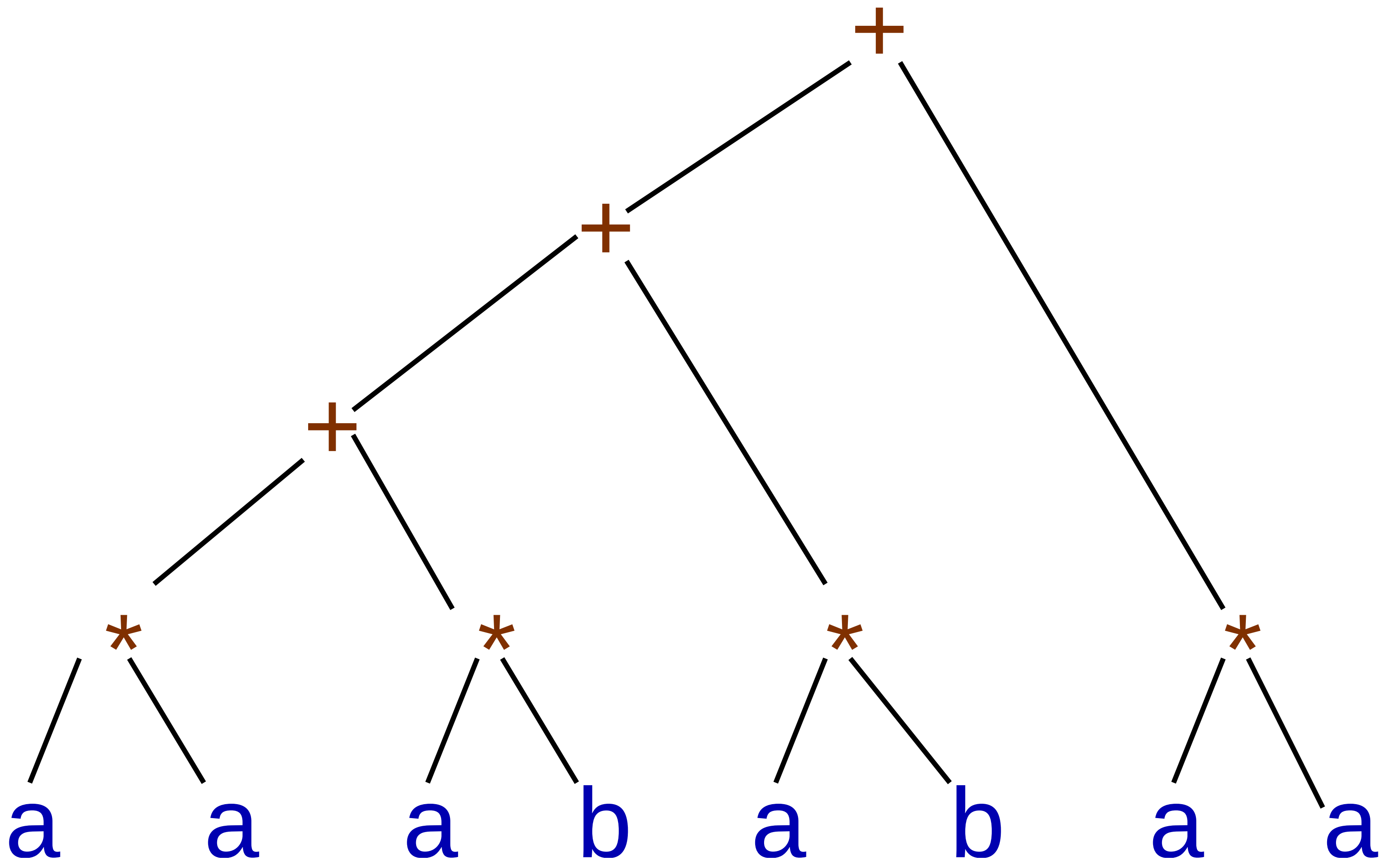
Tree Representations

- 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 may be used in applications like source-to-source translation, or syntax-directed editor etc.

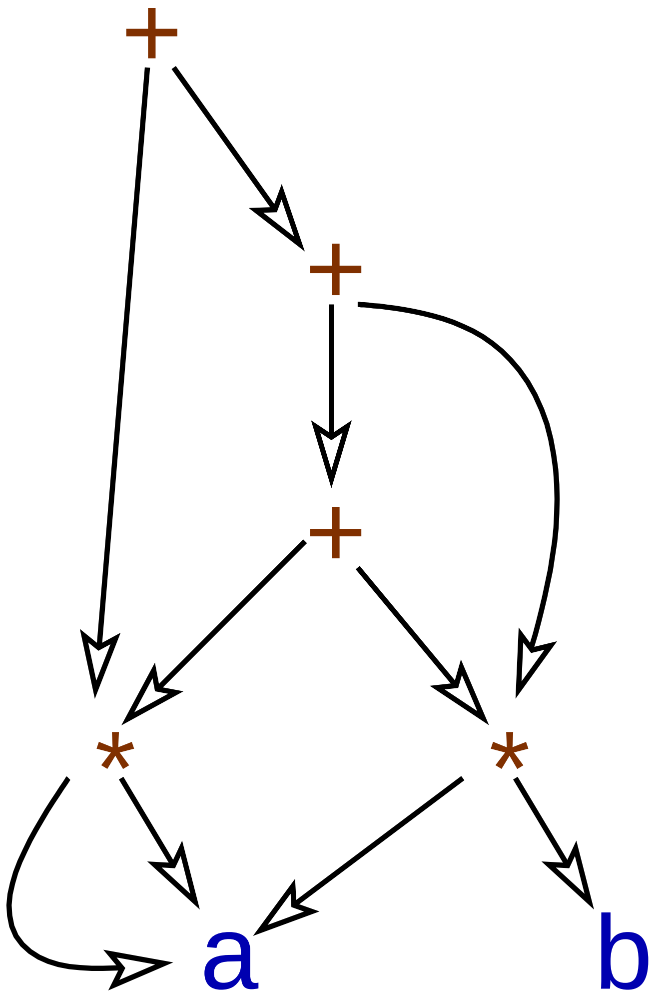
Directed Acyclic Graph (DAG)

- 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.

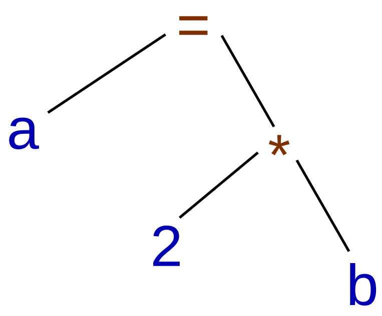
Low-Level Tree

- 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.

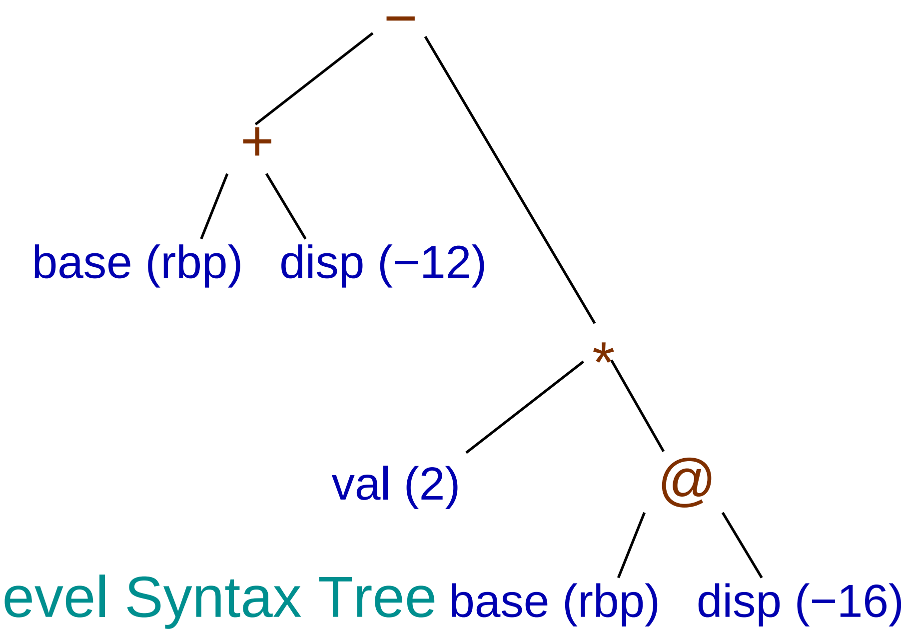
Low-Level Tree

- 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`



High−level Syntax Tree



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.

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 interprocdurial analysis of code. There is an edge from each instance of call to the procedure.

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

$F \rightarrow$ `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

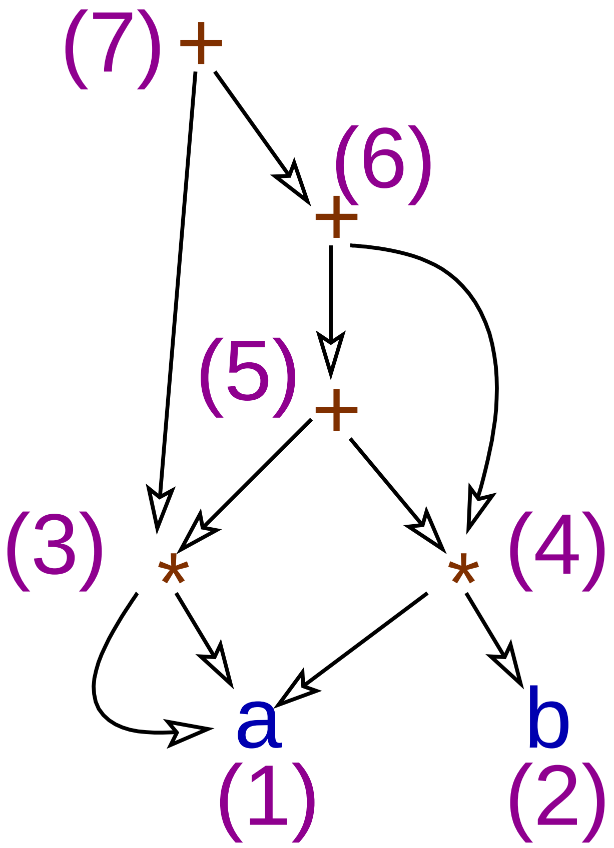
- Nodes are organized in such a way that they can be searched efficiently and shared.
- Often nodes are stored in an array of records with a few fields.
- The first field corresponds to a token or an operator.

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

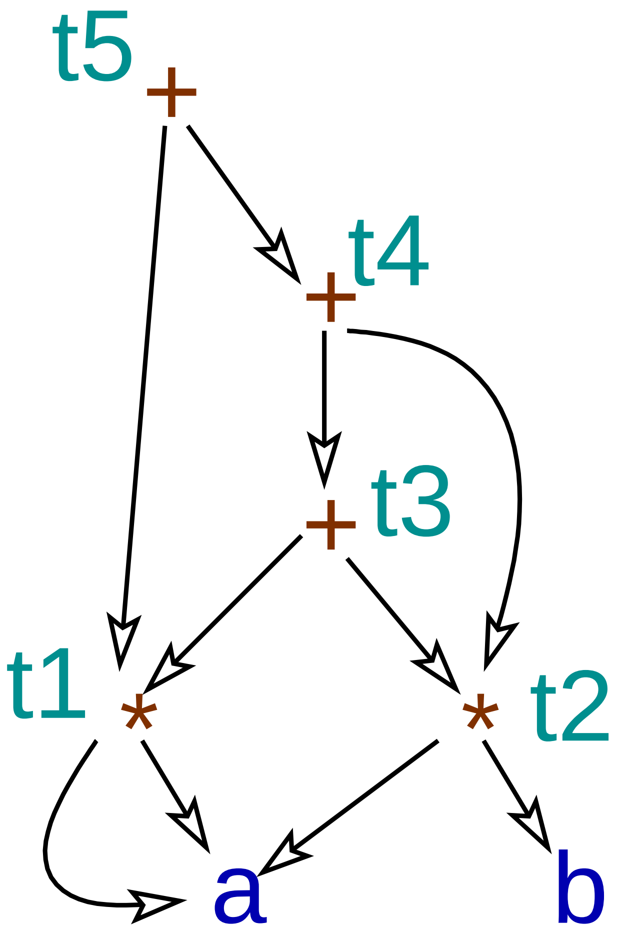
4. Similar.
5. `L` is a label
6. If ‘`a`’ is true, jump to label `L`.
7. If ‘`a`’ is false, jump to label `L`.
8. op is a unary operator.
9. relop is a relational operator.

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.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 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	Op1	Op2	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

- 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) 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

- If the same user variable is defined on more than one control paths[a], they are renamed as distinct variables with appropriate subscripts.
- When more than one control-flow paths join, a ϕ-function is used to combine the variables.

---

[a] Conditional statements.

Static Single-Assignment (SSA) Form

- The ϕ-function selects the value of its arguments depending on the control-flow path (data-flow under control-flow).
- Each name is defined at one place[a]. Use of a name contains information about the location of its definition (data-flow).
- SSA-form tries to encode data-flow under flow-control.

---

[a] So the name static single-assignment (SSA).

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
    f = 1
L2: if i>n goto -


    f = f*i
    i = i + 1
    goto L2
```

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

Note

- We have not talked about the implementation of ϕ-function. It 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

- 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 ← i0, f1 ← f0` on control path from top to L2. But `i1 ← i2, f1 ← 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
```

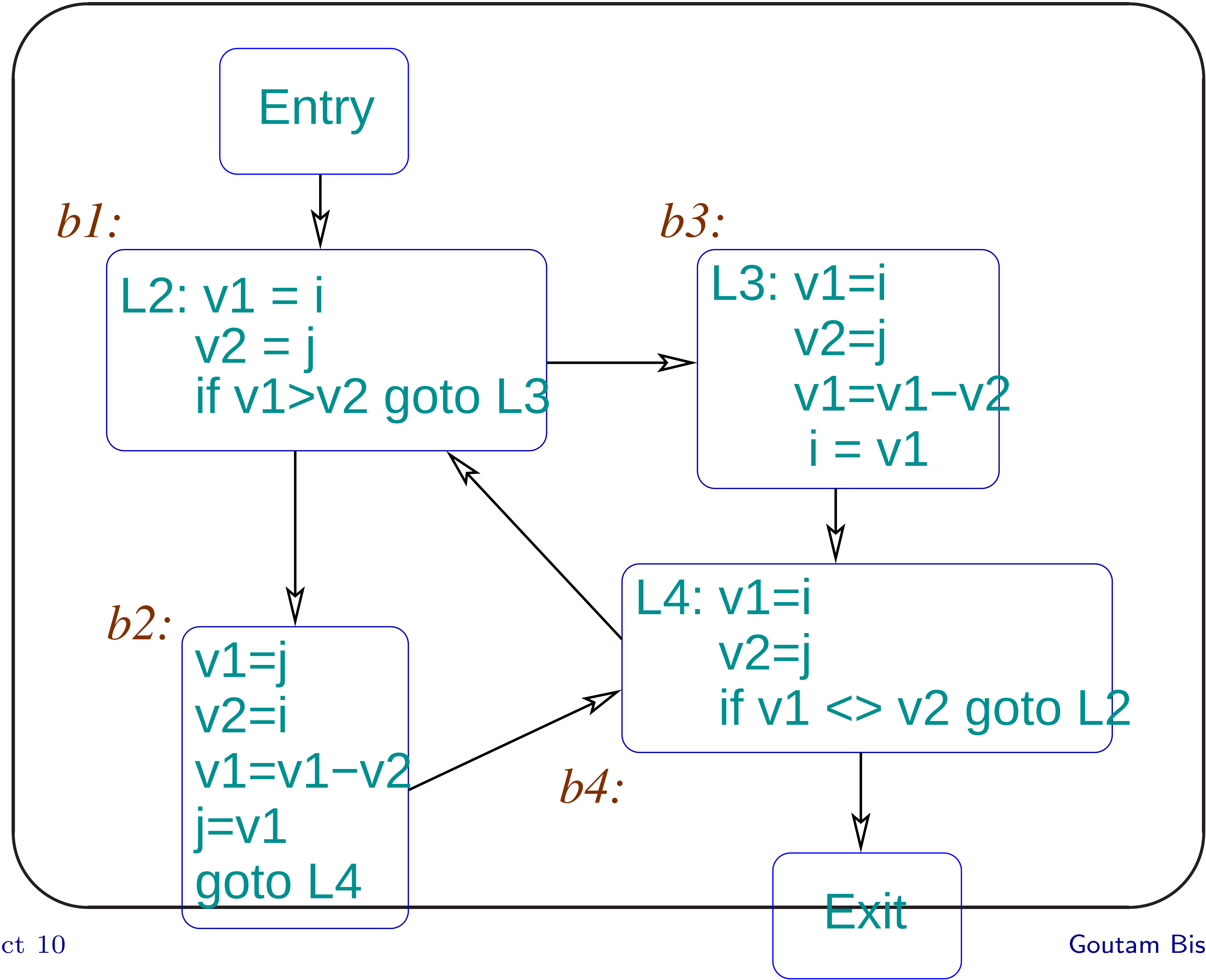
Basic Block - b_4

```
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
```

b2:

```
v1=j
v2=i
v1=v1−v2
j=v1
goto L4
```

b3:

```
L3: v1=i
    v2=j
    v1=v1−v2
     i = v1
```

b4:

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

Exit

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