Code Generation (Part -2)

Course : 2CS701/IT794 – Compiler Construction

Prof Monika Shah

Nirma University

Ref : Ch.9 Compilers Principles, Techniques, and Tools by Alfred Aho, Ravi Sethi, and Jeffrey Ullman

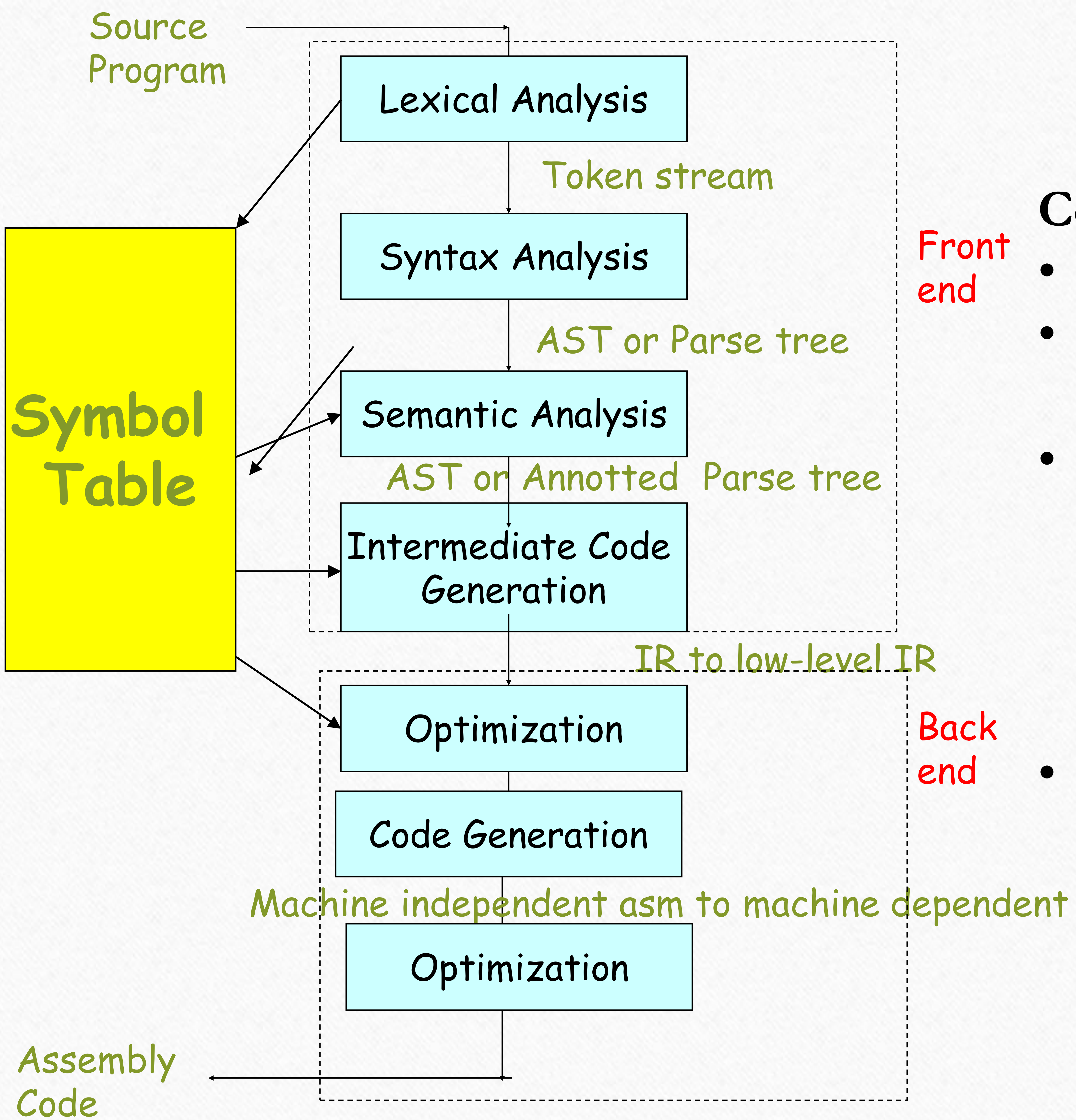
Glimpse

Part-1

- Introduction
- Code Generation Issues

Part-2

- **Basic Code Generation Case Study**
- **Introduction to Basic Block, and Control Flow Diagram**



Code Generation

- Final Phase of Compilation
- Produces a semantically equivalent target program
- Main Functionalities :
  - Instruction selection
  - Register allocation and assignment
  - Instruction ordering

- **Functionalities for QoS**
  - Produce Correct code (semantic preserving)
  - Efficiency
    - Effective use of resources
    - User Heuristics to generate good, but suboptimal code

Case Study of Basic Code Generation

The Borland 3.0 C Compiler for the 80X86

- Expression statement with symbols
- Array reference
- Structure Field reference
- Pointer reference
- If statement
- While statement
- Function call

Case Study of Basic Code Generation

The Borland 3.0 C Compiler for the 80X86

Expression statement

Expression statement : (x = x +3) + 4

Assembly Code :

```
mov ax, word ptr [bp-2]
add ax, 3
mov word ptr [bp-2], ax
add ax, 4
```

Notes

- The *bp* is used as the frame pointer.
- The static simulation method is used to convert the intermediate code into the target code.

Case Study of Basic Code Generation

The Borland 3.0 C Compiler for the 80X86

Array Reference

Array reference Example :

(a [i + 1] = 2) + a [j]

Where, i and j are declared as

int i, j;

int a[10];

(1) mov bx,word ptr [bp-2]
(2) shl bx , 1
(3) l ea ax, word ptr [bp-22]
(4) add bx , ax

(5) mov ax , 2
(6) mov word ptr [bx],ax

(7) mov bx,word ptr [bp-4]
(8) shl bx , 1
(9) l ea dx,word ptr [bp-24]
(1 0) add bx , dx
(1 1) add ax,word ptr [bx]

Case Study of Basic Code Generation

The Borland 3.0 C Compiler for the 80X86

Structure Field Reference

Structure Example

```
typedef struct rec
{  int i;
   char c;
   int j;
} Rec;
Rec x;
x.j =x.i;
```

```
mov ax, word ptr [bp-6]
mov word ptr [bp-3],ax
```

Note:

Integer variable has size 2 bytes;
Character variable has size 1 bytes;
Local variables are allocated only on even-byte boundaries;
The offset computation for j (-6 + 3 = -3) is performed statically by the compiler.

Case Study of Basic Code Generation

The Borland 3.0 C Compiler for the 80X86

Pointer Field Reference

```
typedef struct treeNode
{   int val;
    struct treeNode * lchild,
     * rchild;
} TreeNode;
...
Rec x;
TreeNode *p;
```

The code generated for the statement

p->lchild = p;

is

mov word ptr [si+2], si

And the statement

p = p->rchild;

is

mov si, word ptr [si+4]

Note:

Assume pointer has size 2

Case Study of Basic Code Generation

The Borland 3.0 C Compiler for the 80X86

IF Statement

```
if (x>y)
y++;
else x--;
```

```
        mov bx, word ptr [bp-2]  //x
        mov dx, word ptr [bp -4] //y
        cmp bx , dx
        jle short @1@86
        inc dx
        jmp short @1@114
@1@86 :
        dec bx
@1@114 :
```

Case Study of Basic Code Generation

The Borland 3.0 C Compiler for the 80X86

While Statement

```
while (x<y)
y -= x;
```

```
   jmp short @1@170
@1@142 :
      sub dx , bx
@1@170 :
      cmp bx , dx
      jl short @1@142
```

Case Study of Basic Code Generation

The Borland 3.0 C Compiler for the 80X86

Function Call

- function definition:

```
int f( int x, int y)
{
  return x+y+1  ;
  }
```

- And a corresponding call

```
f  (2+3, 4)
```

```
mov ax,4
push ax
mov ax,5
push ax
call near ptr _f
pop cx
pop cx
```

Notes:

The arguments are pushed on the stack in reverse order;
The caller is responsible for removing the arguments from the stack after the call.
The call instruction on the 80x86 automatically pushes the return address onto the stack.

Case Study of Basic Code Generation

The Borland 3.0 C Compiler for the 80X86

Function Definition

```
_ f   proc near
      push bp
      mov bp , sp
      mov ax, word ptr [bp+4]
      add ax, word ptr [bp+6]
      inc ax
      jmp short @1@58
@1@58 :
      pop bp
      ret
_ f   endp
```

Basic Blocks and Flow of Control

- Partition the intermediate code into basic blocks
  - The flow of control can only enter the basic block through the first instruction in the block. That is, there are no jumps into the middle of the block.
  - Control will leave the block without halting or branching, except possibly at the last instruction in the block.
- The basic blocks become the nodes of a flow graph
- Use : Code Optimization, Register Allocation

Flow Graphs

- A *flow graph* is a graphical depiction of a sequence of instructions with control flow edges
- A flow graph can be defined at the intermediate code level or target code level

```
      MOV 1,R0
      MOV n,R1
      JMP L2
L1:   MUL 2,R0
      SUB 1,R1
L2:   JMPNZ R1,L1
```

```
      MOV 0,R0
      MOV n,R1
      JMP L2
L1:   MUL 2,R0
      SUB 1,R1
L2:   JMPNZ R1,L1
```

Basic Blocks

- A *basic block* is a sequence of consecutive instructions with exactly one entry point and one exit point (with natural flow or a branch instruction)

```
    MOV 1,R0
    MOV n,R1
    JMP L2
L1: MUL 2,R0
    SUB 1,R1
L2: JMPNZ R1,L1
```

```
    MOV 1,R0
    MOV n,R1
    JMP L2
```

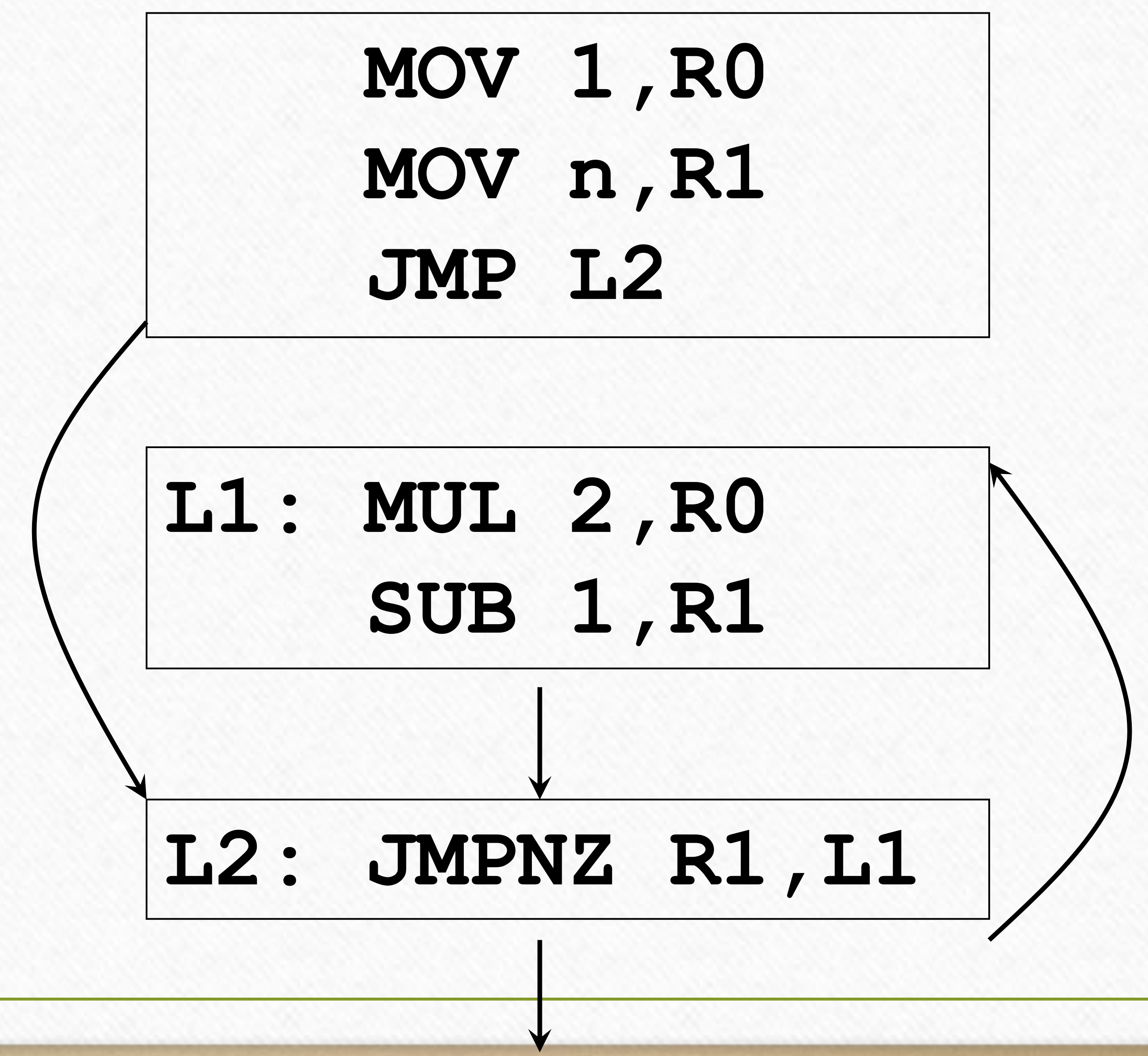
```
L1: MUL 2,R0
    SUB 1,R1
```

```
L2: JMPNZ R1,L1
```

Basic Blocks and Control Flow Graphs

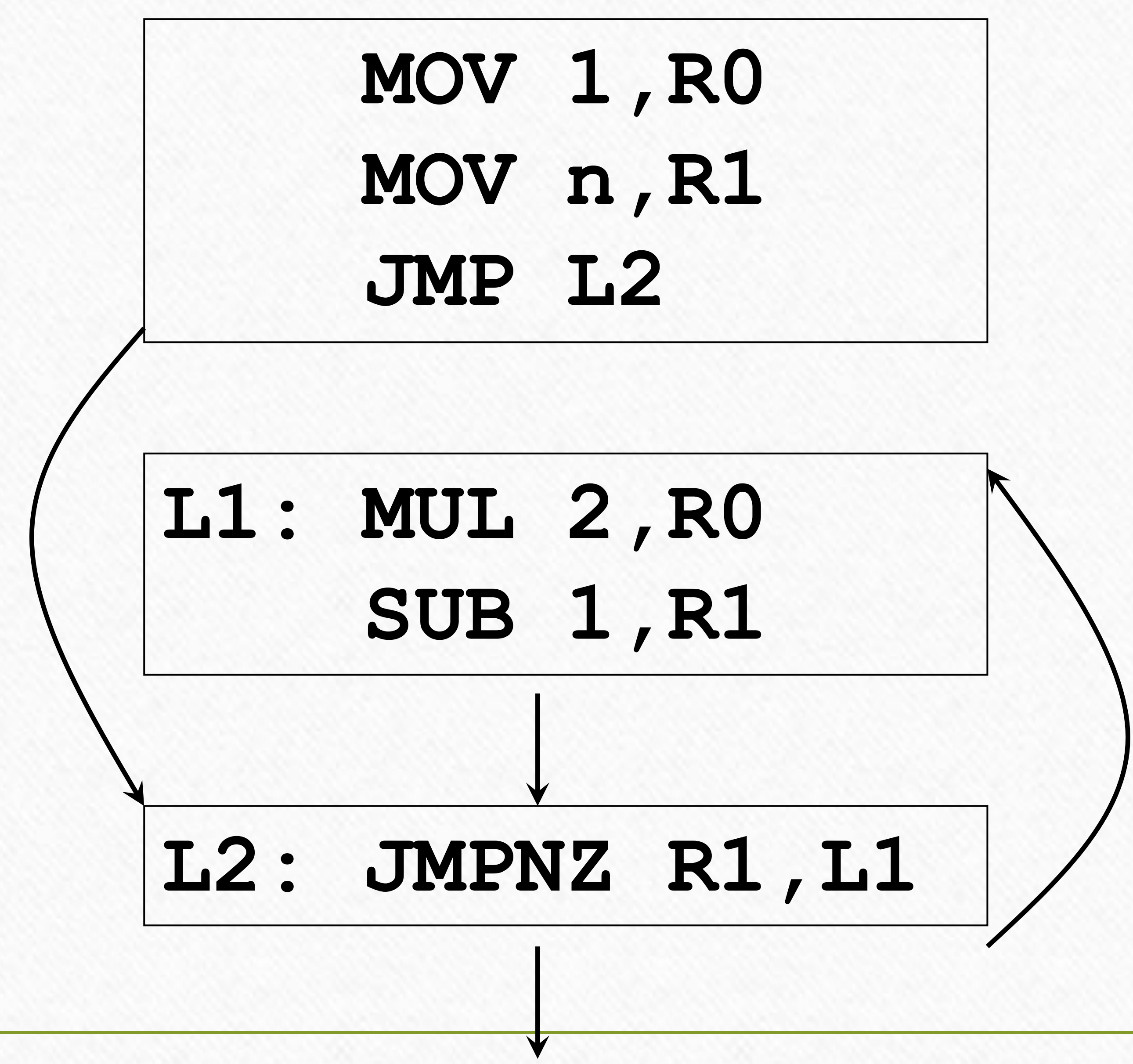
- A *control flow graph* (CFG) is a directed graph with basic blocks B_i as vertices and with edges $B_i \rightarrow B_j$ iff B_j can be executed immediately after B_i

```
     MOV 1,R0
     MOV n,R1
     JMP L2
L1:  MUL 2,R0
     SUB 1,R1
L2:  JMPNZ R1,L1
```



Successor and Predecessor Blocks

- Suppose the CFG has an edge $B_1 \rightarrow B_2$
  - Basic block B_1 is a *predecessor* of B_2
  - Basic block B_2 is a *successor* of B_1



Partition Algorithm for Basic Blocks

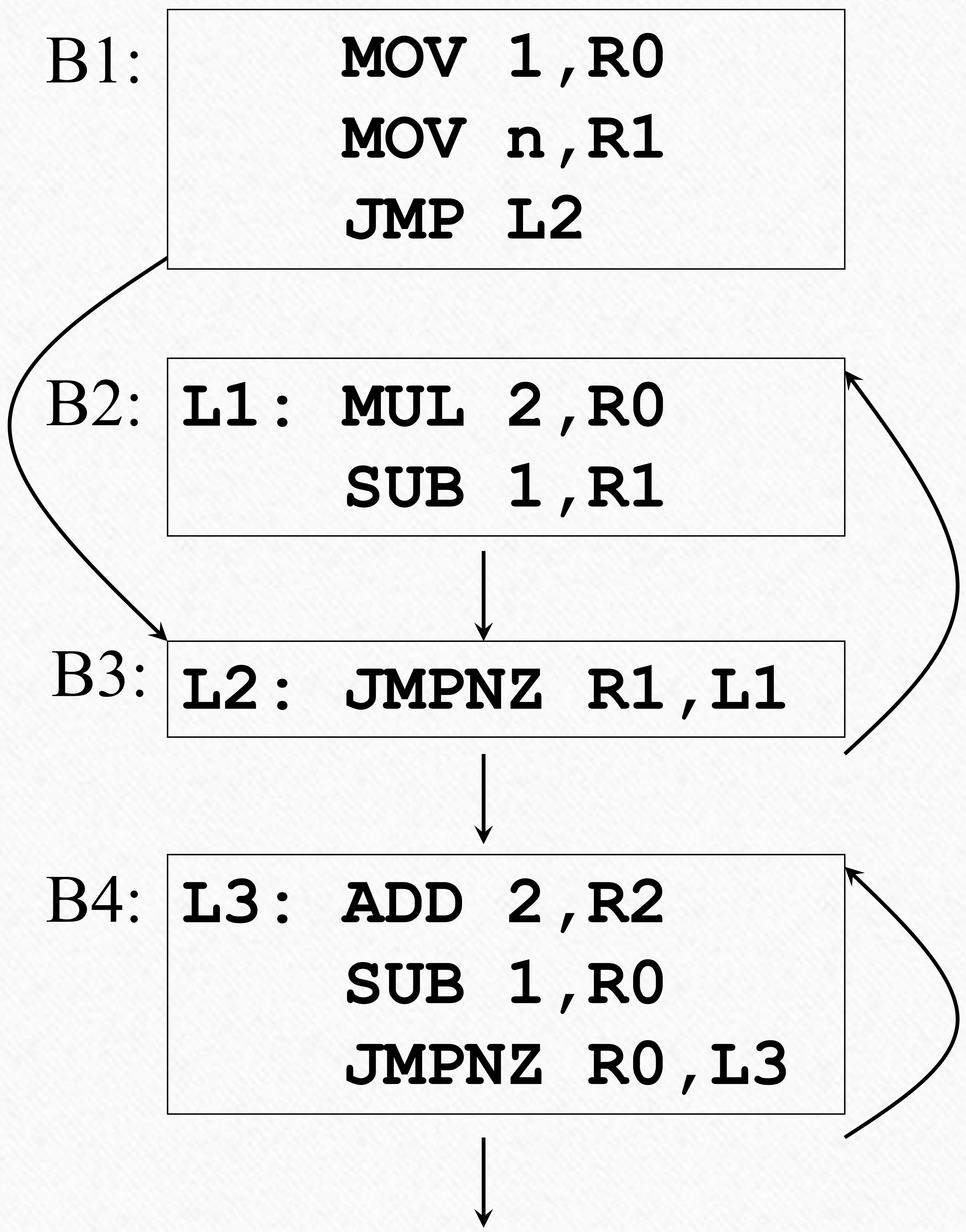
Input: A sequence of three-address statements
Output: A list of basic blocks with each three-address statement in exactly one block

1. Determine the set of *leaders*, the first statements if basic blocks
   a) The first statement is the leader
   b) Any statement that is the target of a goto is a leader
   c) Any statement that immediately follows a goto is a leader
2. For each leader, its basic block consist of the leader and all statements up to but not including the next leader or the end of the program

Loops

- A *loop* is a collection of basic blocks, such that
  - All blocks in the collection are *strongly connected*
  - The collection has a unique *entry*, and the only way to reach a block in the loop is through the entry

Loops (Example)



Strongly connected components:

SCC={ {B2,B3},
{B4} }

Entries:
B3, B4

Equivalence of Basic Blocks

- Two basic blocks are (semantically) *equivalent* if they compute the same set of expressions

```
b  := 0
t1 := a + b
t2 := c * t1
a  := t2
```

```
a  := c * a
b  := 0
```

```
a := c*a
b := 0
```

```
a := c*a
b := 0
```


Blocks are equivalent, assuming `t1` and `t2` are *dead*: no longer used (no longer *live*)

Transformations on Basic Blocks

- A *code-improving transformation* is a code optimization to improve speed or reduce code size
- *Global transformations* are performed across basic blocks
- *Local transformations* are only performed on single basic blocks
- Transformations must be safe and preserve the meaning of the code
  - A local transformation is safe if the transformed basic block is guaranteed to be equivalent to its original form