Program Analysis

Paolo Falcarin

Ca' Foscari University of Venice

Department of Enviromental Sciences, Informatics and Statistics

paolo.falcarin@unive.it

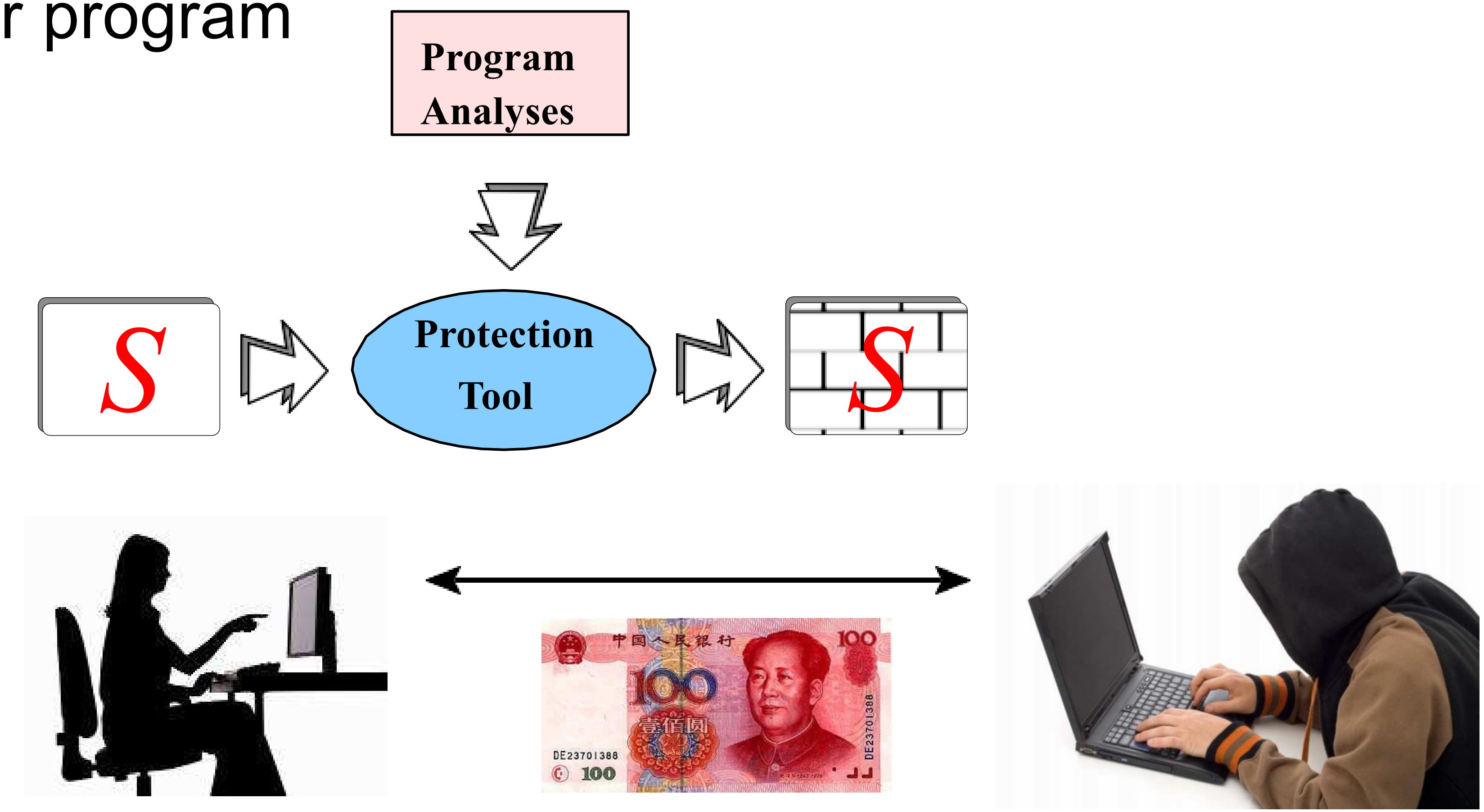
CM0626 – Software Security

CM0631-2 – Software Security

18 February 2025

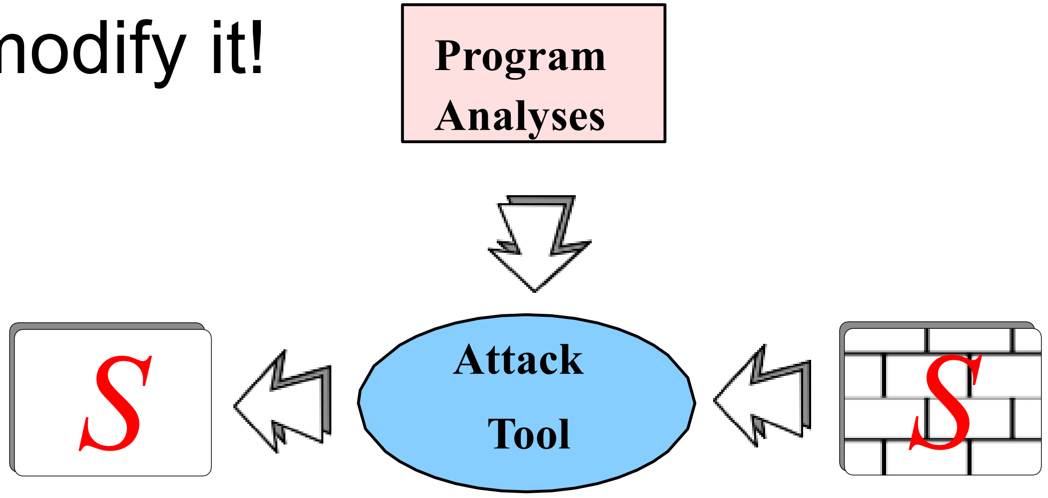
Program Analysis

Defenders analyze their program to protect it!



Malicious Program Analysis

Attackers analyze our program to modify it!



Program Analysis

Ca' Foscari University of Venice

```
int foo() {
  int x;
  int* y;
  printf(x+*y);
}
```

- Who calls foo?
- Who does foo call?
- Is x ever initialized?
- Can y ever be null?
- What will foo print?

Program Analysis

Ca' Foscari University of Venice

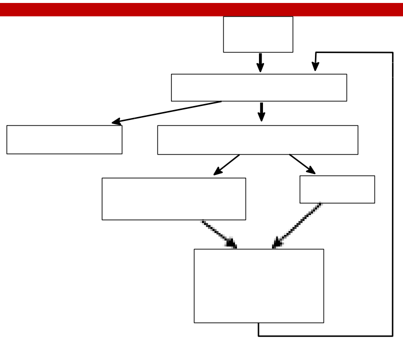
- Attackers: need to analyse our program to modify it!
- Defenders: need to analyse our program to protect it!
- Two kinds of analyses:

1. *static analysis tools* collect information about a program by studying its code;

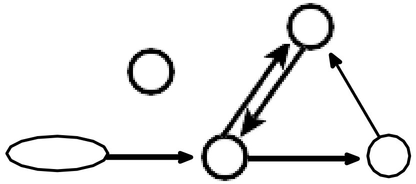
2. *dynamic analysis tools* collect information from executing the program.

Static Analysis

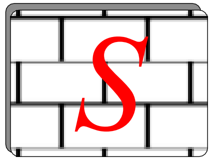
```
add r1,r2
mov r2,2(r3)
jmp L3
```



```
while (...){
   x=x+y;
   A[v]=x
}
```



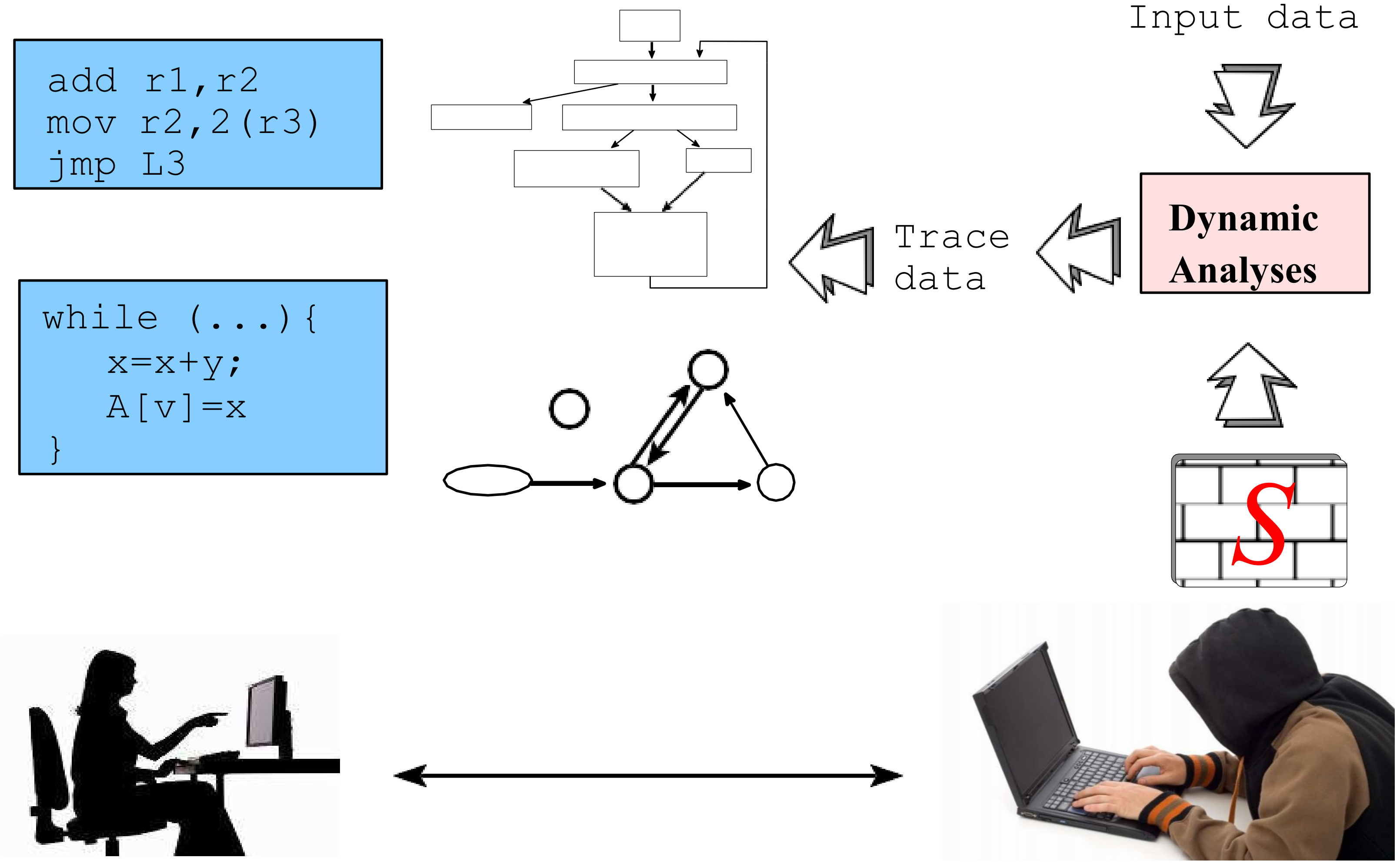
Static Analyses



Static Analyses

- Control-Flow Graphs
  - representation of (possible) control-flow in functions.
- Call graphs
  - representation of (possible) function calls.
- Disassembly
  - turn raw executables into assembly code.
- Decompilation
  - turn raw assembly code into source code.

Dynamic Analysis



Dynamic Analyses

Ca' Foscari University of Venice

- Debugging
  - What path does the program take?
- Tracing
  - Which functions/system calls get executed?
- Profiling
  - What gets executed the most?

Control-Flow Graphs

Credits: Christian Collberg

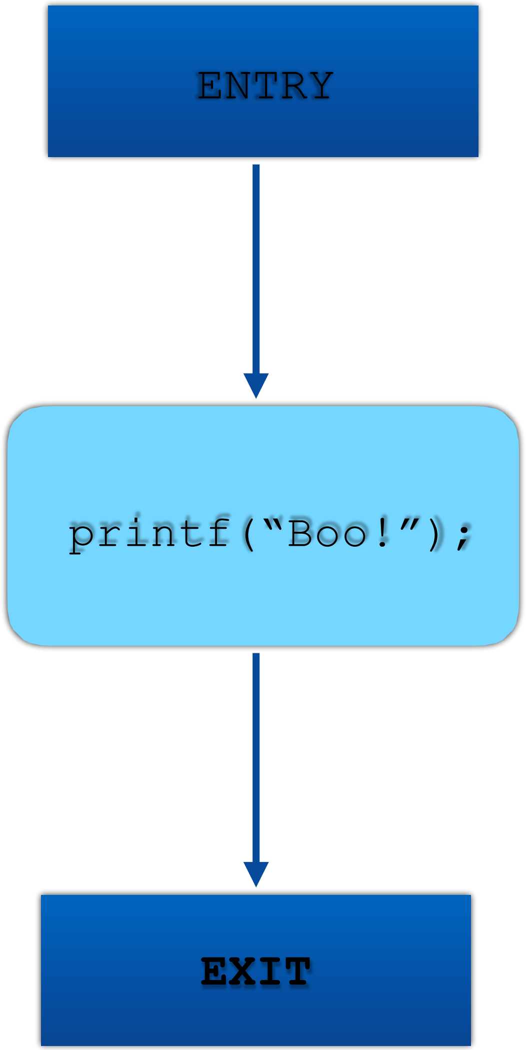
Control-Flow Graphs (CFGs)

Ca' Foscari University of Venice

- A way to represent the possible flow of control inside a function.
- Nodes are called basic blocks.
- Each block consists of straight-line code ending (possibly) in a branch.
- An edge $A \rightarrow B$: control could flow from A to B.
- There is one unique entry node and one unique exit node.

Example

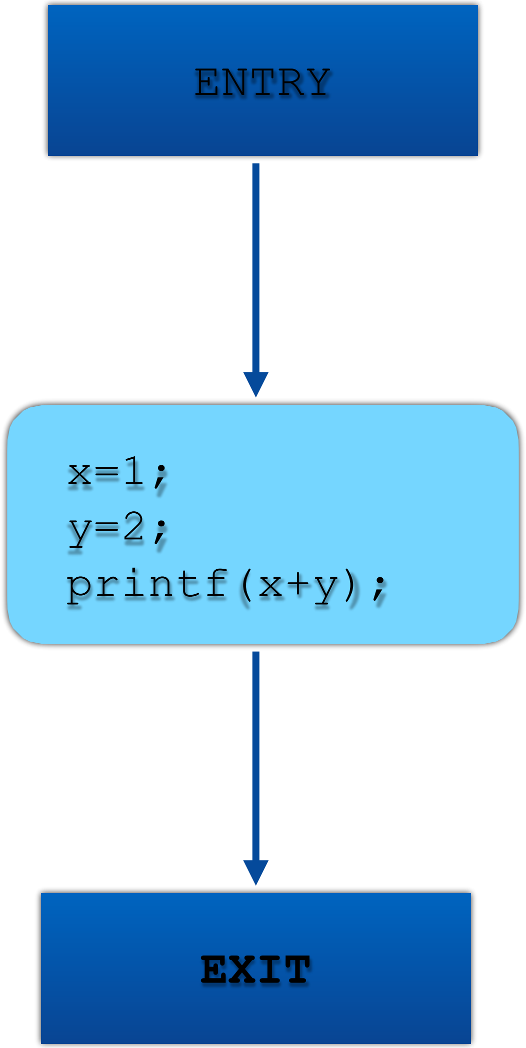
```
int foo() {
   printf("Boo!");
}
```



Ca' Foscari University of Venice

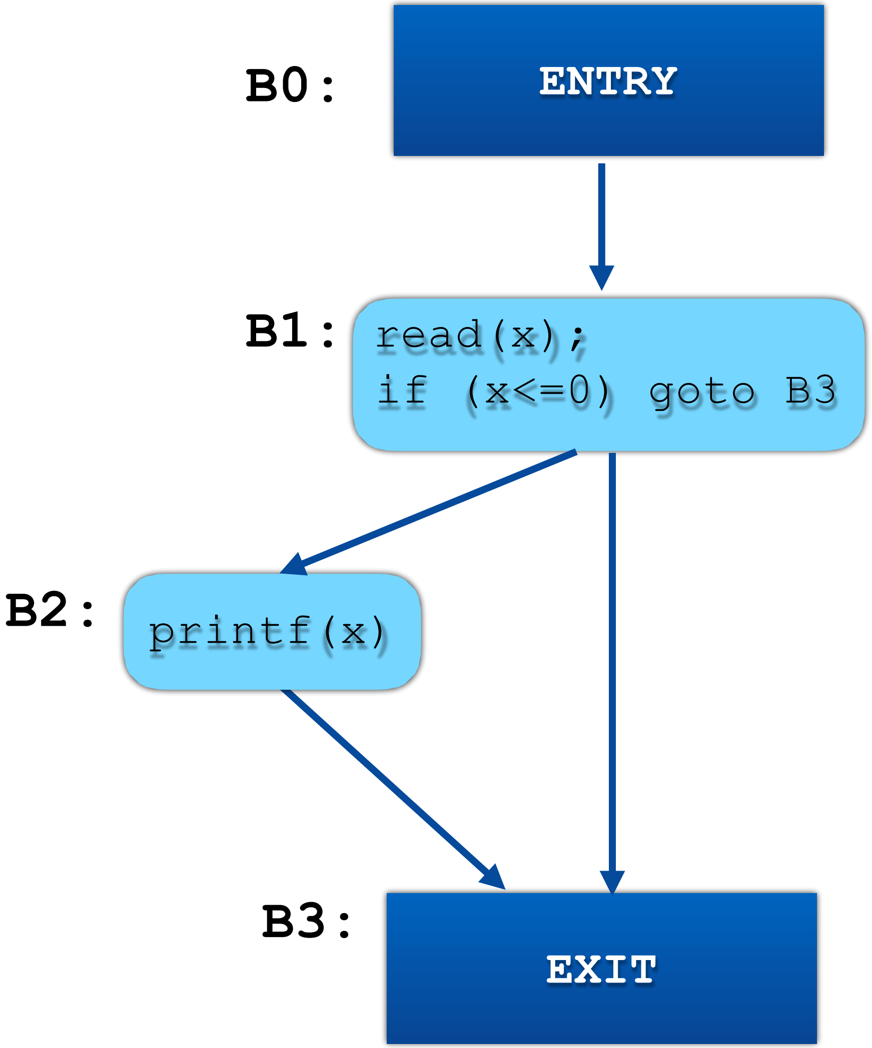
Example

```
int foo() {
    x=1;
    y=2;
    printf(x+y);
}
```



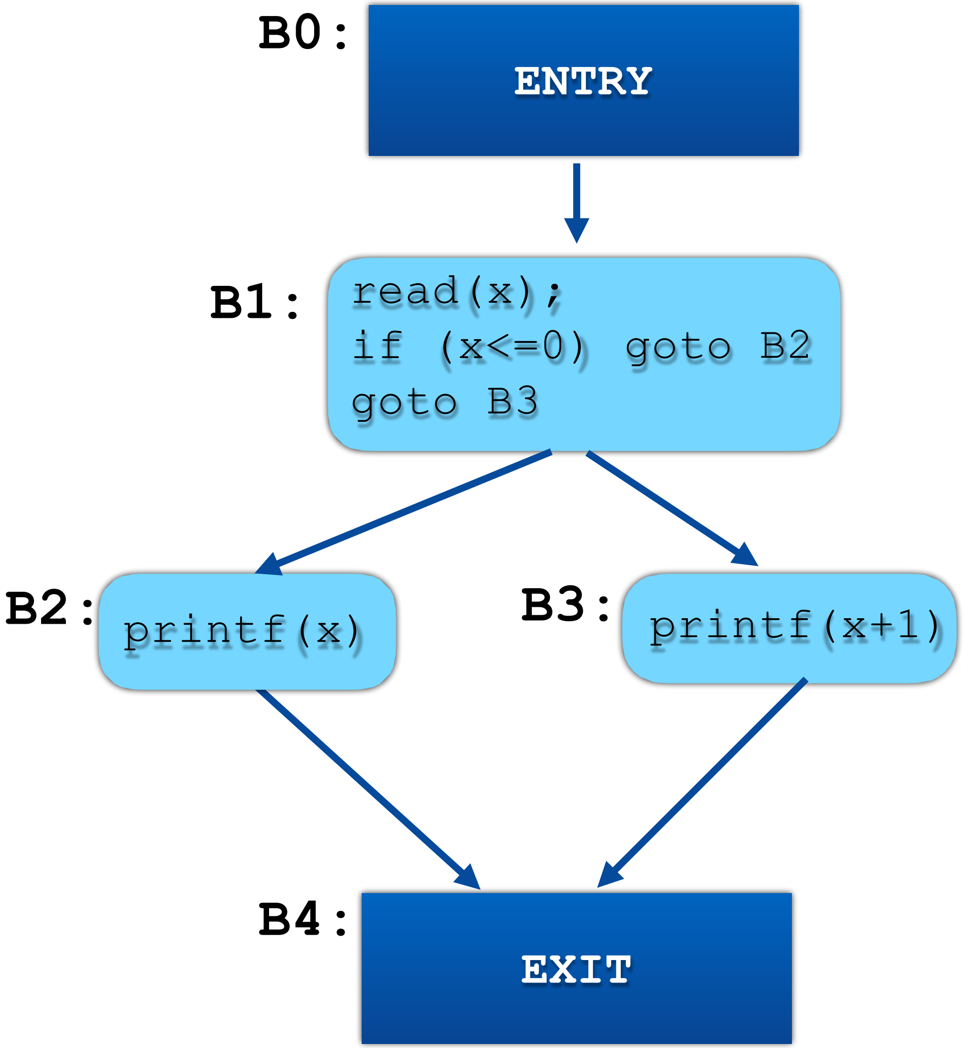
Example

```
int foo() {
  read(x);
  if (x>0)
      printf(x);
}
```



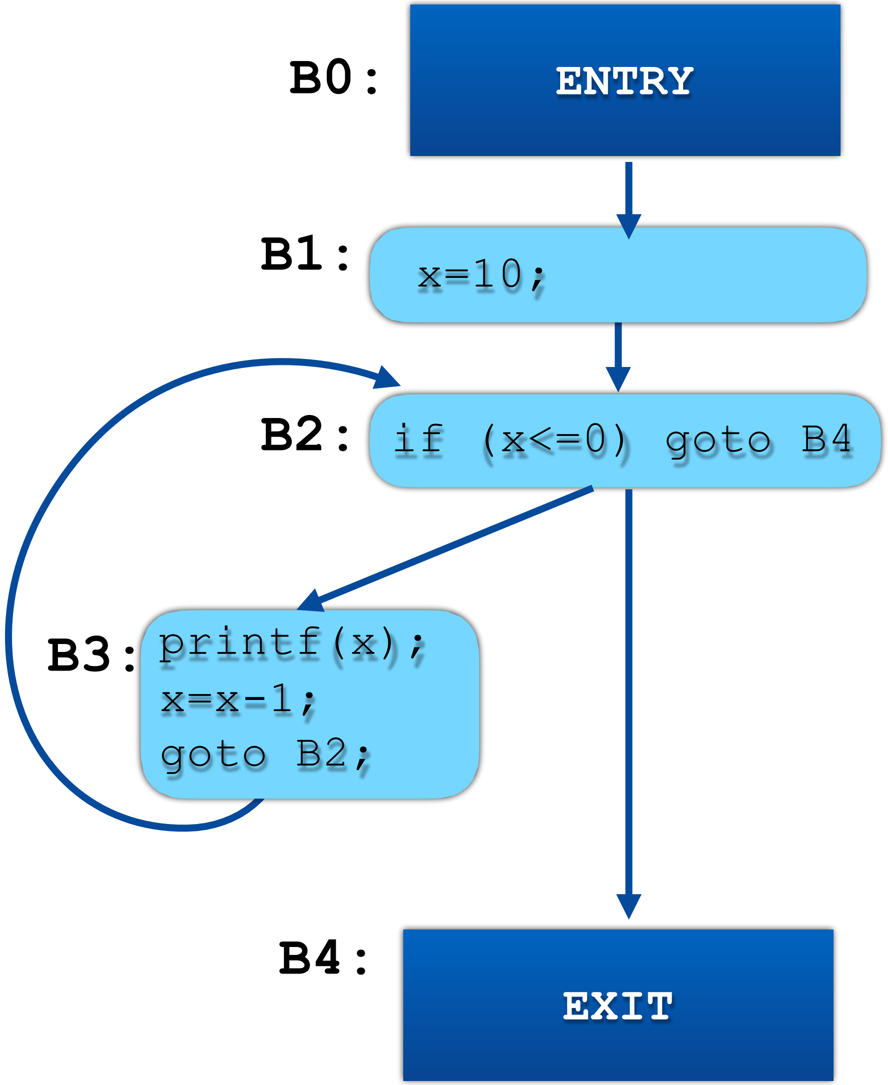
Example

```
int foo() {
  read(x);
  if (x>0)
    printf(x);
  else
    printf(x+1);
}
```



Example

```
int foo() {
  x=10;
  while (x>0){
    printf(x);
    x=x-1;
  }
}
```



Instruction Numbers

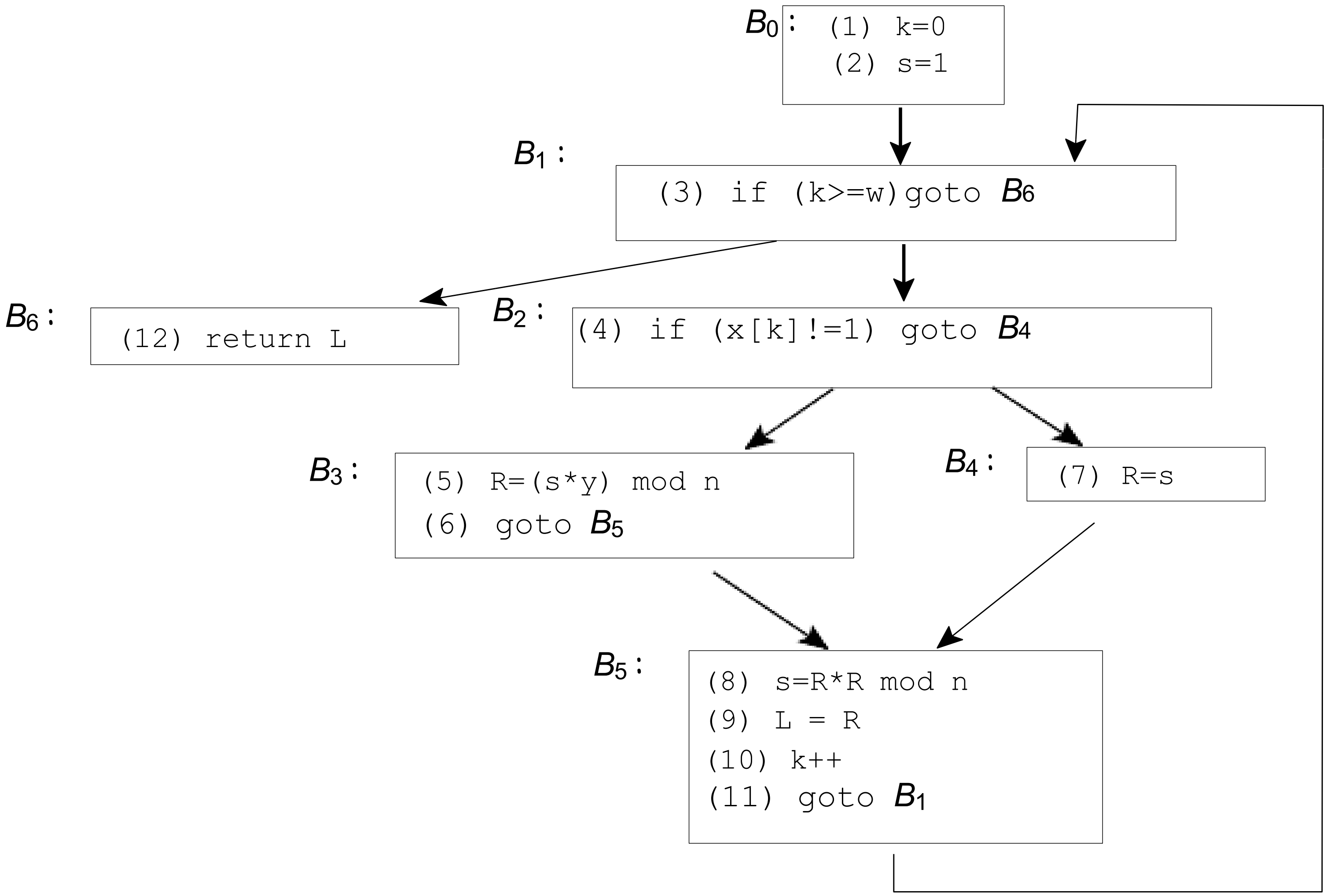
Ca' Foscari University of Venice

```
int modexp(int y,int x[],
int w, int n) {
 int R, L;

 int k = 0;
 int s = 1;
 while (k < w) {
        if (x[k] == 1)
              R = (s*y) % n;
        else
        R = s;
        s = R*R % n;
        L = R;
        k++;
 }
 return L
 }
```

```
(1) k=0
(2) s=1
(3) if (k>=w) goto (12)
(4) if (x[k]!=1) goto (7)
(5) R=(s*y)%n
(6) goto (8)
(7) R= s
(8) s= R* R%n
(9) L= R
(10) k++
(11) goto (3)
(12) return L
```

The resulting graph



Create Three-Address Statements

Ca' Foscari University of Venice

Transform the function into a sequence of simpler statements:

```
x = y + z
if (x < y) goto L
goto L
```

These are called three-address statements.

Build the Graph

Ca' Foscari University of Venice

BUILDCFG(F):

1. Mark every instruction which can start a basic block as a *leader*:
   - the first instruction is a leader; any target of a branch is a leader;
   - the instruction following a conditional branch is a leader.

2. A basic block consists of the instructions from a leader up to, but not including, the next leader.

3. Add an edge $A \rightarrow B$ if A ends with a branch to B or can fall through to B.

In-Class Exercise 1

Ca' Foscari University of Venice

- Turn this function into a sequence of three-address statements.
- Turn the sequence of simplified statements into a CFG.

```
int gcd(int x, int y) {
    int temp;
    while (true) {
        if (x%y == 0) break;
        temp = x%y;
        x = y;
        y = temp;
    }
}
```

In-Class Exercise 2

- Construct the corresponding CFG.

```
X := 20;
WHILE X < 10 DO  X := X-1;
    A[X] := 10;
    IF X = 4
        THEN X := X - 2;
    ENDIF;
ENDDO;
Y := X + 5
```

```
(1)    X := 20
(2)    if X>=10 goto (8)
(3)    X := X-1
(4)    A[X] := 10
(5)    if X<>4 goto (7)
(6)    X := X-2
(7)    goto (2)
(8)    Y := X+5
```

Call Graphs

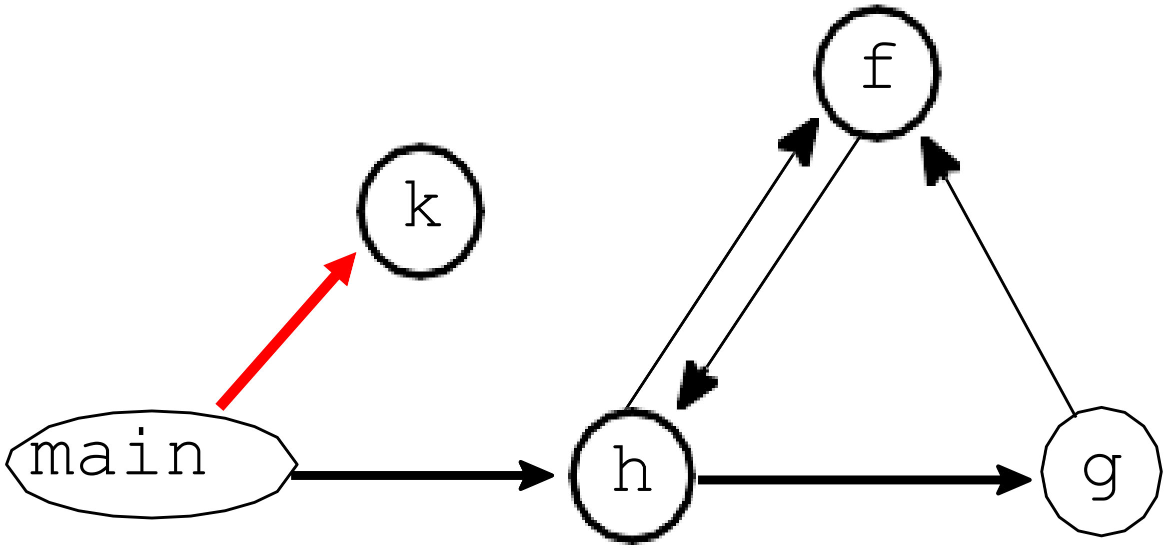
Inter-procedural control flow

- Inter-procedural analysis also considers flow of information between functions.
- Call graphs are a way to represent possible function calls
- Each node represents a function.
- An edge A → B: A might call B.

Building call-graphs

```c
void h();
void f(){ h(); }
void g(){ f(); }
void h() { f(); g(); }
void k() {}

 int main() {
    h();
    void (*p)() = &k;
    p();
 }
```



In-Class Exercise

Ca' Foscari University of Venice

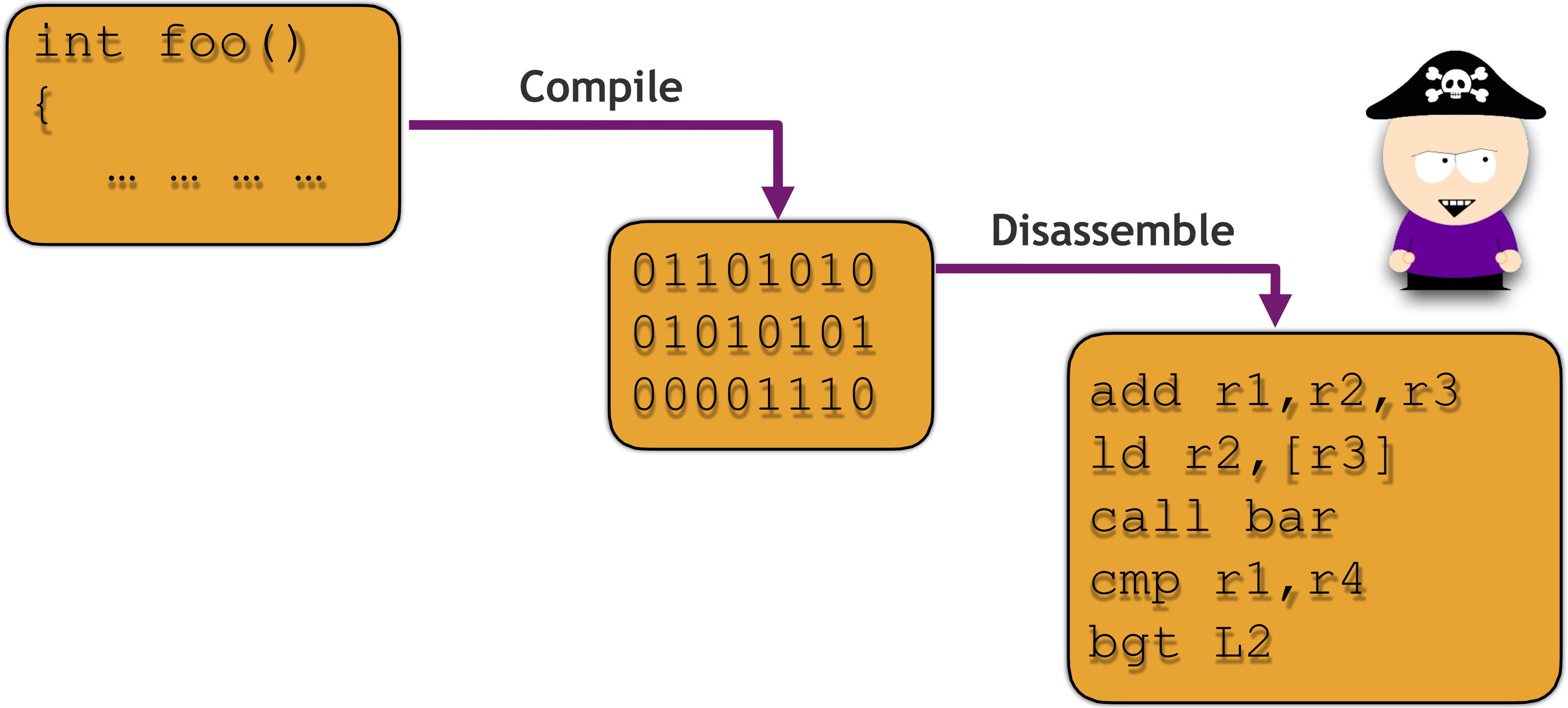
- Build the call graph for this Java program

```
class M {
    public void a () {System.out.println("hello");}
    public void b () {}
    public void c () {System.out.println("world!")}
}
class N extends M {
    public void a () {super.a();}
    public void b () {this.b(); this.c();}
    public void c () {}
}
class Main {
    public static void main (String args[]) {
        M x = (args.length > 0)? new M() : new N();
        x.a();
        M y = new N(); y.b();
    }
}
```

Disassembly

Disassembly

Attackers



Disassembly

```
100000d78:   55                  push   %rbp
100000d79:   48 89 e5            mov    %rsp,%rbp
100000d7c:   48 83 c7 68         add    $0x68,%rdi
100000d80:   48 83 c6 68         add    $0x68,%rsi
100000d84:   5d                  pop    %rbp
100000d85:   e9 26 38 00 00      jmpq   1000045b0
100000d8a:   55                  push   %rbp
100000d8b:   48 89 e5            mov    %rsp,%rbp
100000d8e:   48 8d 46 68         lea    0x68(%rsi),%rax
100000d92:   48 8d 77 68         lea    0x68(%rdi),%rsi
100000d96:   48 89 c7            mov    %rax,%rdi
100000d99:   5d                  pop    %rbp
100000d9a:   e9 11 38 00 00      jmpq   1000045b0
100000d9f:   55                  push   %rbp
```

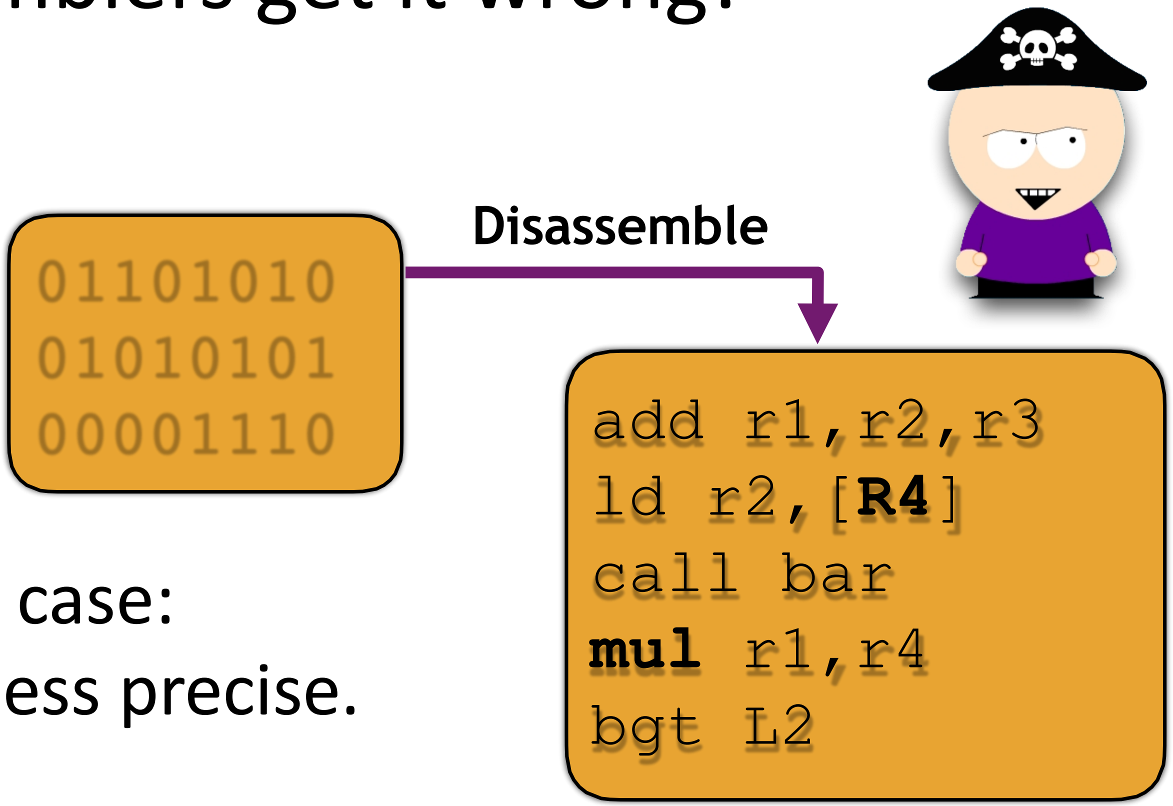
Disassembly

	Address	Code bytes	Assembly
1.	100000d78:	55	push %rbp
2.	100000d79:	48 89 e5	mov %rsp,%rbp
3.	100000d7c:	48 83 c7 68	add $0x68,%rdi
4.	100000d80:	48 83 c6 68	add $0x68,%rsi
5.	100000d84:	5d	pop %rbp
6.	100000d85:	e9 26 38 00 00	jmpq 1000045b0
7.	100000d8a:	55	push %rbp
8.	100000d8b:	48 89 e5	mov %rsp,%rbp
9.	100000d8e:	48 8d 46 68	lea 0x68(%rsi),%rax
10.	100000d92:	48 8d 77 68	lea 0x68(%rdi),%rsi
11.	100000d96:	48 89 c7	mov %rax,%rdi
12.	100000d99:	5d	pop %rbp
13.	100000d9a:	e9 11 38 00 00	jmpq 1000045b0
14.	100000d9f:	55	push %rbp

Disassembly is hard!

Ca' Foscari University of Venice

- And sometimes disassemblers get it wrong!



- In general, this is always the case: program analysis is more or less precise.

Linear Sweep Disassembly Algorithm

Ca' Foscari University of Venice

- Takes a very straightforward approach to locating instructions to disassemble: where one instruction ends, another begins.
- Most difficult decisions faced are where to begin and when to stop.
- The usual solution is to assume that everything contained in sections of a program marked as code (typically specified by the program file's headers) represents machine language instructions.
- Disassembly begins with the first byte in a code section and moves, in a linear fashion, through the section, disassembling one instruction after another until the end of the section is reached.
- No effort is made to understand the program's control flow through recognition of nonlinear instructions such as branches

Linear Sweep Disassembly Algorithm

Ca' Foscari University of Venice

- The length of each instruction is computed and used to determine the location of the next instruction to be disassembled.
- Instruction sets with fixed-length instructions (MIPS, for example) are somewhat easier to disassemble, as locating subsequent instructions is straightforward.
- The main advantage of the linear sweep algorithm is that it provides complete coverage of a program's code sections.
- One of the main disadvantages is that it can confuse data with code.
- Linear sweep is used by the disassembly engines contained in the GNU debugger (gdb), Microsoft's WinDbg debugger, and the objdump utility.

Linear Sweep Algorithm Errors

- This function contains a switch statement, implemented with a jump table to resolve case label targets.
- The jmp statement references an address table , but the disassembler treats the address table as if it were a series of instructions.
- If we treat successive 4-byte groups in the jump table as little-endian values, we see that each represents a pointer to a nearby destination address for one of the various jumps (004012e0, 0040128b, 00401290).
- Thus, the loopne instruction is not an instruction at all, but a failure of the linear sweep algorithm to properly distinguish embedded data from code.

40123f: 55	push ebp
401240: 8b ec	mov ebp,esp
401242: 33 c0	xor eax,eax
401244: 8b 55 08	mov edx,DWORD PTR [ebp+8]
401247: 83 fa 0c	cmp edx,0xc
40124a: 0f 87 90 00 00 00	ja 0x4012e0
401250: ff 24 95 57 12 40 00	**jmp DWORD PTR [edx*4+0x401257]**
401257: e0 12	**loopne 0x40126b**
401259: **40**	inc eax
40125a: **00 8b 12 40 00 90**	add BYTE PTR [ebx-0x6fffbfee],cl
401260: **12 40 00**	adc al,BYTE PTR [eax]

Recursive Descent Disassembly

Ca' Foscari University of Venice

- The recursive descent disassembly algorithm focuses on the concept of control flow
- It determines whether an instruction should be disassembled based on whether it is referenced by another instruction.
- To understand recursive descent, it is helpful to classify instructions according to how they affect the instruction pointer.
- One of the principle advantages of the recursive descent algorithm is its superior ability to distinguish code from data.

Recursive Descent Disassembly issues

Ca' Foscari University of Venice

- A recursive descent disassembler attempts to determine the target of the unconditional jump and continues disassembly at the target address
- It does not guarantee complete code coverage.
- Unfortunately, when the target of a jump instruction depends on a runtime value, it may not be possible to determine the destination of the jump by using static analysis.
- The x86 instruction jmp rax demonstrates this problem.
  - The rax register contains a value only when the program is actually running
  - Since the register contains no value during static analysis, we have no way to determine the target of the jump instruction
  - Thus, we have no way to determine where to continue the disassembly process

Why is disassembly hard. . . ?

- Variable length instruction sets: overlapping instructions.
- Mixing data and code: misclassify data as instructions.
- Indirect jumps: must assume that any location could be the start of an instruction!
- Find the beginning of functions if all calls are indirect.

Why is disassembly hard. . . ?

- Finding the end of functions: if no dedicated return instruction.
- Handwritten assembly code: it will not conform to the standard calling conventions.
- Code compression: the code of two functions may overlap.
- Self-modifying code.

Decompilation

Decompilation

p.c

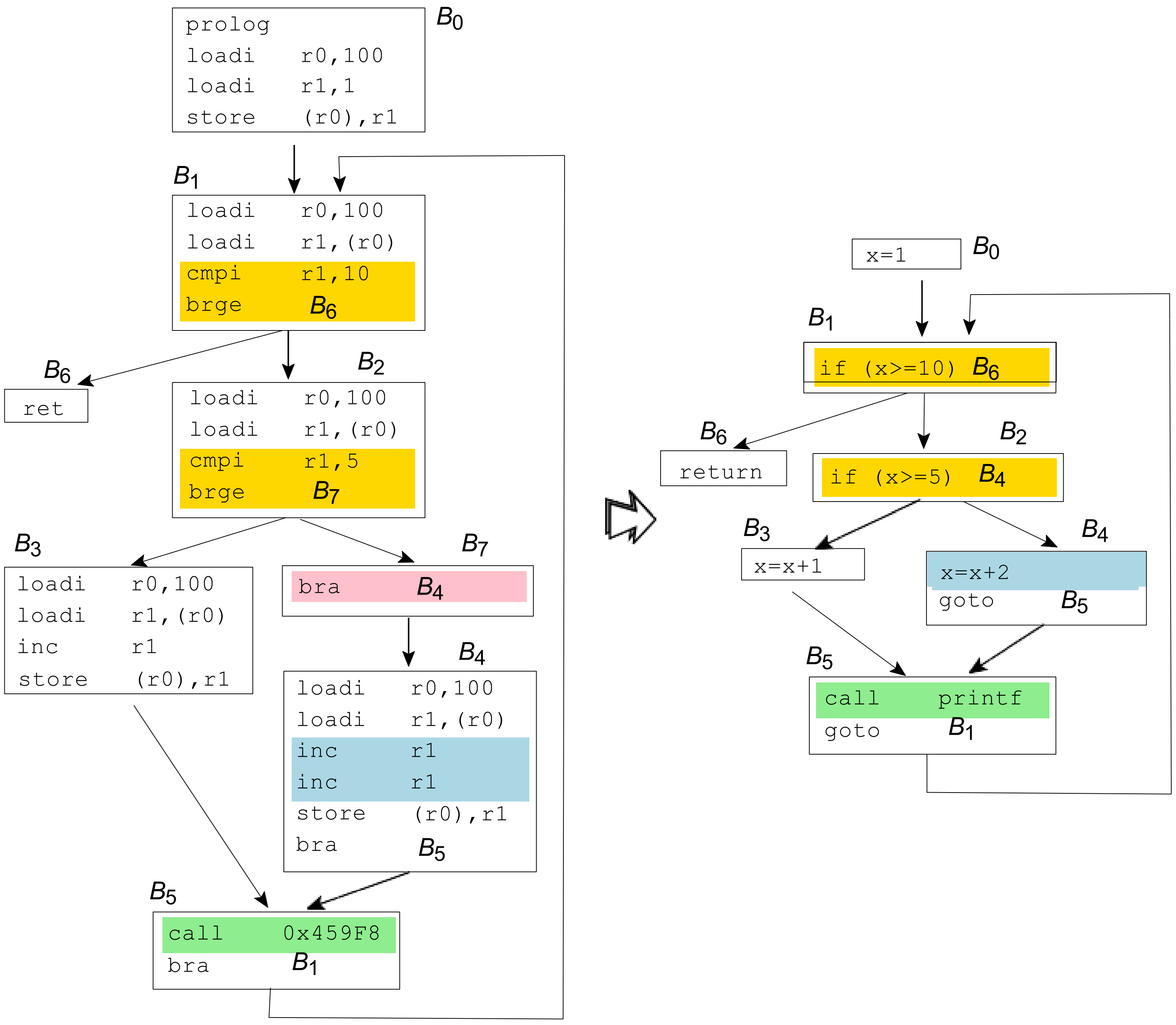
```
while (...){
    a += b;
    Q();
}
```

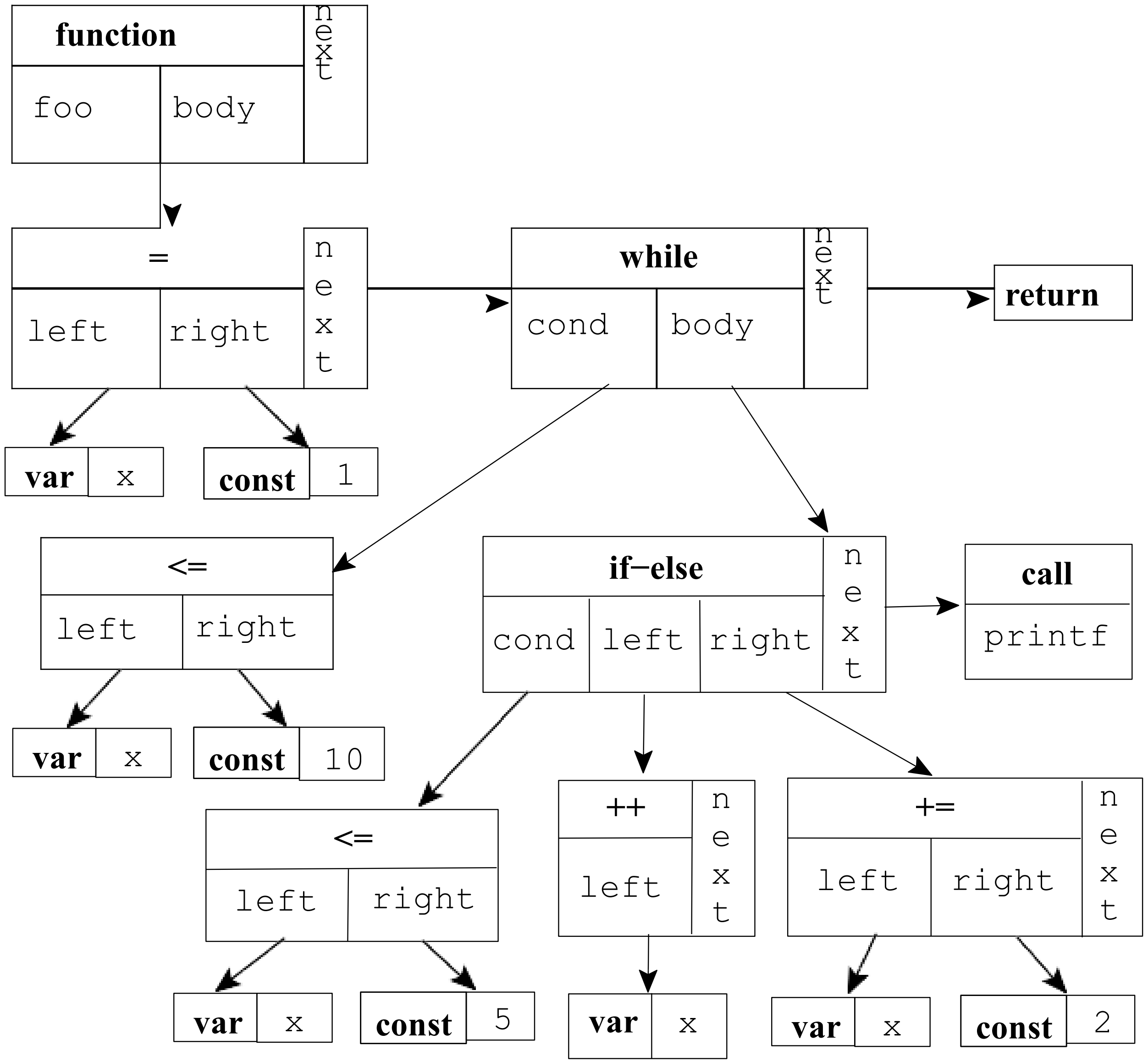
p.as

```
mov  R1,R2
add  R2,R2
jmp L3
L4: call Q
```

p.exe

```
6 9 19 245
99 23 255
50 89 88 90
1 23 45 71
```


prolog
loadi r0,100
loadi r1,1
store (r0),r1
B0
B1
loadi r0,100
loadi r1,(r0)
cmpi r1,10
brge B6
B6
ret
B2
loadi r0,100
loadi r1,(r0)
cmpi r1,5
brge B7
B3
loadi r0,100
loadi r1,(r0)
inc r1
store (r0),r1
B7
bra B4
B4
loadi r0,100
loadi r1,(r0)
inc r1
inc r1
store (r0),r1
bra B5
B5
call 0x459F8
bra B1
x=1
B0
B1
if (x>=10) B6
B6
return
B2
if (x>=5) B4
B3
x=x+1
B4
x=x+2
goto B5
B5
call printf
goto B1




```
void foo() {
   x=1;
   while (x<10} {
      if (x<5)
            x++;
      else
            x+=2;
      printf();
   }
}
```

What’s so hard about decompilation?

Ca’ Foscari University of Venice

- Disassembly: first step of any Decompiler!
- Target language: assembly code may not correspond to any legal source code.
- Standard library functions:
  - call printf() ⇒ call foo96()
- Idioms of different compilers (xor r0, r0 ⇒ r0=0)

What’s so hard about decompilation?

Ca’ Foscari University of Venice

- Artifacts of the target architecture
  - unnecessary jumps-to-jumps.
- Structured control-flow: from mess of machine code branches.
- Compiler optimizations: undo loop, unrolling, shifts and adds ⇒ original multiplication by a constant.
- Loads/stores ⇒ operations on arrays, records, pointers, and objects.

Conclusions

Ca' Foscari University of Venice

- Different Reverse Engineering tools implements their own variations of disassembler algorithms
- Static Analysis cannot be perfect as some assembly instructions require run-time information
- Mixed code-data and variable length instructions of CISC architectures (e.g. x86) make disassembly harder than in RISC architectures (e.g. ARM) where instructions have fixed length