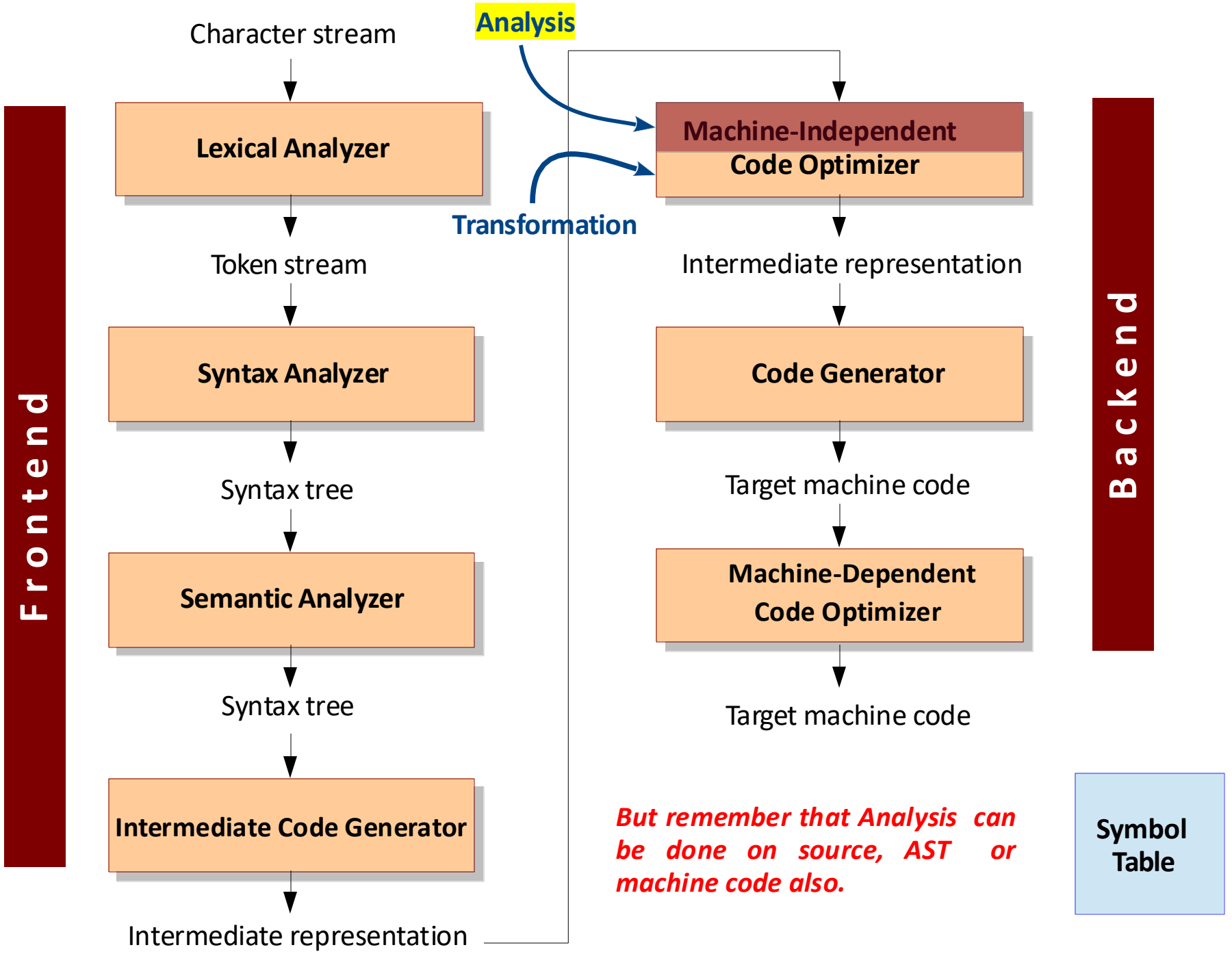
CS202: Software Tools and Techniques for CSE

Lecture 6

Shouvick Mondal

shouvick.mondal@iitgn.ac.in
August 2025


Analysis
Character stream
Lexical Analyzer
Token stream
Syntax Analyzer
Syntax tree
Semantic Analyzer
Syntax tree
Intermediate Code Generator
Intermediate representation
Transformation
Machine-Independent
Code Optimizer
Intermediate representation
Code Generator
Target machine code
Machine-Dependent
Code Optimizer
Target machine code
Frontend
Backend
But remember that Analysis can be done on source, AST or machine code also.
Symbol Table

Static Program Analysis

```
//filename: fact.c
#include <stdio.h>
int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
int main()
{
    int n;
    printf("Enter a non-negative integer: ");
    scanf("%d", &n);
    printf("The factorial of %d is %d.\n", n, factorial(n));
    return 0;
}
```

Show the function call graph of the C code in textual mode and graphical mode.

```c
//filename: fact.c
#include <stdio.h>
int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
int main()
{
    int n;
    printf("Enter a non-negative integer: ");
    scanf("%d", &n);
    printf("The factorial of %d is %d.\n", n, factorial(n));
    return 0;
}
```

[**Textual mode**] (generate function call graph)

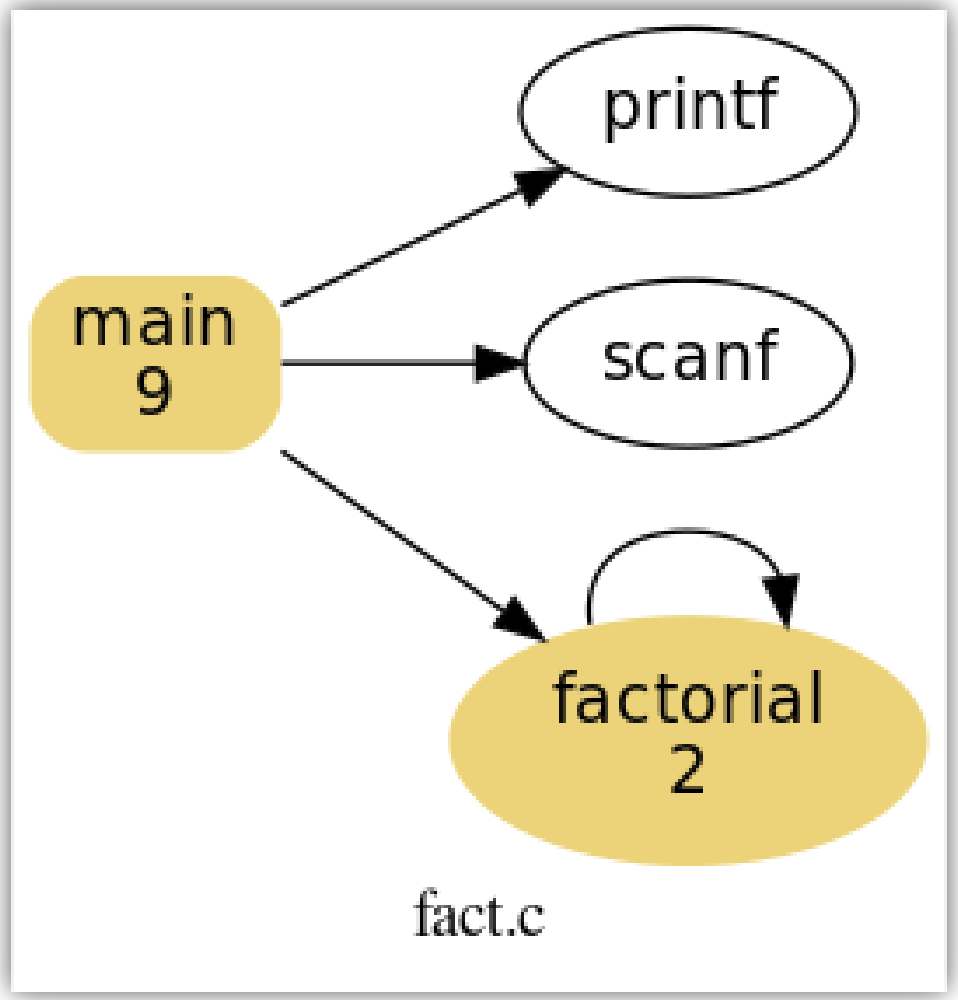
```
cflow fact.c
main() <int main () at fact.c:9>:
    printf()
    scanf()
    factorial() <int factorial (int n) at fact.c:2> (R):
        factorial() <int factorial (int n) at fact.c:2> (recursive: see 4)
```

```c
//filename: fact.c
#include <stdio.h>
int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
int main()
{
    int n;
    printf("Enter a non-negative integer: ");
    scanf("%d", &n);
    printf("The factorial of %d is %d.\n", n, factorial(n));
    return 0;
}
```

[**Graphical mode**] (generate function call graph)

```
cflow2dot -i fact.c
```

```
found cflow at: /usr/bin/cflow
found dot at: /usr/bin/dot
This may take some time... ...
svg produced successfully from dot.
```



Static Program Analysis

```
//filename: fact.c
#include <stdio.h>
int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
int main()
{
    int n;
    printf("Enter a non-negative integer: ");
    scanf("%d", &n);
    printf("The factorial of %d is %d.\n", n, factorial(n));
    return 0;
}
```

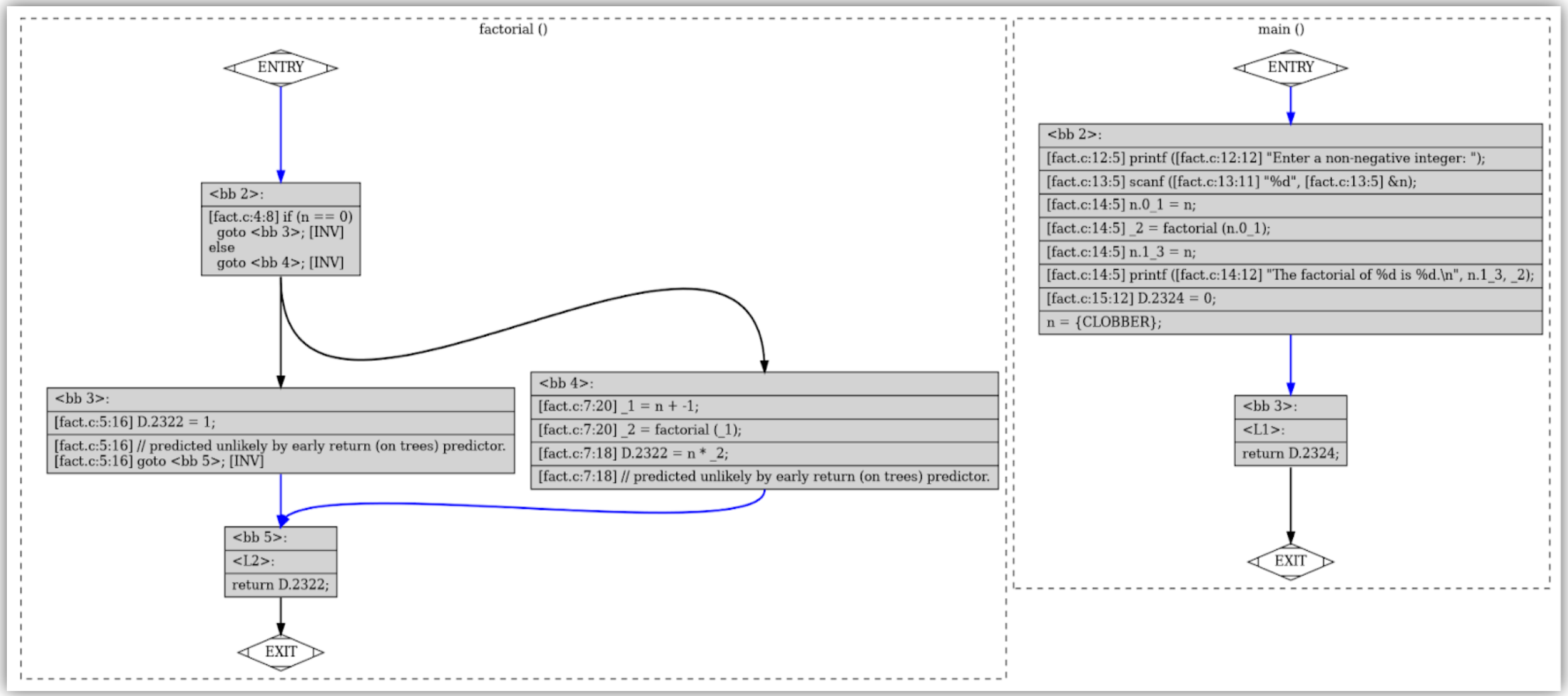
Show the control flow graph of the C code.

Step 1 [Generate control flow graph]

```
gcc -fdump-tree-cfg-graph-lineno fact.c
```

Step 2 [Convert .dot to .png]

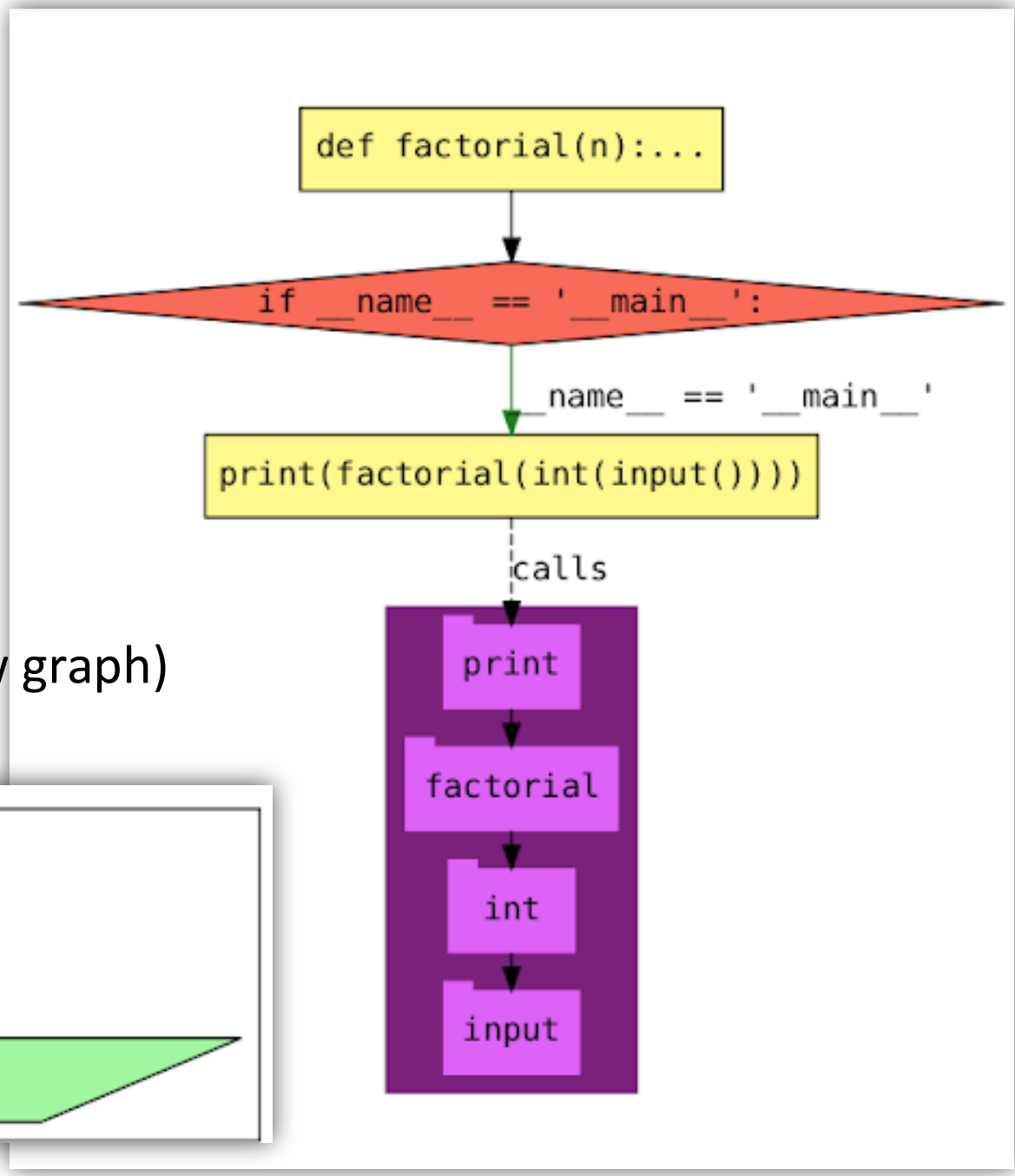
```
dot -Tpng fact.c.012t.cfg.dot -o fact.c.png
```



This is intra-procedural Control Flow Graph (CFG). What is 'intra'/'inter' here?

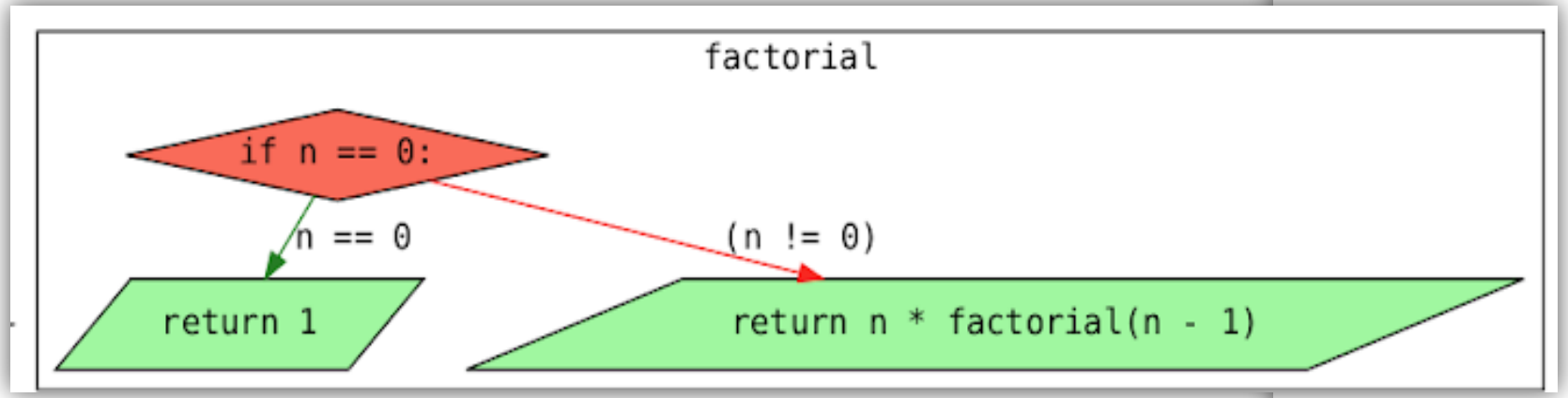
The Pythonic way...

```
#filename: fact.py
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
if __name__ == "__main__":
    print(factorial(int(input())))
```



[**Graphical mode**] (generate extended control flow graph)

`py2cfg fact.py`



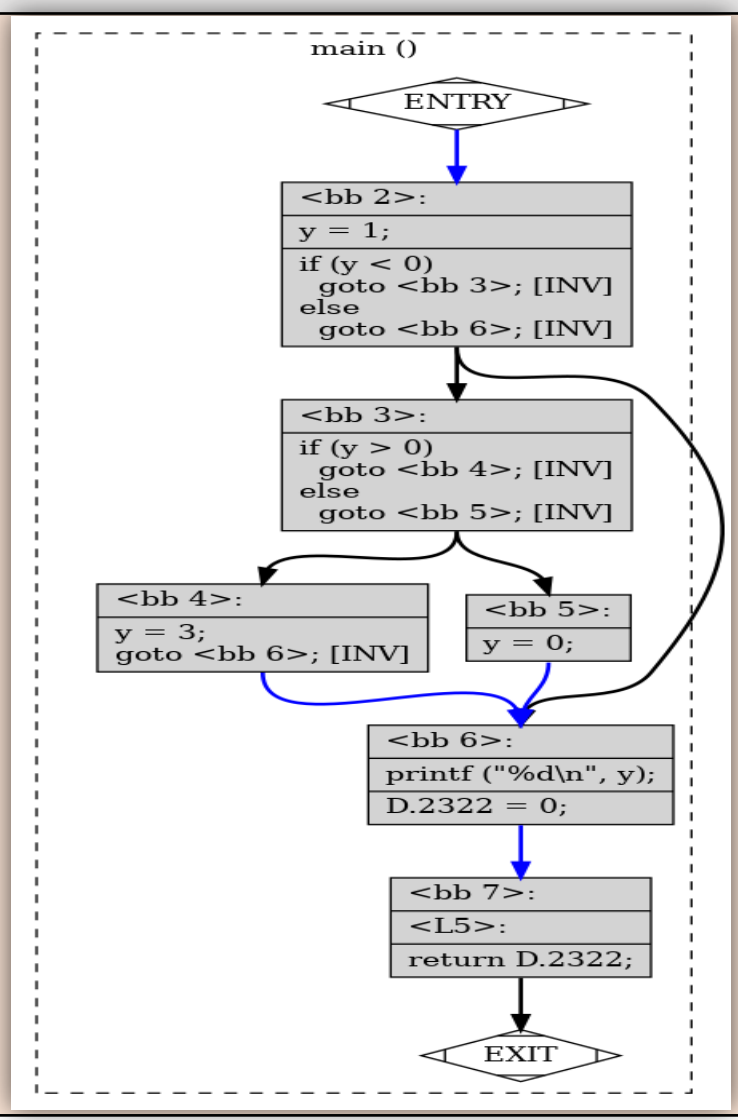
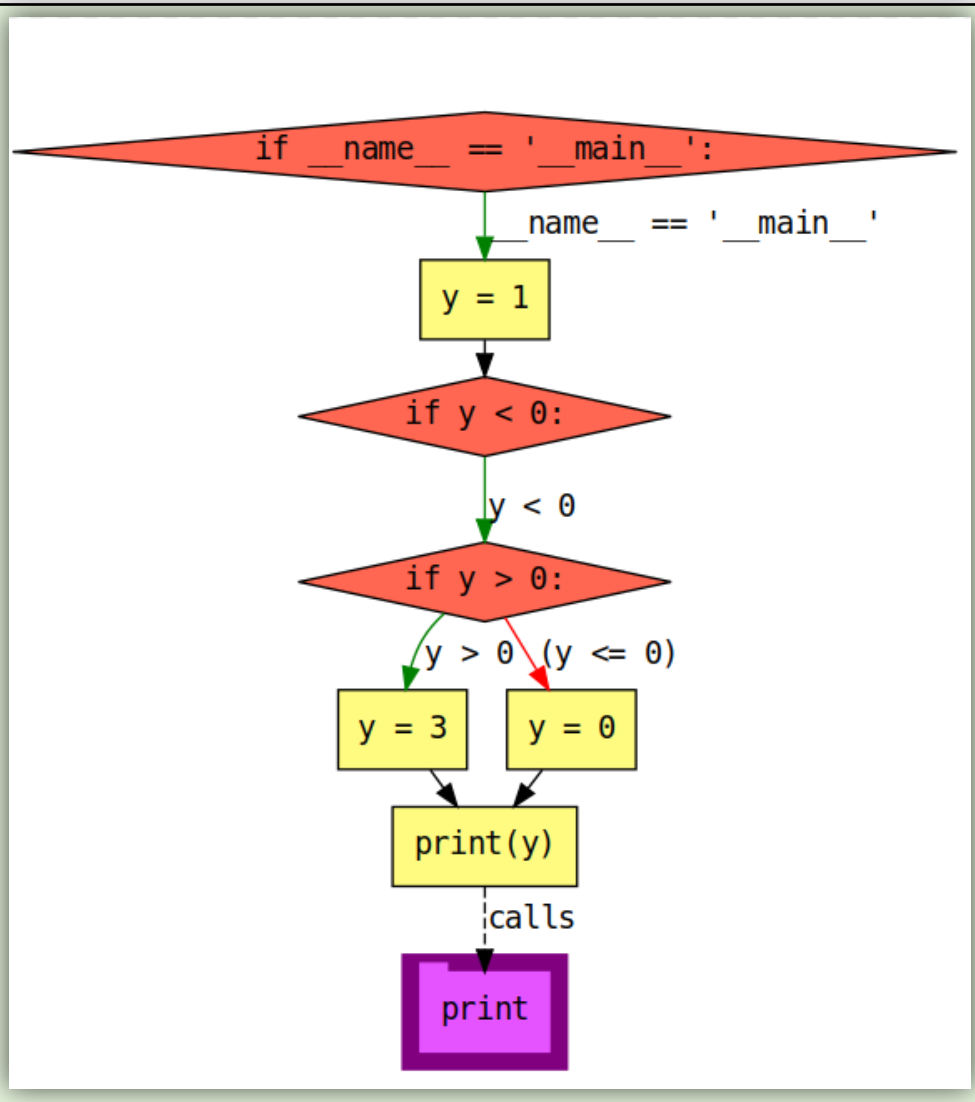
Classwork #1: Draw the CFG for…

C	Python

`#include<stdio.h>`
`int main()`
`{`
`int y=1;`
`if(y<0)`
`if (y>0) y=3;`
`else y=0;`
`printf("%d\n",y);`
`}`
	`if __name__ == "__main__":`
	`y=1;`
	`if(y<0):`
	`if (y>0): y=3;`
	`else: y=0;`
	`print(y);`

Same or different? If yes, why, If not, why not?

Classwork: Draw the CFG for the following…

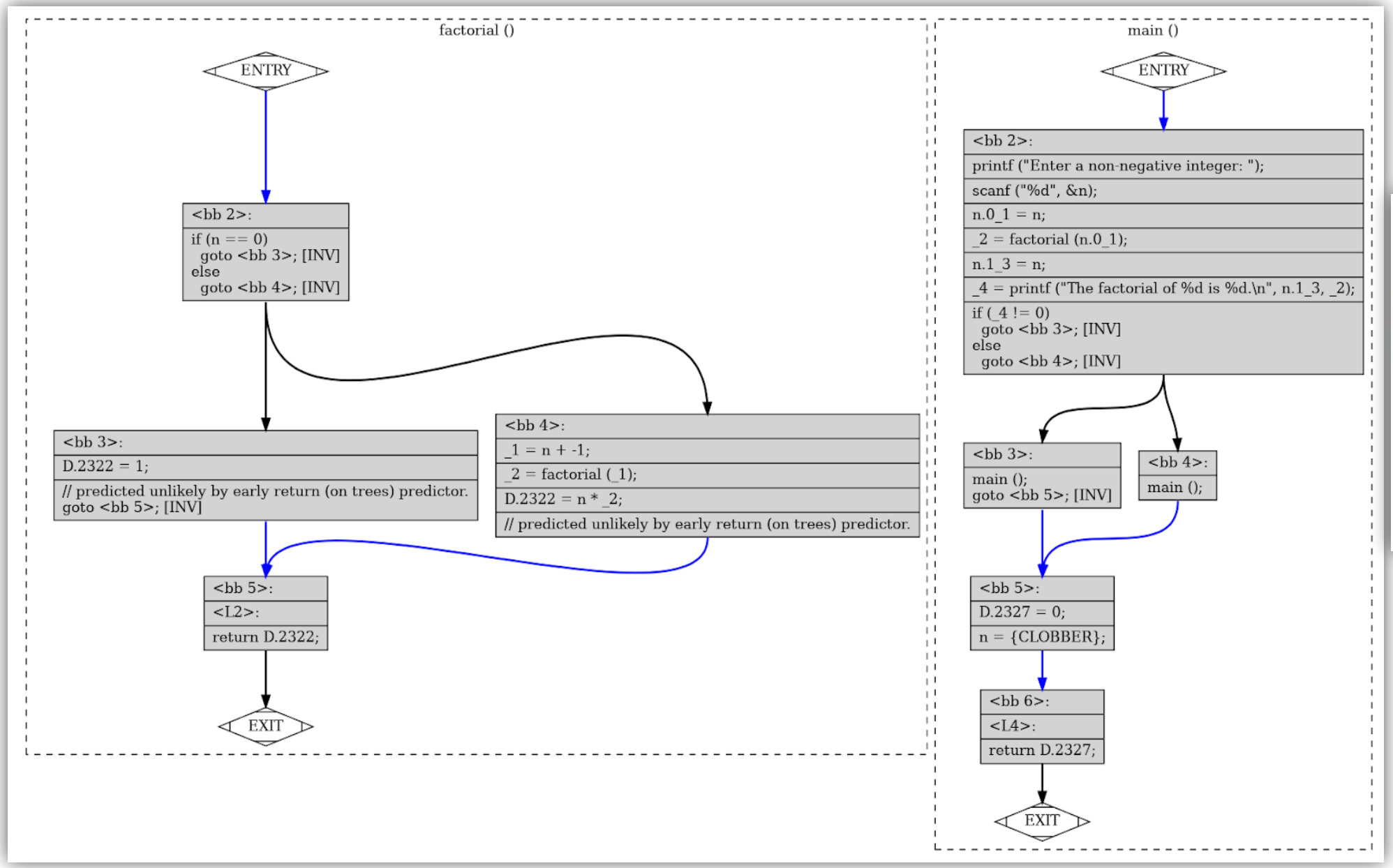
C	Python
	

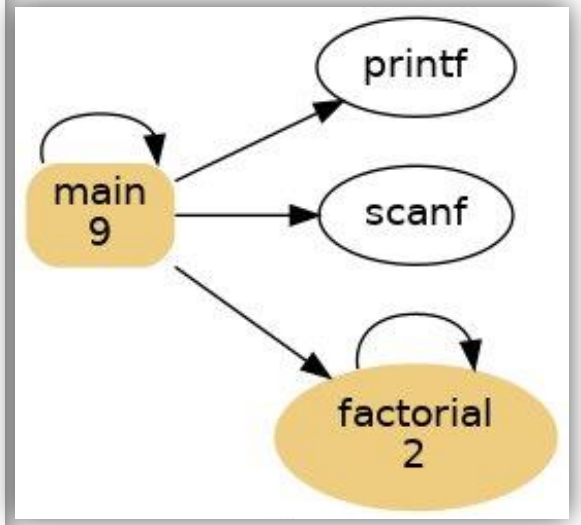
Classwork #2: Draw the CFG for…

```
#include <stdio.h>
int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
int main()
{
    int n;
    printf("Enter a non-negative integer: ");
    scanf("%d", &n);
    if(printf("The factorial of %d is %d.\n", n, factorial(n)))
    {
        main();
    }
    else
        main();
    return 0;
}
```

What will be the call graph like?

Classwork #2: Draw the CFG for…





Classwork #3: Algebraically derive the cyclomatic complexity

```c
#include <stdio.h>
int main()
{
   int n;
   scanf("%d", &n);
   if(n-1)
   {
      printf("%d\n",n-1);
      if(n-2)
      {
         printf("%d\n",n-2);
         if(n-3)
         {
            printf("%d\n",n-3);
            if(n-4)
            {
               printf("%d\n",n-4);
               if(n-5)
               {
                  printf("%d\n",n-5);
               }
            }
         }
      }
    }
    return(0);
}
```

Algebraic

Domain specific recurrence

cc(n)=1+cc(n-1)

=1+1+cc(n-2)

…

=5+cc(n-5)

=5+1 = **6**

Therefore,

cc(main)=6

Sanity check #1

cc(main)

= #regions in CFG

= 5 + 1 = **6**

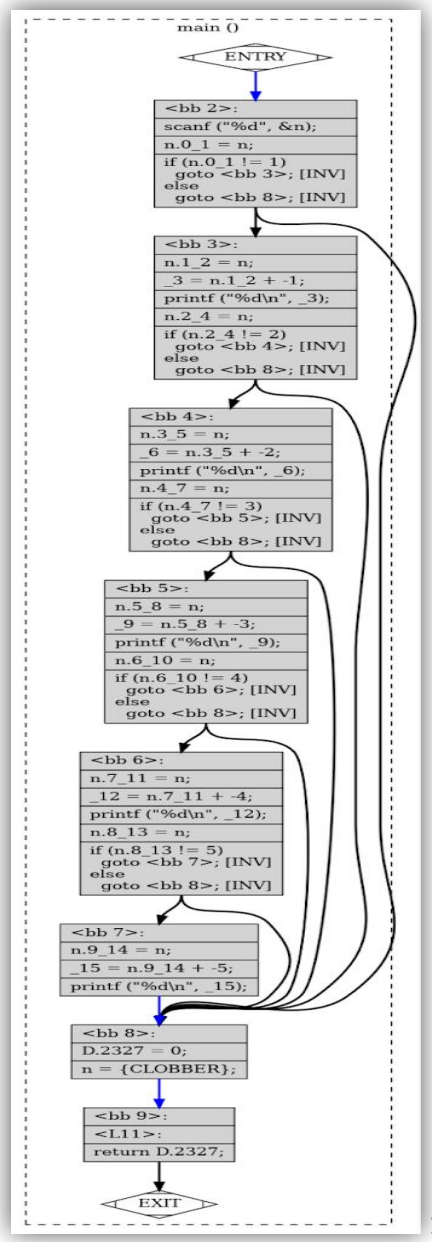
Sanity check #2

cc(main)

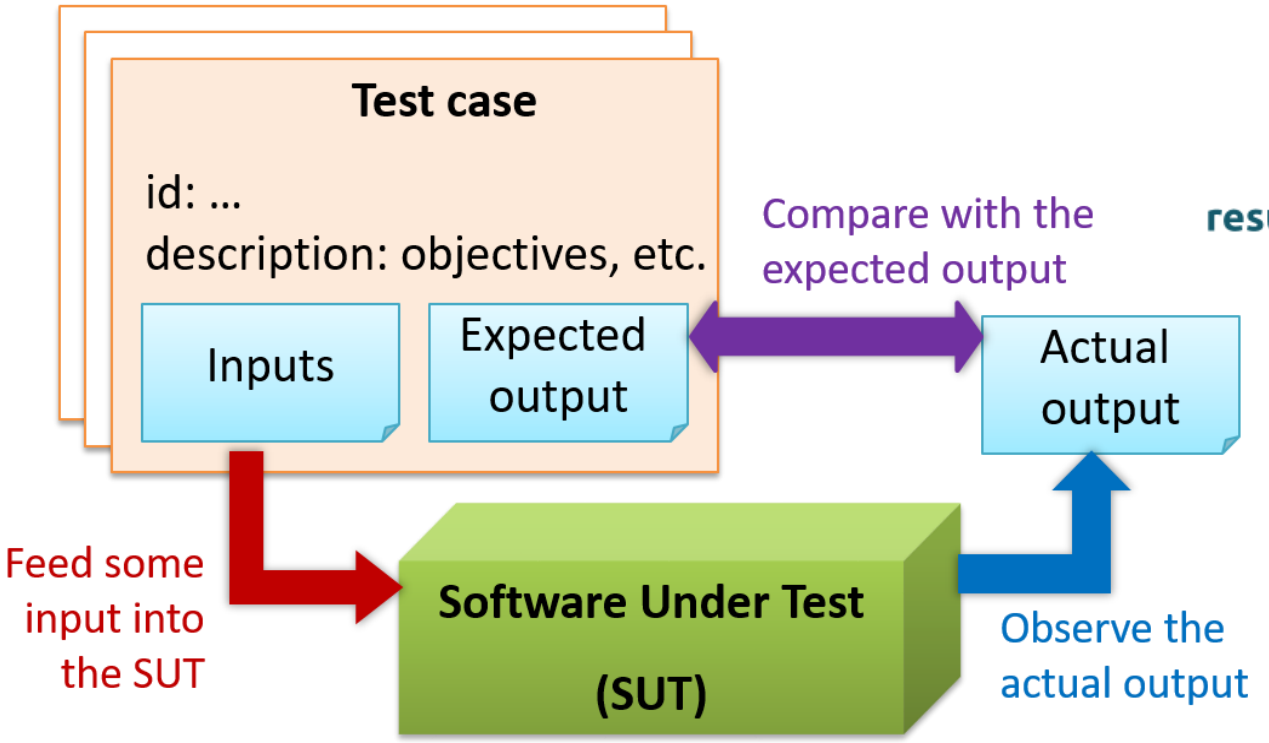
= π + 1

= 5 + 1 = **6**

π is the #decision nodes (here ifs)



How do we test for software bugs? **Test cases**



Test case: a scenario/use case/input which **executes** the software to *observe some interesting events*.

Example of testing-and-debugging a C program (for division): *version **0** with **4** test cases*

```c
//original program
#include<stdio.h>
int main(void)
{
    int x,y;
    float res;
    scanf("%d %d",&x, &y);

    res=x/y;
    printf("%d/%d =%.2f\n",x,y,res);

    return (0);
}
```

```c
//statement of interest
#include<stdio.h>
int main(void)
{
    int x,y;
    float res;
    scanf("%d %d",&x, &y);

    res=x/y;
    printf("%d/%d =%.2f\n",x,y,res);

    return (0);
}
```

Example of testing-and-debugging a C program (for division): *version **0** with **4** test cases (using gcc 9.3.0)*

```c
//statement of interest
#include<stdio.h>
int main(void)
{
    int x,y;
    float res;
    scanf("%d %d",&x, &y);

    res=x/y;
    printf("%d/%d =%.2f\n",x,y,res);

    return (0);
}
```

```
$ ./a.out < t1
1/1 = 1.00

$ ./a.out < t2
5/4 = 1.00

$ ./a.out < t3
3/4 = 0.00

$ ./a.out < t4
15/5 = 3.00
```

Example of testing-and-debugging a C program (for division): *version **1** with **4** test cases (using gcc 9.3.0)*

```c
//statement of interest
#include<stdio.h>
int main(void)
{
    int x,y;
    float res;
    scanf("%d %d",&x, &y);

    res=(float)(x/y);
    printf("%d/%d =%.2f\n",x,y,res);

    return (0);
}
```

```
$ ./a.out < t1
1/1 = 1.00

$ ./a.out < t2
5/4 = 1.00

$ ./a.out < t3
3/4 = 0.00

$ ./a.out < t4
15/5 = 3.00
```

Example of testing-and-debugging a C program (for division): *version **2** with **4** test cases (using gcc 9.3.0)*

```c
//statement of interest
#include<stdio.h>
int main(void)
{
    int x,y;
    float res;
    scanf("%d %d",&x, &y);

    res=(float)x/y;
    printf("%d/%d =%.2f\n",x,y,res);

    return (0);
}
```

```
$ ./a.out < t1
1/1 = 1.00

$ ./a.out < t2
5/4 = 1.25

$ ./a.out < t3
3/4 = 0.75

$ ./a.out < t4
15/5 = 3.00
```

Example of testing-and-debugging a C program (for division): *version **2** with **7** test cases (using gcc 9.3.0)*

```c
//statement of interest
#include<stdio.h>
int main(void)
{
    int x,y;
    float res;
    scanf("%d %d",&x, &y);

    res=(float)x/y;
    printf("%d/%d =%.2f\n",x,y,res);

    return (0);
}
```

```
$ ./a.out < t1
...

$ ./a.out < t5
1/0 = inf

$ ./a.out < t6
0/0 = -nan

$ ./a.out < t7
0/1 = 0.00
```

Texts, References, and Acknowledgements

Online:

- Continuous Integration and Delivery (**CircleCI**: https://circleci.com)
- http://www.cse.iitm.ac.in/~rupesh/teaching/pa/jan19

Textbook:

- Sharp, J. (2022). *Microsoft Visual C# Step by Step*, 10th edition, Microsoft Press.
- Watson, K., Nagel, C., Pedersen, J. H., Reid, J. D., & Skinner, M. (2008). *Beginning Microsoft Visual C# 2008*. John Wiley & Sons.
- Mark J. Price (2024). *C# 13 and .NET 9 – Modern Cross-Platform Development Fundamentals*, 9th edition, Packt Publishing Ltd.

Reference:

- Soni, M. (2016). *DevOps for Web Development*. Packt Publishing Ltd.
- Yusuf Sulistyo Nugroho, Hideaki Hata, and Kenichi Matsumoto. 2020. *How different are different diff algorithms in Git? Use* `--histogram` *for code changes*. Empirical Softw. Engg. 25, 1 (Jan 2020), 790–823.