IT5440- NGUYÊN LÝ VÀ KỸ THUẬT PHÂN TÍCH CHƯƠNG TRÌNH

(Principle and Technique of Program Analysis)

AY 2025-2026

Giảng viên: PGS. TS.Huỳnh Quyết Thắng
Khoa Khoa học máy tính
Trường Công nghệ thông tin và Truyền thông
www.soict.hust.edu.vn/~thanghq

Chapter II. Static Program Analysis

- *Type systems*
- *Verification*
- *Data-flow Analysis*
- *Control Flow Analysis: Path Sensitivity*
- *CFG (Interprocedural Analysis)*
- *Pointer Analysis*

Chapter II. Static Program Analysis

- **Type systems:**
  - A type system is a foundational concept that plays a crucial role in defining how a programming language classifies and manages the data used within it. This blog post aims to introduce you to the concept of type systems, exploring what they are, why they matter, and how they impact software development.
  - A type system can be seen as a set of rules that assigns a property called “type” to various constructs — such as variables, expressions, functions, or modules — that a computer program is composed of. These types define the kind of data that can be handled by these constructs and the operations that can be performed on them. By enforcing these rules, type systems help in preventing errors, making code more robust and easier to understand.

Chapter II. Static Program Analysis

- **Type systems:**
  - Error detection: Type systems catch a wide range of errors at compile-time or run-time, reducing the likelihood of bugs and improving the reliability of the software.
  - Documentation: The types themselves serve as a form of documentation, providing valuable information about the data being manipulated and the operations being performed.
  - Refactoring: Strong type systems can simplify code refactoring by ensuring that changes in one part of the program do not inadvertently affect other parts in unexpected ways.
  - Optimization: Type information can be used by compilers to optimize code, leading to better performance.

Chapter II. Static Program Analysis

- **Type systems:** A type system can be formally defined as a mechanism in programming languages designed to prevent certain types of errors by classifying values and expressions into types and ensuring that operations are used correctly according to these classifications. It helps in identifying inconsistencies and invalid operations before the program is executed, usually during the compilation phase.
- Type systems also manage data types through static and dynamic type checking:
  - Static type checking: This occurs at compile-time, where the types of all variables and expressions are known and checked before the program runs. Languages like Java, C++, and Haskell use static type checking to catch type errors early in the development process.
  - Dynamic type checking: This occurs at run-time, where the types are checked as the program executes. Languages like Python, JavaScript, and Ruby employ dynamic type checking, allowing for more flexibility but potentially leading to type errors during execution.

Type systems

- Type systems are crucial for ensuring code correctness and reliability for several reasons:
  - **Error prevention**: By catching type errors at compile-time or run-time, type systems help prevent common programming mistakes that could lead to bugs, crashes, or unintended behavior.
  - **Code safety**: Type systems enforce strict rules on how data can be used, ensuring that operations are safe and meaningful. This reduces the likelihood of security vulnerabilities and runtime errors.
  - **Improved documentation**: Types serve as implicit documentation for the code. They provide insight into what kind of data is expected and how it should be manipulated, making the code easier to read and understand.
  - **Enhanced refactoring**: With a robust type system, refactoring code becomes safer and more manageable. Changes in one part of the program are less likely to introduce errors in other parts, as the type system will enforce consistency.
  - **Optimization**: Compilers can use type information to optimize the generated machine code, improving the performance of the program.

Type systems

- Static typing: In statically typed languages, type checking is performed at compile-time. This means that the type of each variable and expression is known and enforced before the program is run. Static typing helps catch type errors early in the development process, leading to more reliable and predictable code.
  - Examples: Java, C, C++, Haskell, Rust.
  - Advantages: Early error detection, performance optimization, and enhanced code completion and refactoring support in development environments.
  - Disadvantages: Reduced flexibility, longer development cycles due to the need for explicit type declarations.

Type systems

- Dynamic typing: In dynamically typed languages, type checking is performed at run-time. This allows for more flexibility, as variables can change types, and operations can be performed on the fly. However, it also means that type errors can occur during execution, potentially leading to run-time crashes.
  - Examples: Python, JavaScript, Ruby, PHP.
  - Advantages: Greater flexibility, faster prototyping, and more concise code due to the absence of explicit type declarations.
  - Disadvantages: Increased risk of run-time errors, potential performance overhead, and challenges in refactoring and code maintenance.

Type systems

- Strong typing: Strongly typed languages enforce strict rules about how types can be interchanged or operated on. Implicit type conversions are minimal or non-existent, ensuring that type errors are less likely to occur.
  - Examples: Haskell, Python, Java, Rust.
  - Advantages: Increased code safety and predictability, as type mismatches are caught early.
  - Disadvantages: Can require more boilerplate code to handle type conversions explicitly.
- Weak typing: Weakly typed languages are more lenient with type conversions, often performing implicit conversions between types. This can lead to unexpected behavior if not carefully managed.
  - Examples: C, C++, JavaScript.
  - Advantages: More flexible and concise code, as implicit conversions reduce the need for explicit type handling.
  - Disadvantages: Increased risk of subtle bugs and unpredictable behavior due to unexpected type coercions.

Type systems

- **Nominal typing:** Nominal typing relies on explicit declarations and names of types. Two types are considered compatible if they are explicitly declared to be the same type or if one is a subtype of the other.
  - **Examples**: Java, C++, Swift.
  - **Advantages**: Clear and explicit type relationships, making code easier to understand and maintain.
  - **Disadvantages**: Less flexibility in reusing types, as type compatibility is based on explicit declarations.
- **Structural typing:** Structural typing determines type compatibility based on the structure or shape of the types, rather than their explicit declarations. If two types have compatible structures (e.g., the same properties or methods), they are considered compatible.
  - **Examples**: TypeScript, Go.
  - **Advantages**: Greater flexibility in type reuse and composition, leading to more versatile and adaptable code.
  - **Disadvantages**: Can be more challenging to understand and maintain the code, as type relationships are less explicit.

Chapter II. Static Program Analysis

- Type inference: Type inference is a powerful feature in programming languages that allows the compiler to automatically deduce the types of expressions without explicit type annotations from the programmer
- Type inference simplifies the coding process by allowing the compiler to infer the types of variables, function parameters, and return types based on the context in which they are used. This reduces the need for verbose type declarations, making the code cleaner and more concise while still maintaining the benefits of a statically typed system.
- How type inference works: Type inference algorithms analyze the code to determine the most specific type that satisfies all constraints placed on an expression. For example, if a variable is initialized with an integer value, the compiler infers its type as an integer. The process involves:
  - Gathering information: Collecting all available information about the types from the code context.
  - Constraint solving: Solving the constraints to find the most specific type that satisfies them.

Type systems

- Benefits of type inference:
  - Reduced boilerplate code: Type inference eliminates the need for repetitive and verbose type annotations, making the code cleaner and more readable.
  - Improved developer productivity: By automating the type assignment process, developers can write code faster and focus on solving problems rather than managing types.
  - Enhanced code maintainability: Cleaner code with fewer type annotations is easier to read and maintain. Changes in the types of variables or functions are automatically managed by the compiler, reducing the risk of errors during refactoring.
  - Early error detection: Despite the lack of explicit type annotations, type inference still provides the benefits of static type checking. This helps in catching type-related errors at compile time rather than at runtime.

Type systems

- Advanced type system concepts: Type systems in programming languages can include advanced features that enhance the expressiveness and safety of code. Here, we explore three significant advanced type system concepts: generics and parametric polymorphism, dependent types, and type checking and type safety.
- Generics and parametric polymorphism: Explanation of generics and parametric polymorphism
- Generics, also known as parametric polymorphism, allow functions, classes, and data structures to operate with any data type without sacrificing type safety. This concept enables the creation of reusable and type-safe code components.

Type systems

- In C++, templates are used to implement **generics**.

```
template <typename T>
class Box {
   T value;
public:
   void setValue(T value) {
      this->value = value;
   }
   T getValue() {
      return value;
   }
};
Box<int> integerBox;
integerBox.setValue(123);
int value = integerBox.getValue();
```

- Here, Box<T> is a template class, allowing T to be any type specified by the user.

Type systems

- Dependent types: Introduction to dependent types and their use cases
  - Dependent types are types that depend on values. This powerful concept allows for more precise type information and can enforce more complex invariants at compile time.
- Type checking and type safety: Explanation of type checking and its importance
  - Type checking is the process of verifying and enforcing the constraints of types to ensure that operations in a program are type-safe. It prevents type errors that could lead to runtime crashes or incorrect behavior.

```
int add (int a, int b) {
  return a + b;
}
// add("hello", "world");
```

Result: ?

Type systems - Conclusion

- Benefits of using type systems:
  - Improved code quality and maintainability: Type systems enforce rules about how values of different types can be used and interacted with in a program. This helps in catching errors early in the development process, leading to higher-quality code.
    - Consistency: By ensuring that variables and functions are used consistently with their declared types, type systems help prevent many common programming errors, such as mismatched data types or incorrect function usage.
    - Refactoring: Strong type systems make it easier to refactor code, as the compiler can help ensure that changes do not introduce type-related errors. This leads to more maintainable and adaptable codebases.
  - Enhanced error detection and debugging: Type systems catch type-related errors at compile time (for statically typed languages) or provide runtime checks (for dynamically typed languages), reducing the likelihood of runtime errors and making debugging easier.
    - Early detection: In statically typed languages, many errors are caught during compilation, before the code is even run. This allows developers to fix issues early, reducing the cost and complexity of debugging.
    - Clear error messages: Type systems often provide clear and specific error messages that indicate exactly where and what the problem is, making it easier to diagnose and resolve issues quickly.

Type systems - Conclusion

Benefits of using type systems:

- Better documentation and code readability: Types serve as a form of documentation, providing valuable information about the expected inputs and outputs of functions, the structure of data, and the intended use of variables.
- Self-documenting code: Well-typed code can often explain itself, as the types provide context and meaning. For example, a function signature can indicate what type of data the function processes and what it returns, making the code easier to understand.
- Improved communication: Types make it easier for developers to communicate their intentions and assumptions, both within a team and with future maintainers of the code. This reduces misunderstandings and helps ensure that the code is used correctly.
- Performance optimizations enabled by type information: Type systems can enable various performance optimizations by providing the compiler with more information about the types of data being used.
- Optimized compilation: Knowing the types of variables and expressions allows the compiler to generate more efficient machine code, as it can make assumptions and optimizations based on type information.
- Runtime efficiency: In languages with type inference and strong typing, the runtime can avoid certain checks and operations that would be necessary in dynamically typed languages, leading to faster execution and reduced overhead.

Verification

Verification is a process of determining if the software is designed and developed as per the specified requirements.
Verification is done to check if the software being developed has adhered to these specifications at every stage of the development life cycle. The verification ensures that the code logic is in line with specifications.

Data-flow analysis

Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program. A program's control-flow graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate. The information gathered is often used by compilers when optimizing a program. A canonical example of a data-flow analysis is reaching definitions.
The most commonly used program representation.

Program representation: Basic blocks

A basic block in program P is a sequence of consecutive statements with a single entry and a single exit point. Thus a block has unique entry and exit points.

Control always enters a basic block at its entry point and exits from its exit point. There is no possibility of exit or a halt at any point inside the basic block except at its exit point. The entry and exit points of a basic block coincide when the block contains only one statement.

Control Flow Graph (CFG)

A control flow graph (or flow graph) G is defined as a finite set N of nodes and a finite set E of edges. An edge (i, j) in E connects two nodes n_i and n_j in N. We often write G= (N, E) to denote a flow graph G with nodes given by N and edges by E.

Control Flow Graph (CFG)

In a flow graph of a program, each basic block becomes a node and edges are used to indicate the flow of control between blocks.

An edge (i, j) connecting basic blocks b_i and b_j implies that control can go from block b_i to block b_j.

We also assume that there is a node labeled Start in N that has no incoming edge, and another node labeled End, also in N, that has no outgoing edge.

Paths

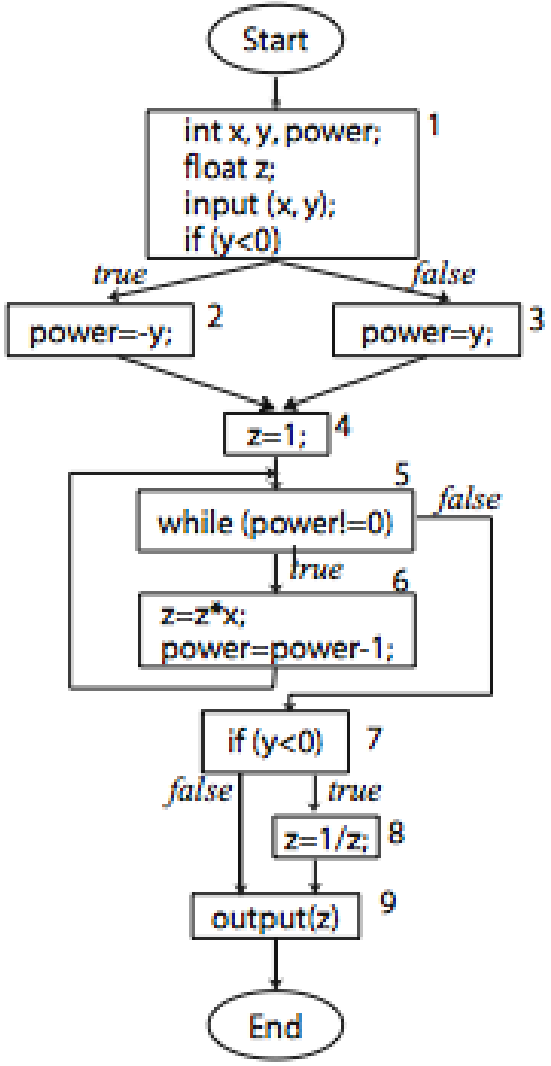
Consider a flow graph G= (N, E). A sequence of k edges, k>0, (e_1, e_2, … e_k) , denotes a path of length k through the flow graph if the following sequence condition holds.

Given that n_p, n_q, n_r, and n_s are nodes belonging to N, and 0< i<k, if $e_i = (n_p, n_q)$ and $e_{i+1} = (n_r, n_s)$ then $n_q = n_r$. }

Complete path: a path from start to exit
Subpath: a subsequence of a complete path

Paths: infeasible paths

A path p through a flow graph for program P is considered feasible if there exists at least one test case which when input to P causes p to be traversed.



p_1= (Start, 1, 3, 4, 5, 6, 5, 7, 8, 9, End)
p_2= (Start, 1, 2, 4, 5, 7, 9, End)

Number of paths

There can be many distinct paths through a program. A program with no condition contains exactly one path that begins at node Start and terminates at node End.

Each additional condition in the program can increases the number of distinct paths by at least one.

Depending on their location, conditions can have a multiplicative effect on the number of paths.

A Simplified Version of CFG

- Each statement is represented by a node
  - For readibility.
  - Not for efficient implementation.

Dominator

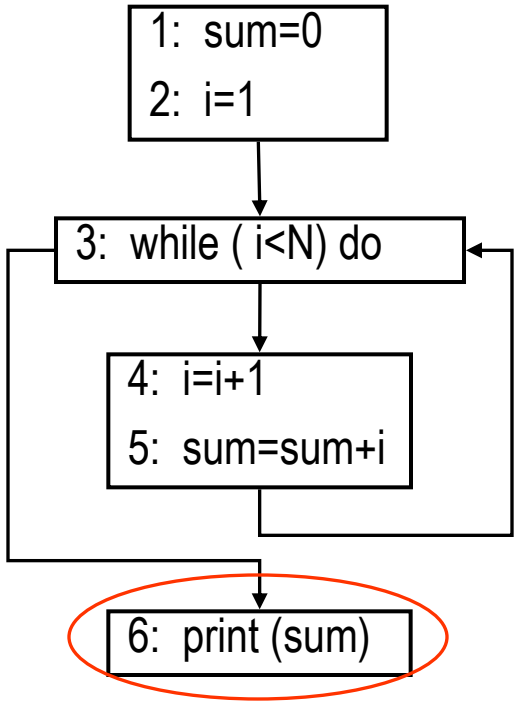
- X dominates Y if all possible program paths from START to Y have to pass X.

Dominator

- X strictly dominates Y if X dominates Y and X!=Y

```
1:    sum=0
2:    i=1
3:    while ( i<N) do
4:          i=i+1
5:          sum=sum+i
      endwhile
6:  print(sum)
```

SDOM(6)={1,3}

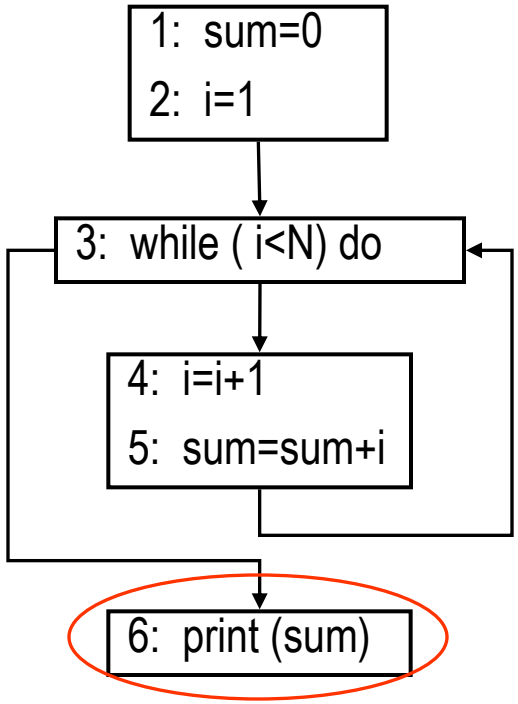


Dominator

- X is the immediate dominator of Y if X is the last dominator of Y along a path from Start to Y.

```
1:    sum=0
2:    i=1
3:    while ( i<N) do
4:          i=i+1
5:          sum=sum+i
      endwhile
6:  print(sum)
```

IDOM(6)={3}

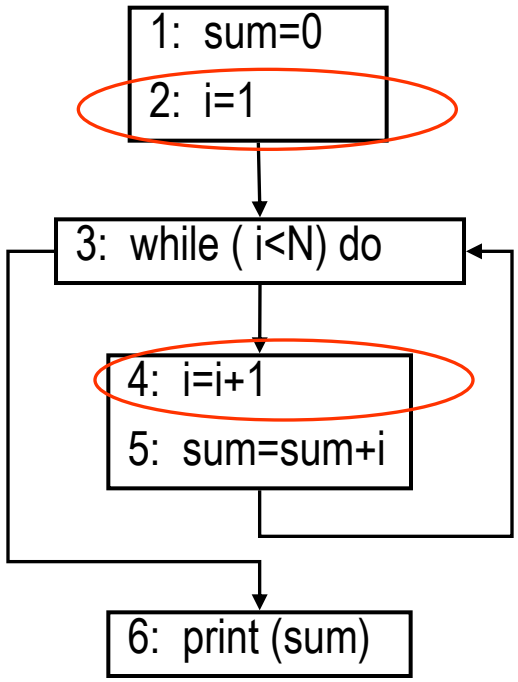


Postdominator

- X post-dominates Y if every possible program path from Y to End has to pass X.
  - Strict post-dominator, immediate post-dominance.

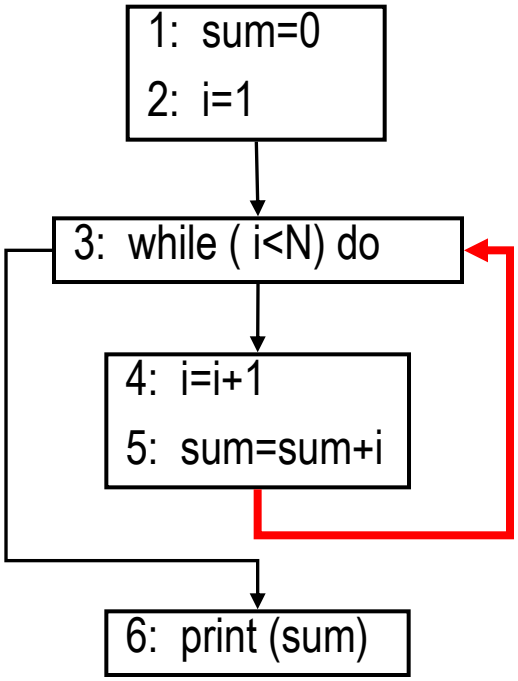
```
1:    sum=0
2:    i=1
3:    while ( i<N) do
4:          i=i+1
5:          sum=sum+i
      endwhile
6:  print(sum)
```

SPDOM(4)={3,6} IPDOM(4)=3



Back Edges

- A back edge is an edge whose head dominates its tail
  - Back edges often identify loops

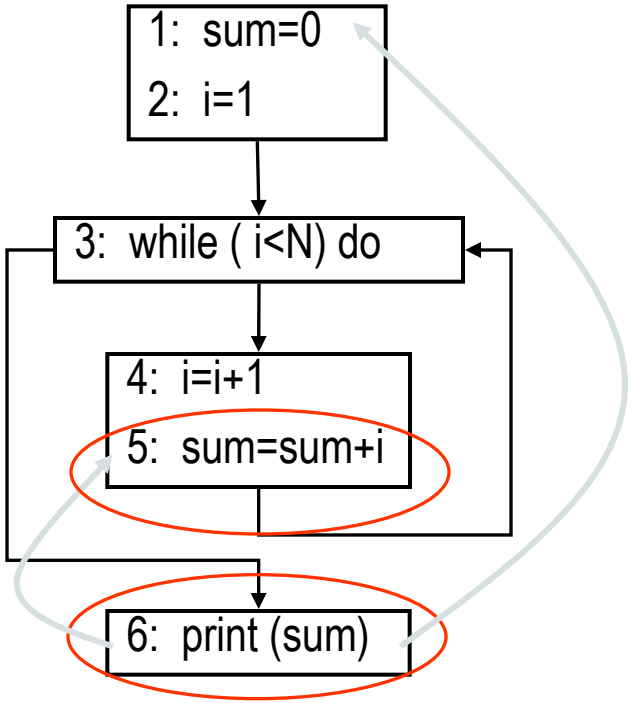


Program Dependence Graph

- The second widely used program representation.
- Nodes are constituted by statements instead of basic blocks.
- Two types of dependences between statements
  - Data dependence
  - Control dependence

Data Dependence

- X is data dependent on Y if (1) there is a variable v that is defined at Y and used at X and (2) there exists a path of nonzero length from Y to X along which v is not re-defined.



Computing Data Dependence is Hard in General

- Aliasing
  - A variable can refer to multiple memory locations/objects.

```
1:   int x, y, z …;
2:   int * p;
3:   x=…;
4:   y=…;
5:   p = & x;
6:   p=p +z;
7:   … = *p;
```

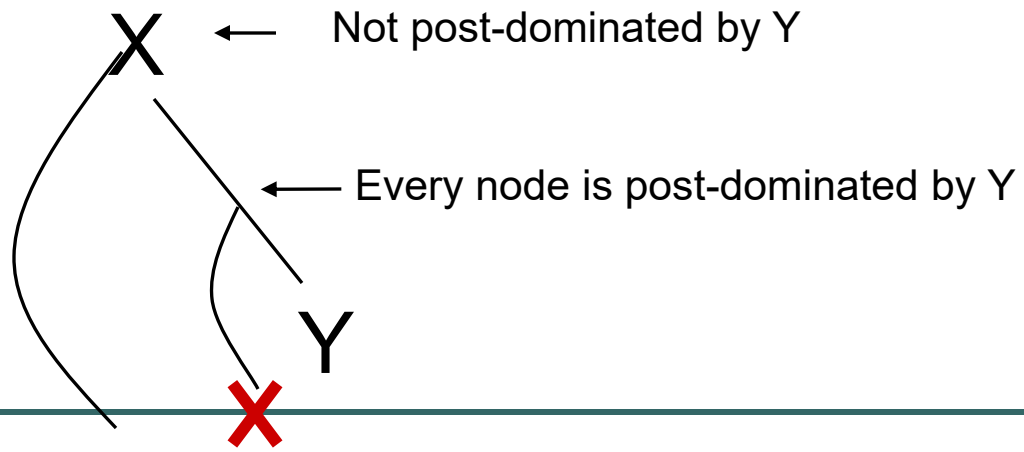
```
1:   foo (ClassX  x,    ClassY  y) {
2:      x.field= …;
3:      …=y.field;
4:   }
```

```
foo ( o, o);

o1=new ClassX( );
o2= new ClassY( );
foo ( o1, o2);
```

Control Dependence

- Intuitively, Y is control-dependent on X iff X directly determines whether Y executes (statements inside one branch of a predicate are usually control dependent on the predicate)
  - X is not strictly post-dominated by Y ⇒ There is a path from X to End that does not pass Y or X==Y
  - there exists a path from X to Y s.t. every node in the path other than X and Y is post-dominated by Y ⇒ No such paths for nodes in a path between X and Y.



Control Dependence - Example

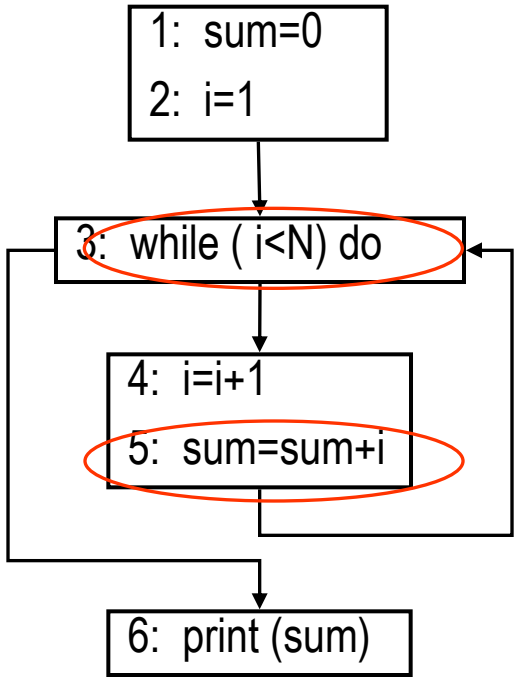
Y is control-dependent on X iff X directly determines whether Y executes

- X is not strictly post-dominated by Y
- there exists a path from X to Y s.t. every node in the path other than X and Y is post-dominated by Y

```
1:    sum=0
2:    i=1
3:    while ( i<N) do
4:          i=i+1
5:          sum=sum+i
      endwhile
6:  print(sum)
```

CD(5)=3

CD(3)=3, tricky!

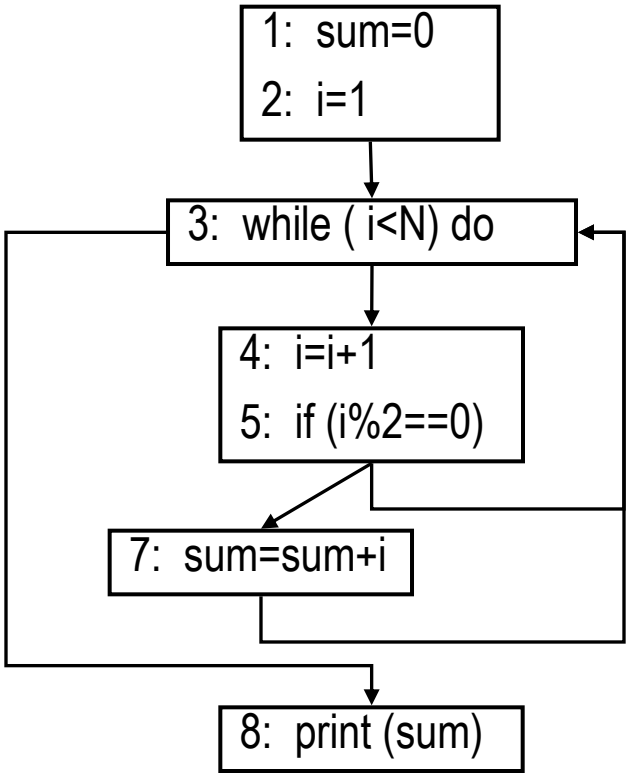


Note: Control Dependence is not Syntactically Explicit

Y is control-dependent on X iff X directly determines whether Y executes

- X is not strictly post-dominated by Y
- there exists a path from X to Y s.t. every node in the path other than X and Y is post-dominated by Y

```
1:    sum=0
2:    i=1
3:    while ( i<N) do
4:          i=i+1
5:          if (i%2==0)
6:               continue;
7:          sum=sum+i
      endwhile
8:    print(sum)
```



Control Dependence is Tricky!

Y is control-dependent on X iff X directly determines whether Y executes

- X is not strictly post-dominated by Y
- there exists a path from X to Y s.t. every node in the path other than X and Y is post-dominated by Y

- Can a statement control depends on two predicates?

Control Dependence is Tricky!

Y is control-dependent on X iff X directly determines whether Y executes

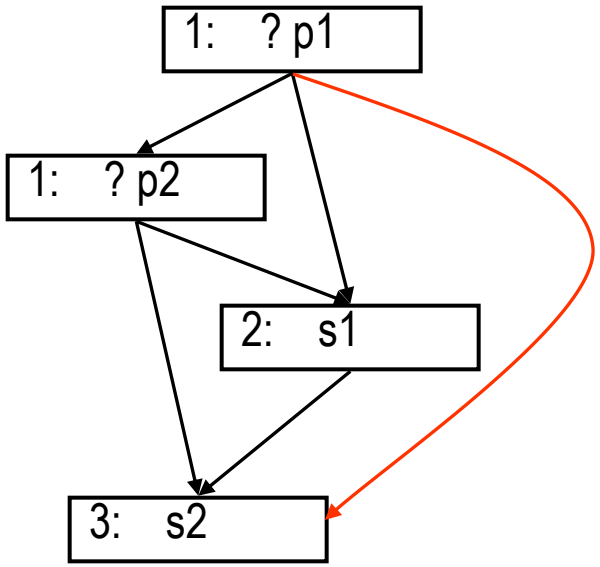
- X is not strictly post-dominated by Y
- there exists a path from X to Y s.t. every node in the path other than X and Y is post-dominated by Y

- Can one statement control depends on two predicates?

```
1:    if ( p1 || p2 )
2:            s1;
3:    s2;
```

What if ?

```
1:    if ( p1 && p2 )
2:            s1;
3:    s2;
```



The Use of PDG

- A program dependence graph consists of control dependence graph and data dependence graph
- Why it is so important to software reliability?
  - In debugging, what could possibly induce the failure?
  - In security

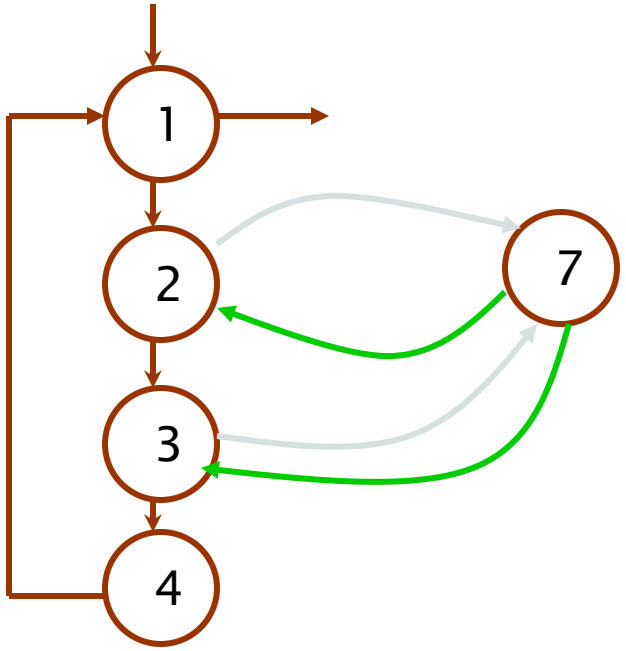
```
p=getpassword( );
…
send (p);
```

```
p=getpassword( );
…
if (p==“zhang”) {
    send (m);
}
```

Super Control Flow Graph (SCFG)

- Besides the normal intraprocedural control flow graph, additional edges are added connecting
  - Each call site to the beginning of the procedure it calls.
  - The return statement back to the call site.

```
1:    for (i=0; i<n; i++) {
2:       t1= f(0);
3:       t2 = f(243);
4:       x[i] = t1 + t2 + t3;
5:   }
6:  int  f (int v) {
7:    return (v+1);
8:  }
```



Call Graph (CG)

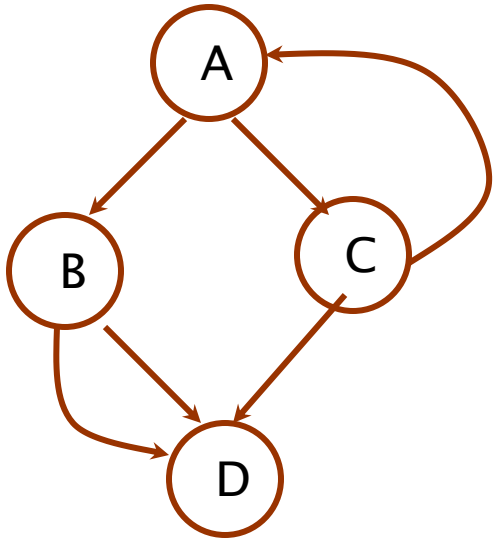
- Each node represents a function; each edge represents a function invocation

```
void A( ) {
   B( );
   C( );
}

void B( ) {
L1:   D( );
L2:   D( );
}

void C ( ) {
   D( );
   A( );
}

void D ( ) {
}
```

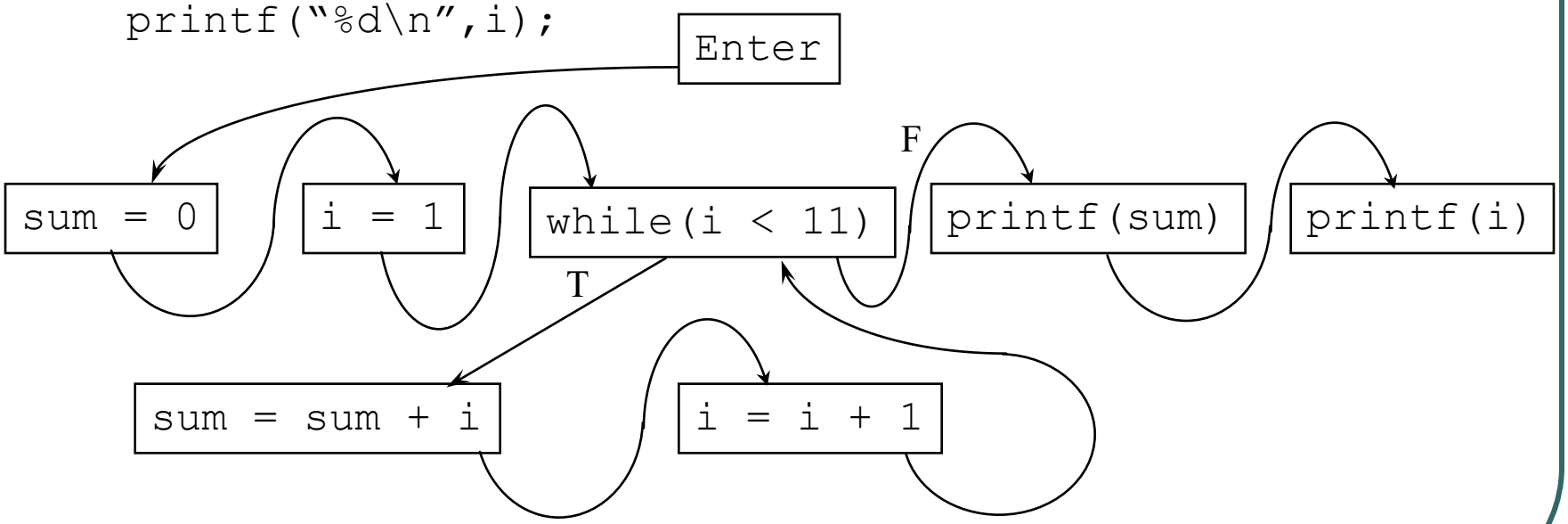


Use of CG

- CFI
- Android framework access control inconsistencies
- Hidden behavior detection in Android apps

Control Flow Graph

```
int main() {
        int sum = 0;
        int i = 1;
        while (i < 11) {
                sum = sum + i;
                i = i + 1;
        }
        printf("%d\n",sum);
        printf("%d\n",i);
}
```



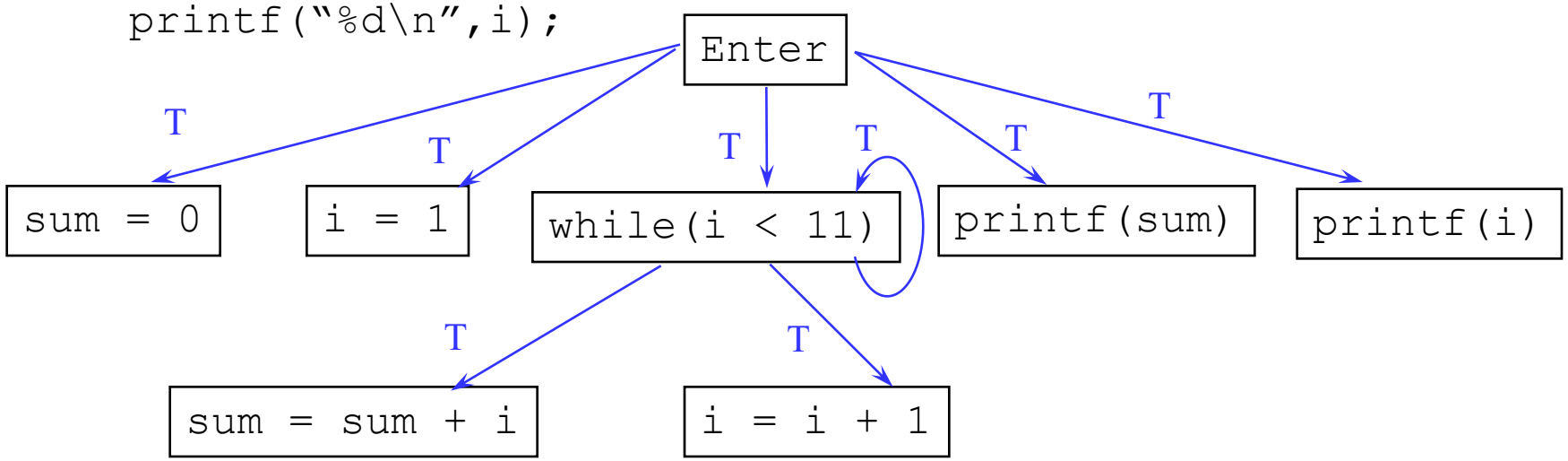
Control Dependence Graph

```
int main() {
        int sum = 0;
        int i = 1;
        while (i < 11) {
                sum = sum + i;
                i = i + 1;
        }
        printf("%d\n",sum);
        printf("%d\n",i);
}
```

Control dependence

$p \xrightarrow{T} q$ q is reached from p if condition p is true (T), not otherwise.

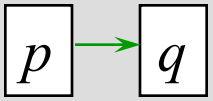
$p \xrightarrow{F} q$ Similar for false (F).



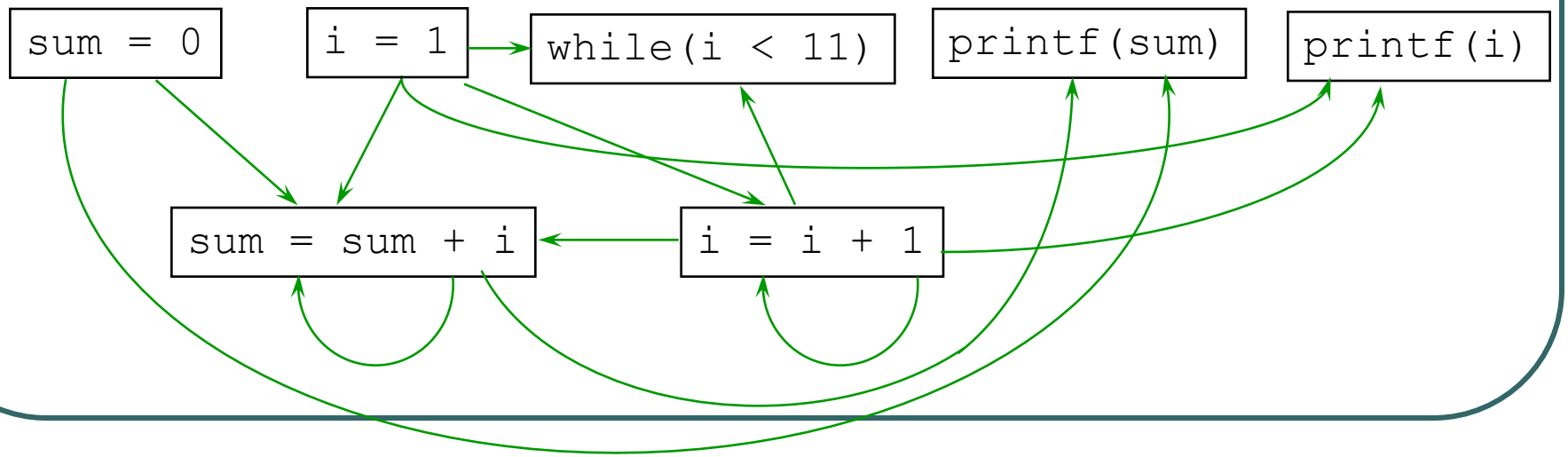
Flow Dependence Graph

```
int main() {
        int sum = 0;
        int i = 1;
        while (i < 11) {
                sum = sum + i;
                i = i + 1;
        }
        printf("%d\n",sum);
        printf("%d\n",i);
}
```

Flow dependence



Value of variable assigned at *p* may be used at q.

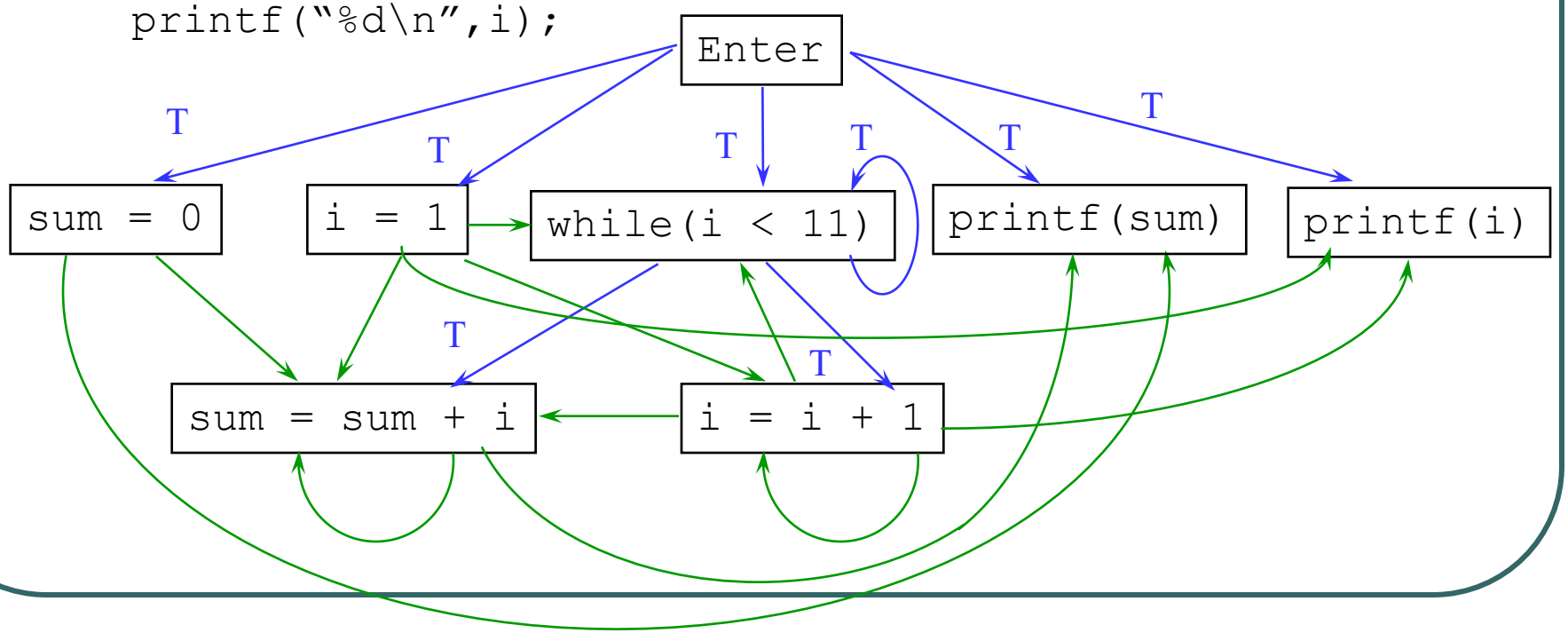


Program Dependence Graph (PDG)

```
int main() {
        int sum = 0;
        int i = 1;
        while (i < 11) {
                sum = sum + i;
                i = i + 1;
        }
        printf("%d\n",sum);
        printf("%d\n",i);
}
```

Control dependence

Flow dependence

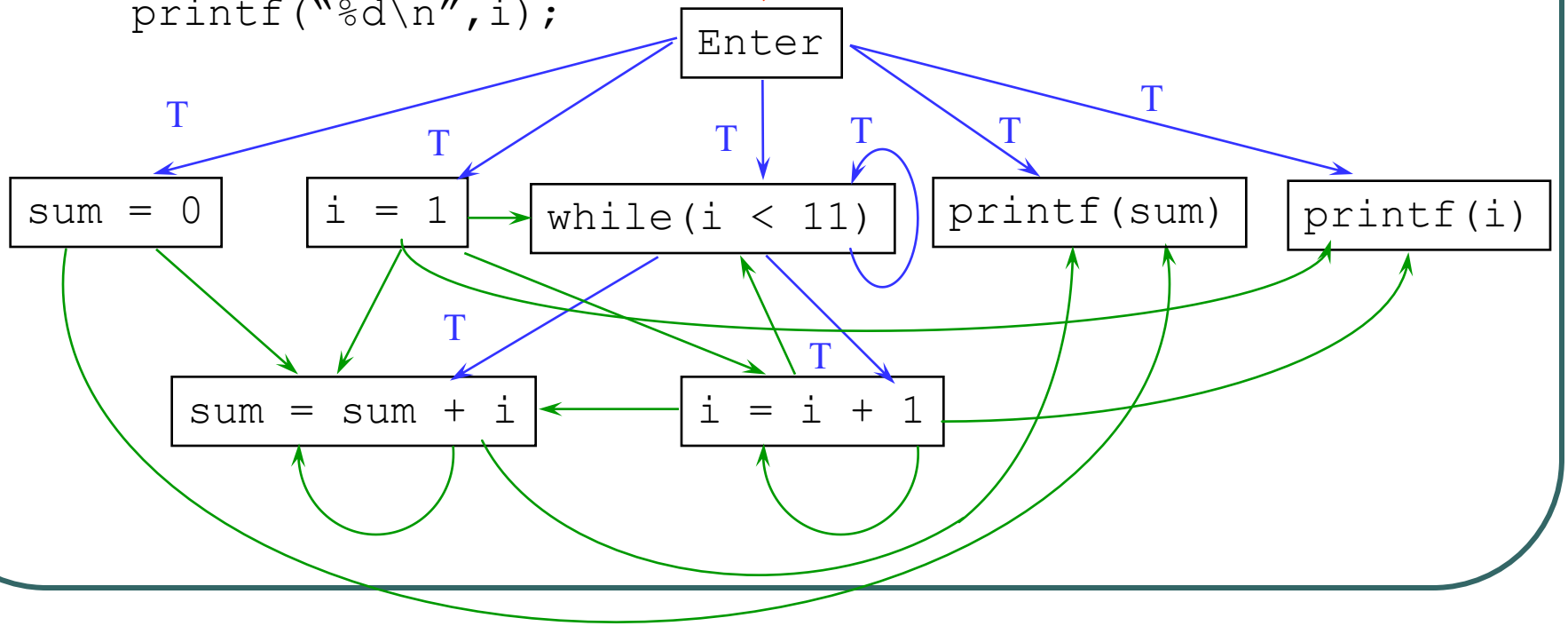


Program Dependence Graph (PDG)

```
int main() {
        int i = 1;
        int sum = 0;
        while (i < 11) {
                sum = sum + i;
                i = i + 1;
        }
        printf("%d\n",sum);
        printf("%d\n",i);
}
```

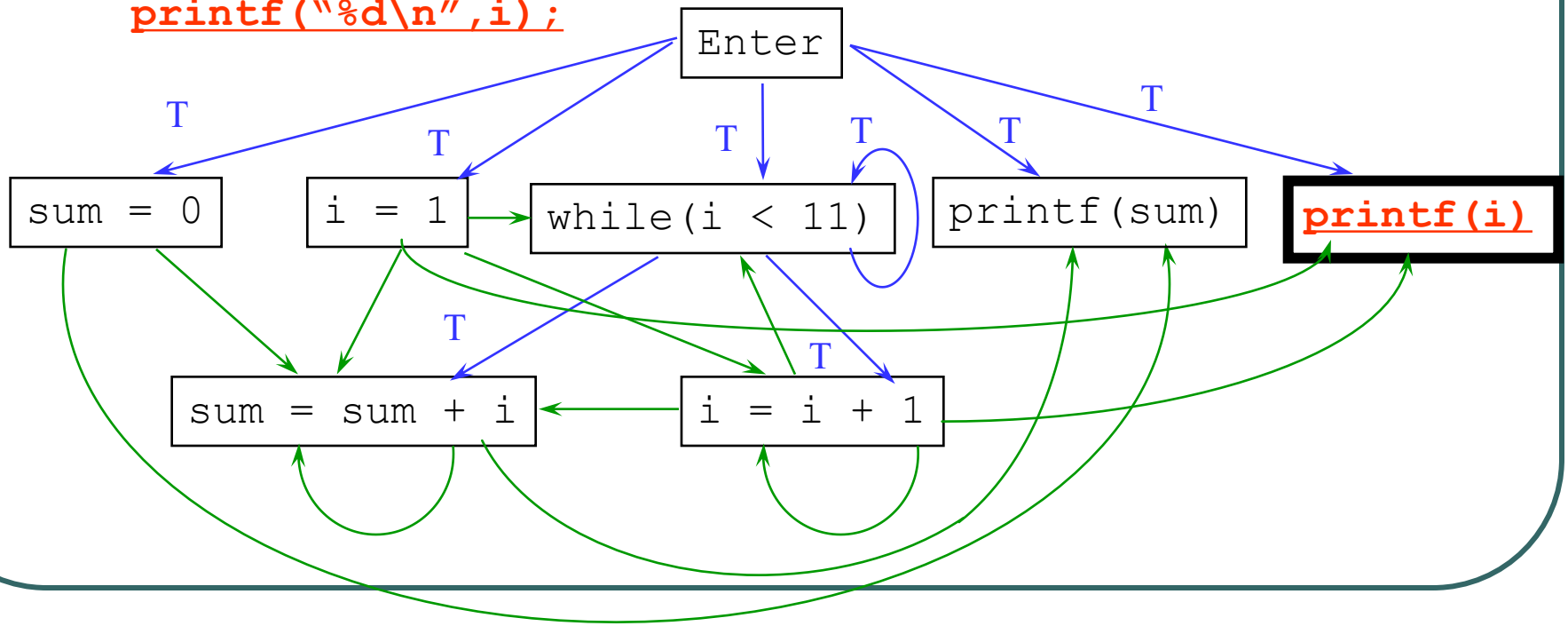
Opposite Order

Same PDG



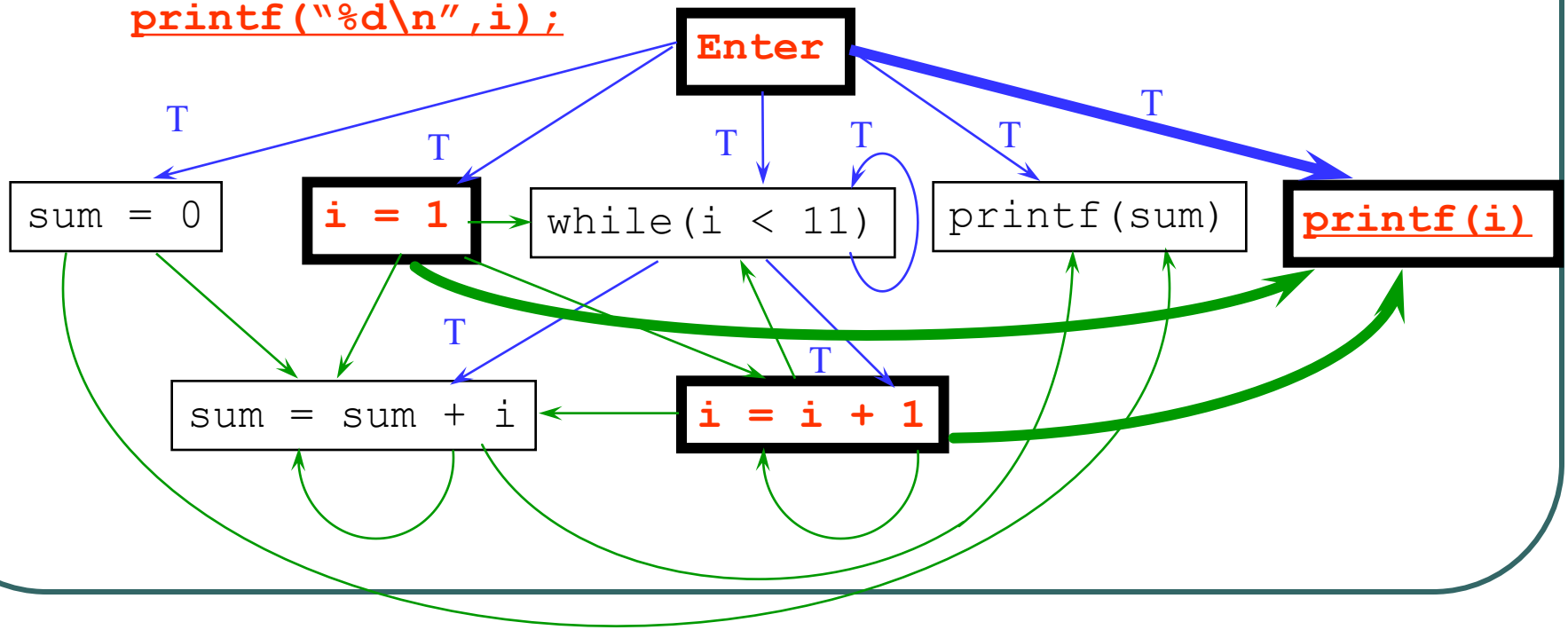
Backward Slice

```
int main() {
        int sum = 0;
        int i = 1;
        while (i < 11) {
                sum = sum + i;
                i = i + 1;
        }
        printf("%d\n",sum);
        printf("%d\n",i);
}
```



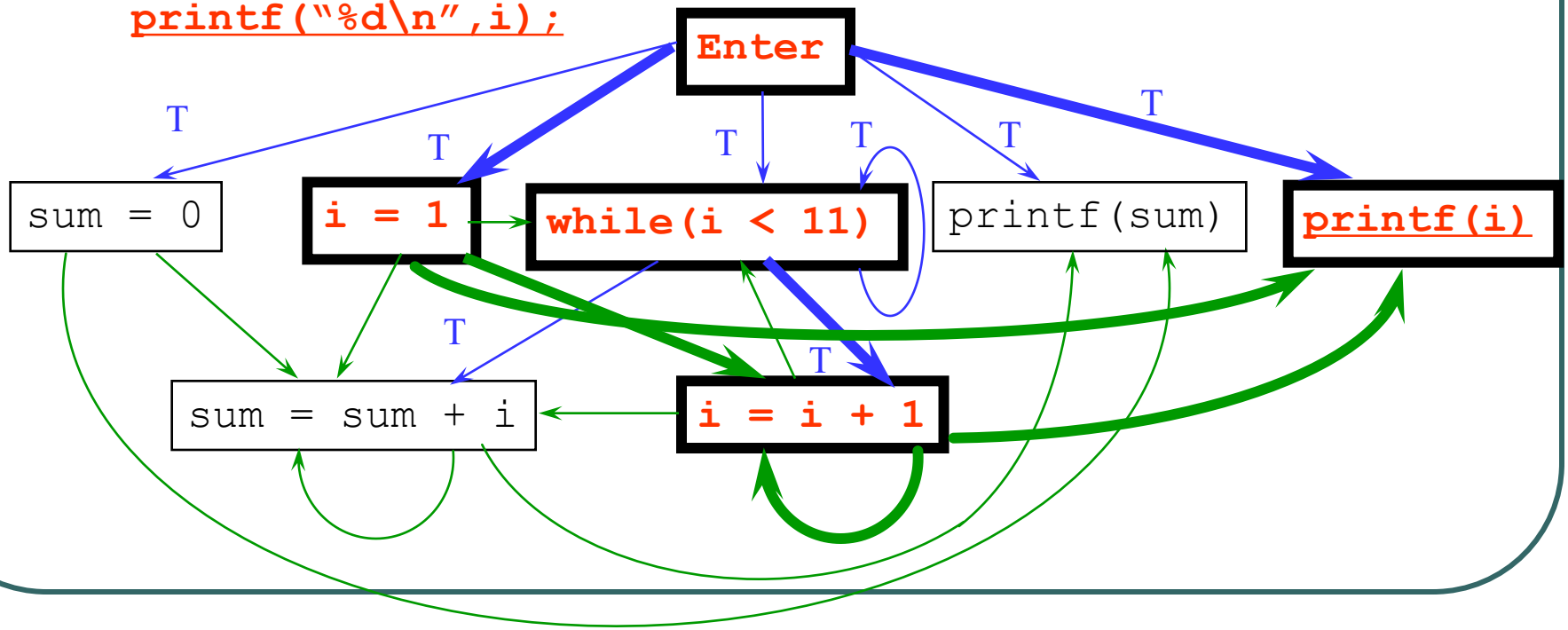
Backward Slice (2)

```
int main() {
        int sum = 0;
        int i = 1;
        while (i < 11) {
                sum = sum + i;
                i = i + 1;
        }
        printf("%d\n",sum);
        printf("%d\n",i);
}
```



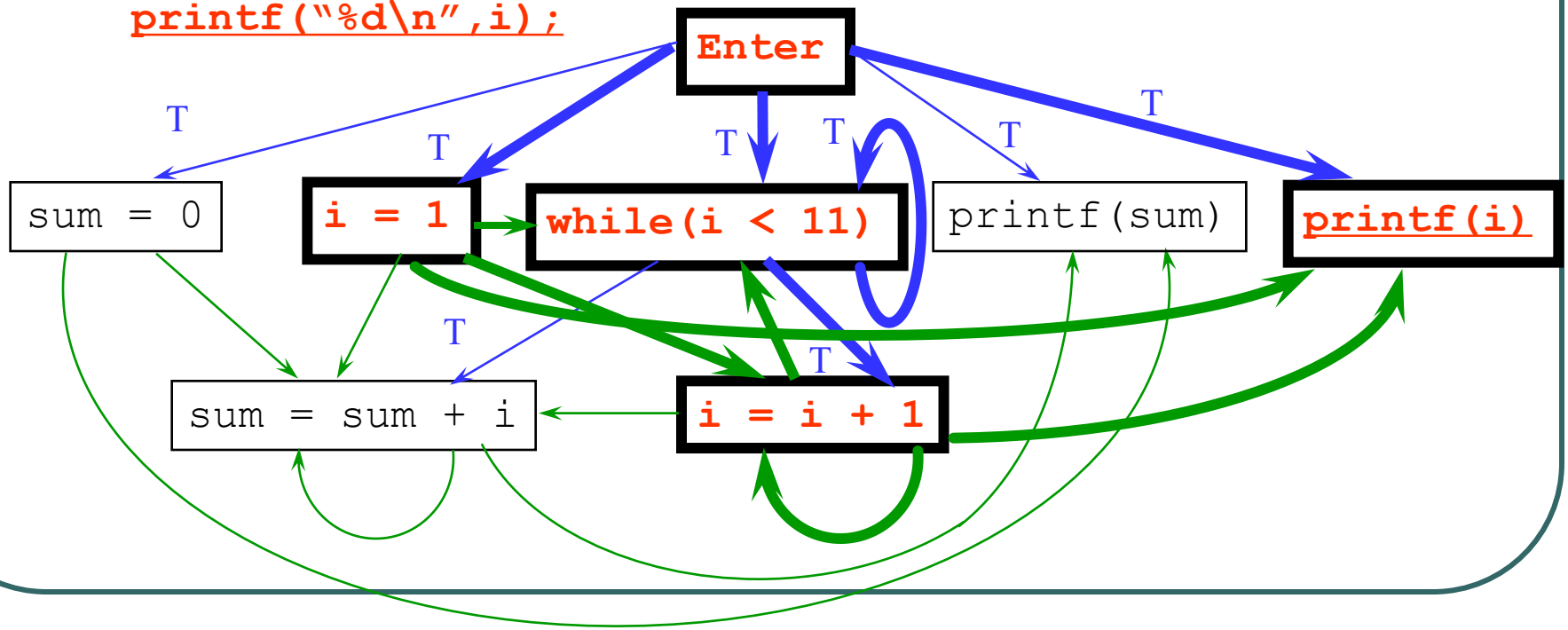
Backward Slice (3)

```
int main() {
        int sum = 0;
        int i = 1;
        while (i < 11) {
                sum = sum + i;
                i = i + 1;
        }
        printf("%d\n",sum);
        printf("%d\n",i);
}
```



Backward Slice (4)

```
int main() {
        int sum = 0;
        int i = 1;
        while (i < 11) {
                sum = sum + i;
                i = i + 1;
        }
        printf("%d\n",sum);
        printf("%d\n",i);
}
```



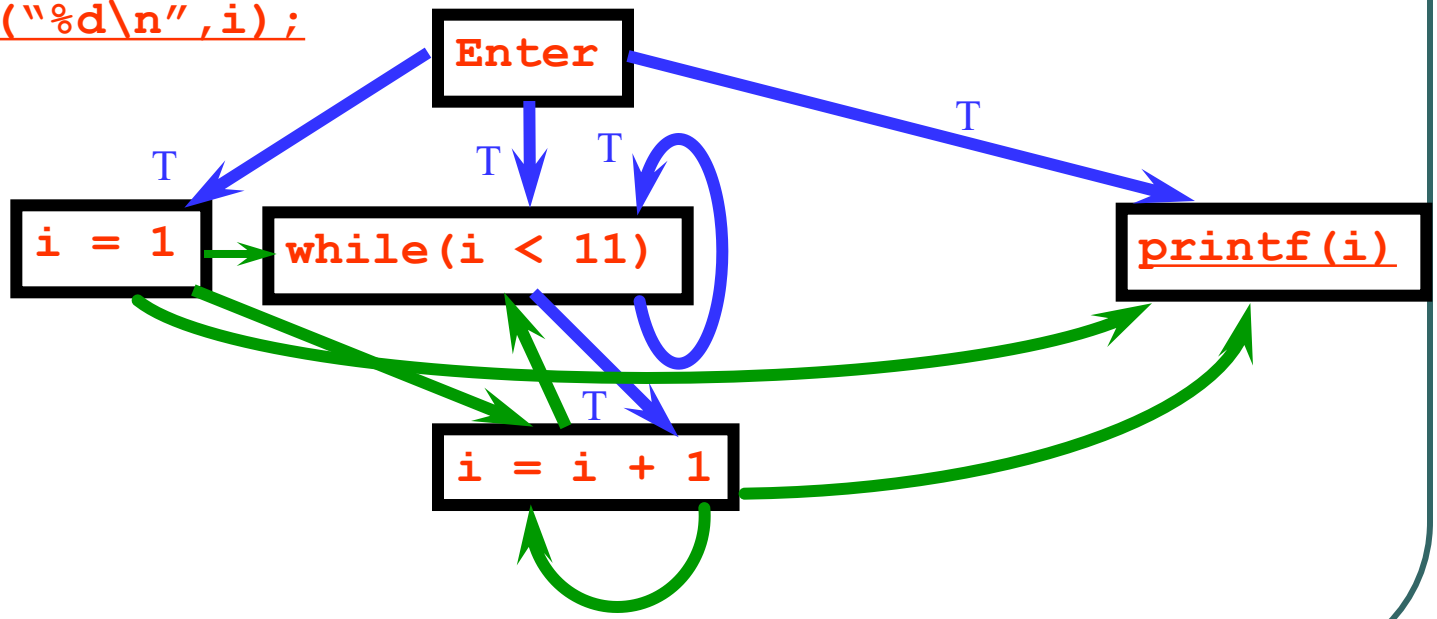
Slice Extraction

```
int main() {

        int i = 1;
        while (i < 11) {

                i = i + 1;
        }

        printf("%d\n",i);
}
```



Memory Allocation

- 3 important task:
  - Determine memory **requirement**
    To represent value of data items.
  - Determine memory **allocation**
    To implement lifetime & scope of data item.
  - Determine memory **mapping**
    To access value in non-scalar data items.
- Binding: A memory binding is an association b/w memory address attributes of the data item & address of memory area.
- Topic list:
  A. Static and dynamic memory allocation
  B. Memory allocation in block structured language
  C. Array allocation & access

Static & Dynamic Allocation

Static Memory Allocation

1. Memory is allocated to variable before execution of program begins.
2. Performed during compilation.
3. At execution no allocation & de-allocation is performed.
4. Allocation to variable exist even if program unit is not active.

Dynamic Memory Allocation

1. Memory allocation of variable is done all the time of execution of program.
2. Performed during execution.
3. Allocation & de-allocation actions occurs only at execution.
4. Allocation to variable exist only if program unit is active.

Static Memory Allocation

5. Variable remains allocated permanently till execution does not end.
6. Eg: Fortran
7. No. of flavors or types
8. Adv: Simplicity and faster access.
9. Memory wastage compared to dynamic.

Memory
Code(A)
Data (A)
Code (B)
Data (B)
Code (C)
Data (C)

Dynamic Memory Allocation

5. Variables swap from and to allocation state & free state.
6. Eg: PL/I, ADA, Pascal.
7. Two types/ flavors
   - 1. automatic allocation
   - 2. Program controlled allo
8. Adv: Recursion support DS size dynamically.
9. Less memory wastage compared to formal.

Memory: Only A is active
Code(A)
Code (B)
Code (C)
Data (A)

Memory: A calls B. A & B is active
Code(A)
Code (B)
Code (C)
Data (A)
Data (B)

Automatic v/s Program Controlled Dynamic Allocation

Automatic Dynamic Allocation

1. Implies memory binding performed at execution initiation time of program unit.
2. Memory is allocated to declared variables when execution starts.
3. De-allocated when program unit is exited.
4. Different memory areas may be allocated to same variable in different activation of program unit.
5. Implemented using stack since entry, exit is LIFO by nature.
6. Implemented variables of program are accessed using displacement from this pointer.

Program Controlled Dynamic Allocation

1. Implies memory binding performed during the execution of program unit.
2. Memory is allocated not at execution but when used for very first time during execution.
3. De-allocated when arbitrary points are left while execution.
4. Here allocation is done when program modules start purely based on scope of variables.
5. Implemented using heap DS, as not always LIFO by nature.
6. Pointer is needed now to point to each memory area allocated.

Memory allocation in Block Structured Language

- Block contain data declaration
- There may be nested structure of block
- Block structure uses dynamic memory allocation
- Eg: PL/I, Pascal, ADA
- Sub Topic List
  1. Scope Rules
  2. Memory Allocation & Access
  3. Accessing Non-local variables
  4. Symbol table requirement
  5. Recursion
  6. Limitation of stack based memory allocation

1. Scope Rules:

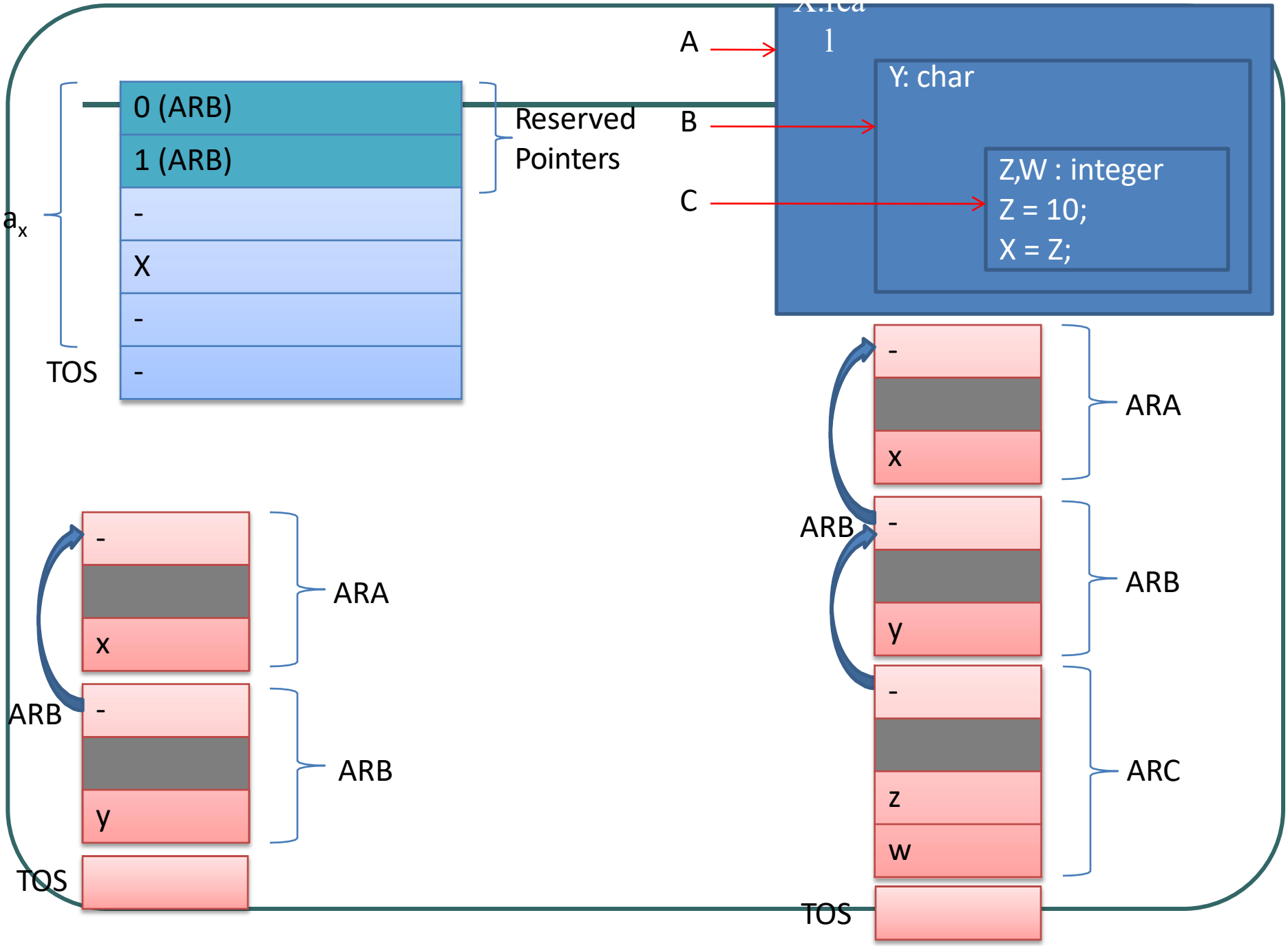
- If variables var_i is created with name $name_i$ in block B.
- Rule 1: var_i can be accessed in any statement situated in block B.
- Rule 2: Rule 1 + 'B' is enclosed in 'B' unless 'B' contains declaration using same name $name_i$.
- Eg: variables B = local
  B' = non-local

```
A{
        x,y,z : integer;
    B{  g : real;
        C{      h,z : real;
        }C
    }B
    D{  i,j : integer;
    }D
}A
```

Block	Accessibility of Variables	
	Local	Non-Local
A	xA,yA,zA	
B	gB	xA,yA,zA
C	hC, zC	xA,yA,gB
D	iD, jD	xA, yA, zA

2. Memory allocation & Access

- Implemented using extended stack model.
- Automatic dynamic allocation is implemented using extended stack model.
- Minor variation: each record has two reserved pointers, determining its scope.
- AR: Activation Record
- ARB: Activation Record Block
- See the following figure:
  - When execution starts, state changes from (a) to state (b).
  - When execution exit block C, state change (a) to (b).


A
B
C
Y: char
Z,W : integer
Z = 10;
X = Z;
0 (ARB)
1 (ARB)
-
X
-
-
Reserved
Pointers
a_x
TOS
-
x
ARA
ARB
-
y
ARB
TOS
-
x
ARA
ARB
-
y
ARB
-
z
w
ARC
TOS

- **Allocation:**
  1. TOS := TOS + 1;
  2. TOS* := ARB;
  3. ARB := TOS;
  4. TOS := TOS + 1;
  5. TOS* := ……(secreserved pointer 2)
  6. TOS := TOS + n;
- So address of ‘z’ is <ARB> + z;
- **De-allocation:**
  1. TOS := ARB – 1;
  2. ARB := ARB* ;

3. Accessing non-local variables

- n1_var : is a non local variable
- b_def : block defined into
- b_use : block in use

- Then textual ancestor of block b_use is a block which encloses block b_use that is b_def.
- Level m ancestor is a block which immediately encloses level m-1 ancestor.
- S_nest_b_use: static nesting level of block b_use.
- Rule: when b_use is in execution, b_def must be active.
- i.e ARBb_def exist in stack while execution ARb_use.
- n1_var are accessed by start address of

  ARb_def + dn1_var

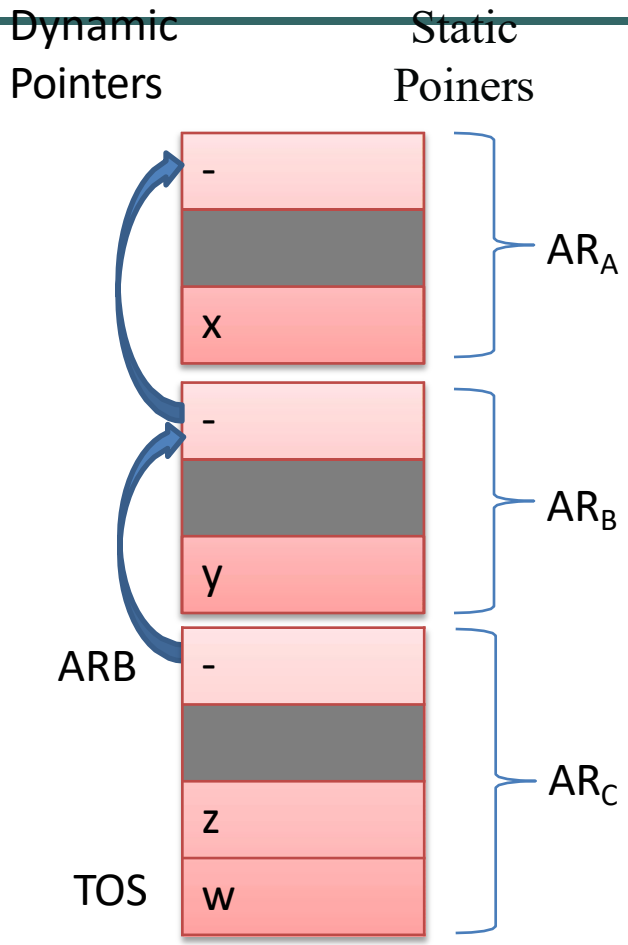
  where, dn1_var is displacement of n1_var in ARb_def.
- Lets learn few more terms
  - Static Pointers
  - Display

(i) Static Pointer:

- Access of non-local variables is implemented using second reserved pointer in AR.
- It has 1 (ARB)
  0 (ARB)
- 1 (ARB) is static pointer.
- At the time of creation of AR for Block B its static pointer is set to point AR of static ancestor of b.
- Access non-local variables:

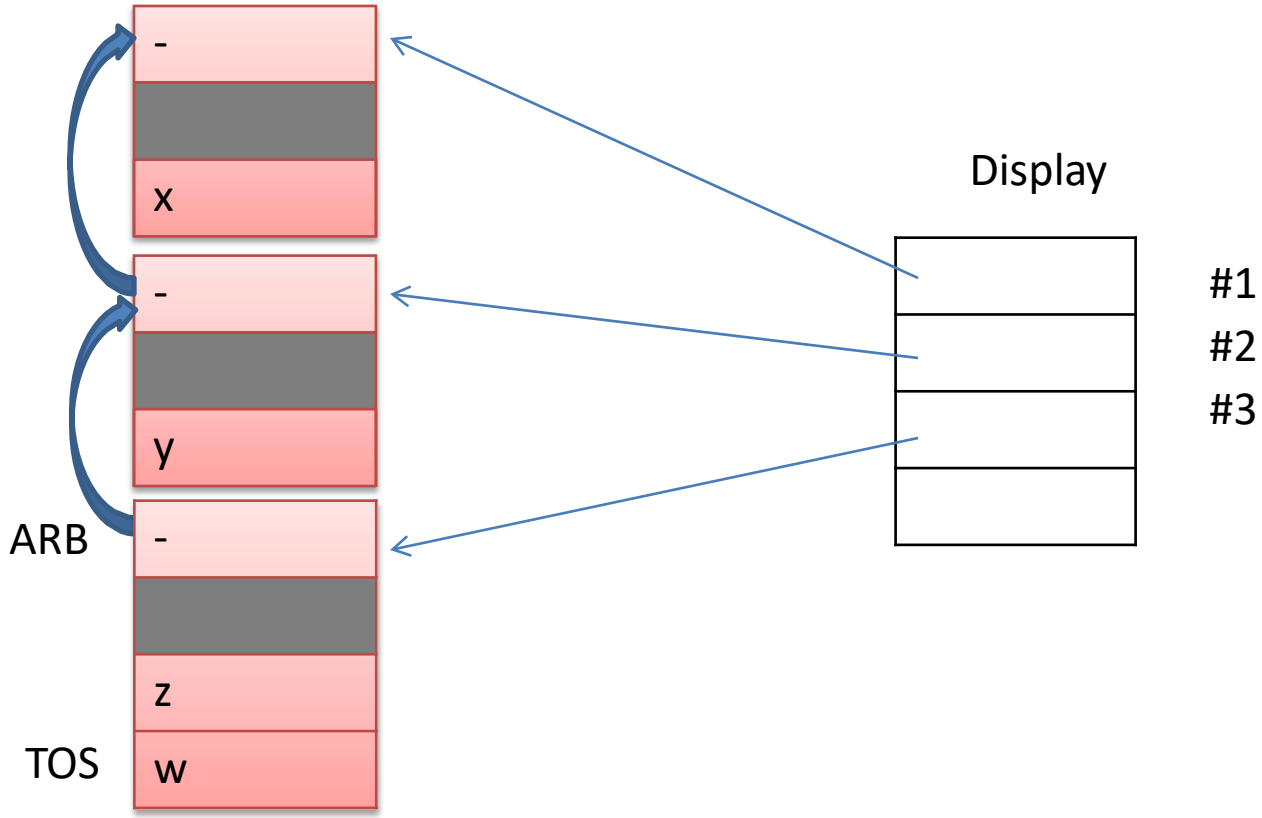
  1. r := ARB;
  2. Repeat step-3 m times
  3. r := 1(r)
  4. Access n1_var using address <r> + dn1_var.

- Example: Status after execution:
  - TOS := TOS + 1
  - TOS* := address of AR at level 1 ancestor.
  - <r> to access x at statement x:= z
    - r:= ARB;
    - r:= 1(r);
    - r:= 1(r);
    - Access x using address <r> + dx.


Dynamic
Pointers
Static
Poiners
-
x
AR_A
-
y
AR_B
ARB
-
z
AR_C
TOS
w

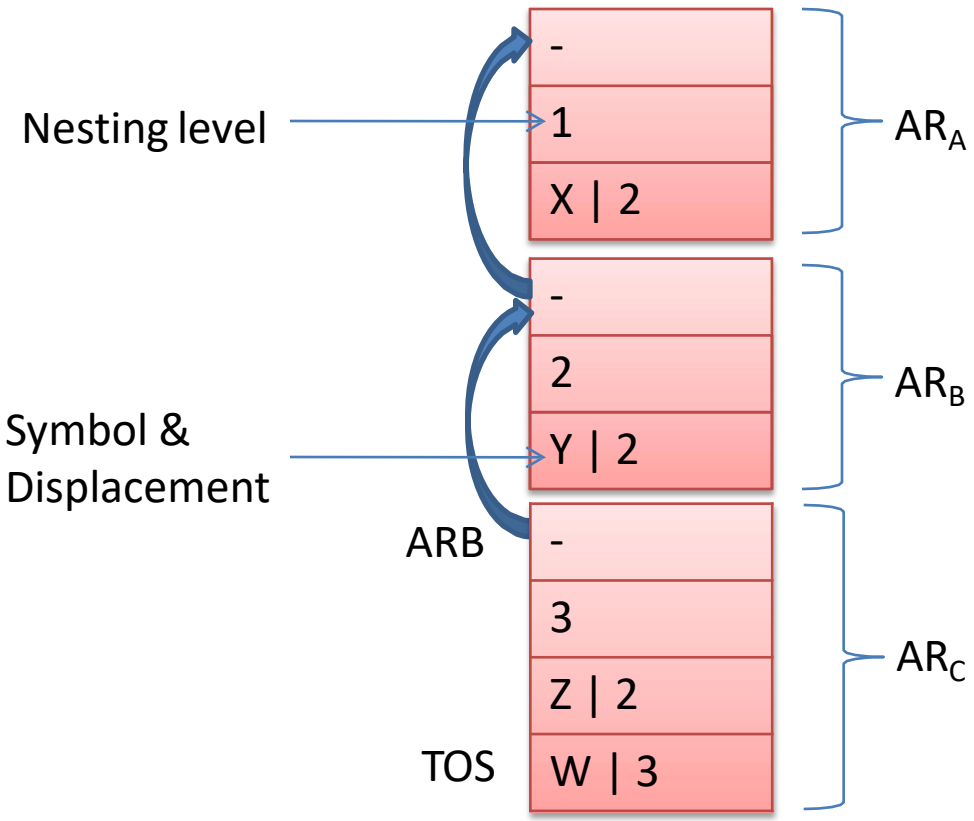
(ii) Displays:

- Display (1) = address of level (S_nest_b-1) ancestor of B.
- Display (2) = address of level (S_nest_b-2) ancestor of B.
- Display [S_nestb-1] = address of level 1 ancestor of B.
- Display [S_nestb-1] = address of AR_B.
- For large value of level difference, it is expensive to access non-local variables using static pointers.
- Display is an array used to improve the efficiency of non-local variables accessibility.
- Eg: r:= display[1]
  access x using address <r> + dx.


-
x
-
y
ARB
-
z
TOS
w
Display
#1
#2
#3

4. Symbol Table Requirement:

- To improve dynamic allocation & access, compiler should perform following task.
  - Determine static nesting level of b_current
  - Determine variable designated with scope rules.
  - Determine static nesting level of block & dv.
  - Generate code.
- Extended stack model is used bcz it has
  - Nesting level of b_current
  - Symbol table for b_current.
- Symbol table has
  - Symbol
  - Displacement.


Nesting level
Symbol &
Displacement
ARB
TOS
-
1
X | 2
-
2
Y | 2
-
3
Z | 2
W | 3
AR_A
AR_B
AR_C

Array allocation & Access

- A[5,10]. 2D array arranged column wise.
- Lower bound is 1.
- Address of element A[s1,s2] is determined by formula:
- Ad.A[s1,s2] = Ad.A[1,1] + { (s2-1)xn + (s1-1)} x k.
- Where,
  - n is number of rows.
  - k is size, number of words required by each element.
- General 2D array can be represented by a[l1:u1,l2:u2]
- The formula becomes
  Ad.A[S1,S2] = Ad.A[l1,l2] + {(s2-l2) x (u1-l1+1) + (s1-l1)} x k.
- Defining the range:
  - Range1: u1-l1+1
  - Range2: u2-l2+1
- Ad.A[s1,s2]
  = Ad.A[l1,l2] + {(s2-l2) x range1 + (s1-l1)} x k.
  = Ad.A[l1,l2] – (l2 x range1 + l1) x k + (s2 x range1 +s1) x k.
  = Ad.A[0,0] + (s1 x range1 + s1) x k.
- We can compute following values at compilation time or allocation time (given that subscripts are constants).

A[1,1]
-
-
A[5,1]
A[1,2]
-
A[5,2]
-
-
A[1,10]
-
A[5,10]

Homework 2

Lựa chọn 1 công cụ/ngôn ngữ lập trình mà nhóm thống nhất là nắm vững nhất.

1) Minh họa ý tưởng phân tích chương trình sử dụng Dataflow Analysis

2) Minh họa ý tương phân tích mã nguồn chương trình sử dụng kỹ thuật CFG

3) Minh họa các ý thuật cấp phát bộ nhớ và giải phóng bộ nhớ

4) Minh họa ý tưởng sử dụng và giải phóng pointer