What is Scope rule? Explain with example

- A scope in any programming is a region of the program where a defined variable can have its exixtence and beyond that a variable it cannot be accessed.
- These rules determine where an identifier can be used and accessed, ensuring proper resolution of references and avoiding conflicts.
- Following are the basic four scope rules used in the system programming :

File scope :

- The Variables declared outside of any function or block have file scope, often referred to as File scope or global scope.
- They are typically declared at the top of a source file. The file-scoped variables are visible to all functions within the same file and are accessible from the point of declaration to the end of the file.
- They are accessible from any function within the file and can be accessed globally, making them available to all functions.
- Example :

```
// File scope variable
int globalVar = 10;
void function1() {
   // Accessing globalVar within the function
   int x = globalVar;
}
```

Block scope :

- Variables declared within a pair of curly braces {} have block scope. This includes variables declared in loops, if statements, or any other compound statement.
- Their visibility is limited to the block in which they are declared, and outside the block, these variables are not visible.
- They are accessible only within the block where they are defined and persist only within the block. Memory is allocated when the block is entered and deallocated when the block is exited.
- Example :

```
void exampleFunction() {
   // Block scope variable
   int localVar = 20;
```

```
    // localVar is only accessible within this block
    if (condition) {
      int anotherVar = 30;
      // localVar is accessible here
      // anotherVar is only accessible within this if block
    }
    // anotherVar is not accessible here
  }
```

Function Prototype Scope:

- In these the scope of variables begin right after the declaration in function parameter list.
- In C, function prototypes can declare variables, and these have function prototype scope.
- Their visibility is limited to the function prototype scope and they are accessible only within the function prototype
- Example :

```
// Function prototype with file scope
void myFunctionPrototype(int param1, int param2);
int main() {
  myFunctionPrototype(1, 2);
    // myFunctionPrototype is visible and can be called here
}
void someOtherFunction() {
    myFunctionPrototype(3, 4);
    // myFunctionPrototype is visible and can be called here
}
```

Function Scope :

- Variables declared within a function have function scope. Their visibility is limited to the function in which they are declared, and they are accessible only within the function.
- The lifetime of function-scoped variables extends from the point of declaration until the end of the function.
- They are created when the function is called and cease to exist when the function execution completes.

- Example :

```
void myFunction() {
   // Function scope variable
   int localVar = 42;

   // localVar is only accessible within this function
}
```

What is Memory Allocation? Differentiate Static and Dynamic memory allocation

- The placement of blocks of information in a memory system is called memory allocation
- To allocate memory it is necessary to keep in information of available memory in the system.if sufficient memory is not available, swapping of blocks is done.
- There are typically two types of memory allocation - Static Memory Allocation and Dynamic Memory Allocation. Each type has distinct characteristics and use cases, offering different advantages and limitations.

S.No	Static Memory Allocation	Dynamic Memory Allocation
1	In the static memory allocation, variables get allocated permanently, till the program executes or function call finishes.	In the Dynamic memory allocation, variables get allocated only if your program unit gets active.
2	Static Memory Allocation is done before program execution.	Dynamic Memory Allocation is done during program execution.
3	It uses stack for managing the static allocation of memory	It uses heap for managing the dynamic allocation of memory
4	It is less efficient	It is more efficient
5	In Static Memory Allocation, there is no memory re-usability	In Dynamic Memory Allocation, there is memory re-usability and memory can be freed when not required

6	In static memory allocation, once the memory is allocated, the memory size can not change.	In dynamic memory allocation, when memory is allocated the memory size can be changed.
7	In this memory allocation scheme, we cannot reuse the unused memory.	This allows reusing the memory. The user can allocate more memory when required. Also, the user can release the memory when the user needs it.
8	In this memory allocation scheme, execution is faster than dynamic memory allocation.	In this memory allocation scheme, execution is slower than static memory allocation.
9	In this memory is allocated at compile time.	In this memory is allocated at run time.
10	In this allocated memory remains from start to end of the program.	In this allocated memory can be released at any time during the program.
11	**Example:** This static memory allocation is generally used for array.	**Example:** This dynamic memory allocation is generally used for linked list.

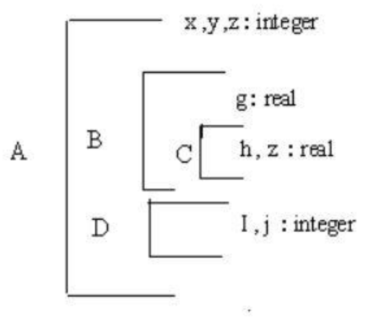
Discuss in brief Memory allocation in Block structured languages

- The placement of blocks of information in a memory system is called memory allocation
- To allocate memory it is necessary to keep in information of available memory in the system.if sufficient memory is not available, swapping of blocks is done**.**
- In block-structured languages, memory allocation is closely tied to the concept of blocks, which are program segments containing data declarations.
- These languages often employ dynamic memory allocation to handle the creation and destruction of blocks during program execution.

Scope Rules

- If a variable 'var' is created with the name 'n' in a block b.
- 'var' can be accessed in any statement situated in block b.
- 'var' can be accessed in any statement situated in a block b. which is enclosed in b, unless b| contains a declaration using the same name.

Consider the sample program given below.



Block	Accessible variable Local	Non-local
A	x_A, y_A, z_A	----------
B	g_B	x_A , y_A , z_A
C	h_C, z_C	x_A , y_A , g_B
D	i_D , j_D	x_A , y_A , z_A

Variable z_A is not accessible inside block C since C contains a declaration using the same name. Thus z_A and z_C are two distinct variables.

Memory Allocation and Access in Block-Structured Languages:

1. **Automatic Memory Allocation:**
    - Block-structured languages commonly implement automatic memory allocation.
    - This means that memory is allocated and deallocated dynamically during program execution based on the creation and destruction of blocks.
2. **Activation Record (AR):**
    - In the context of memory allocation, each block activation is associated with a record known as an Activation Record (AR).
    - An Activation Record contains variables specific to one activation of a block.
    - These records help manage the state of each block during its execution.
3. **Extended Stack Model:**
    - The extended stack model is used to organize memory during program execution.
    - The stack grows and shrinks as blocks are entered and exited.
    - Each block activation has its corresponding Activation Record.
4. **Activation Record Base (ARB):**
    - During execution, the Activation Record Base (ARB) is a pointer that indicates the beginning of the Top of Stack (TOS) record.
    - It serves as a reference point for accessing variables within the current block activation.
5. **Accessing Local Variables:**
    - Local variables within a block can be accessed using the formula: **<ARB> + dx**.
    - Here, **<ARB>** is the Activation Record Base, and **dx** represents the displacement from the starting point to the position where the variable is stored.

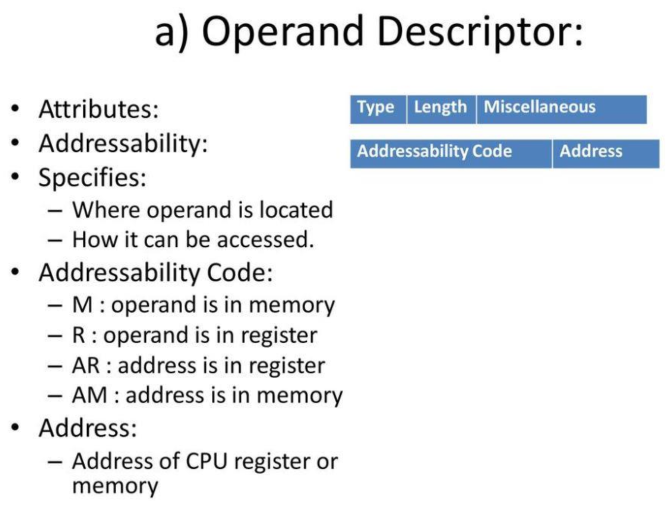
What are Operand descriptors and Register descriptors explain with an example

- The operand descriptor type is an essential component of the interface between the high and low level code-generation modules.
- Operand descriptors in macroinstructions play a crucial role in specifying the behavior and characteristics of operands
- They provide a mechanism for defining the nature of operands and how they should be processed during macroinstruction expansion**.**

Purpose of Operand Descriptors:

1. Define how operands in a macroinstruction should be interpreted and processed.
2. Specify the type, format, and possible values associated with each operand.

- Each operand description begins with an explanation of the operand functionOperand descriptors adhere to general rules for coding operands, including specifying quantities (e.g., decimal integers, expressions, or registers)



a) Operand Descriptor:

- Attributes:
- Addressability:
- Specifies:
  - Where operand is located
  - How it can be accessed.
- Addressability Code:
  - M : operand is in memory
  - R : operand is in register
  - AR : address is in register
  - AM : address is in memory
- Address:
  - Address of CPU register or memory

Type	Length	Miscellaneous

Addressability Code	Address

Register descriptors –

- A register descriptor is a data structure that describes the characteristics of a register in a computer's processor.
- The register descriptor is used by the compiler and runtime system to manage the use of registers in the processor. It allows the compiler to select the appropriate registers for different operations and helps the runtime.
- In some systems, the register descriptor may also include additional information, such as the register's location in the processor, the register's current state (e.g., whether it is in use or available), or other implementation-specific details.

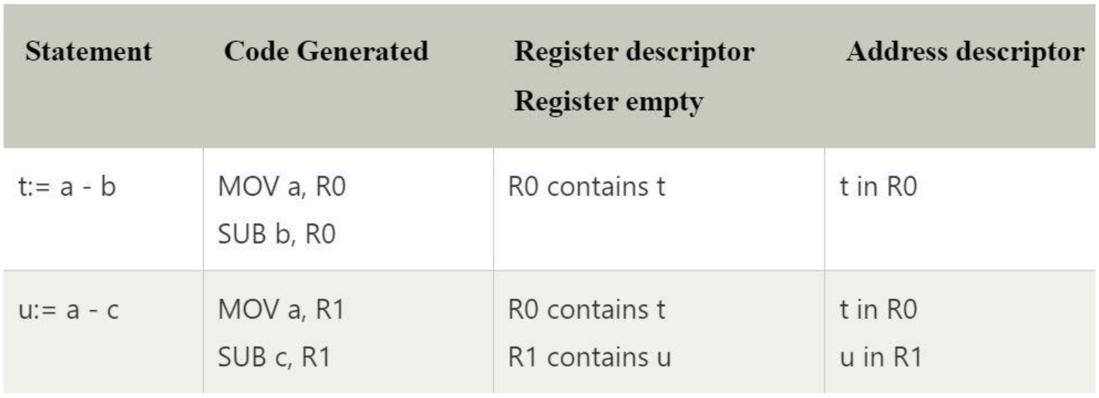
It typically includes the following information:

- The name of the register: This is a human-readable identifier for the register, such as “eax” or “r3”.
- The size of the register: This is the number of bits in the register, which determines the range of values that it can hold. For example, a 32-bit register can hold values from 0 to 2^32 – 1.
- The purpose of the register: This is the intended use of the register, such as storing data values, holding addresses, or storing status information.
- The register’s role in the instruction set: This is the specific role that the register plays in the instruction set of the processor. For example, some registers may be used as source operands, while others may be used as destination operands.

Applications of Register Descriptors:

- **Compilers:** Used to enhance code performance by guiding the compiler in utilizing registers efficiently for storing variables.
- **Virtual Machines:** Assist in executing code more efficiently by describing available registers and permissible value types.
- **Debuggers**: Provide insights into program state by displaying current register values during debugging.
- **Operating Systems:** Help describe available registers for user programs, ensuring proper usage and preventing unauthorized access.

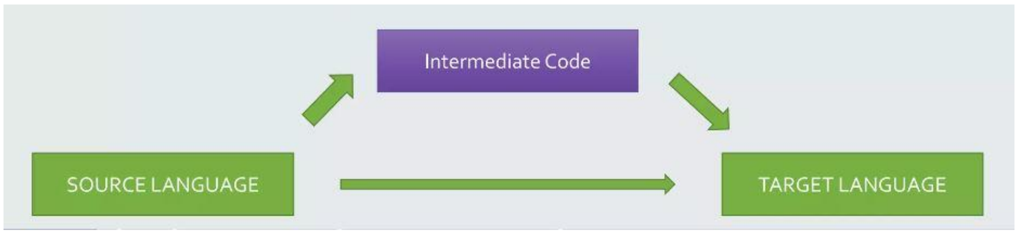
Example :



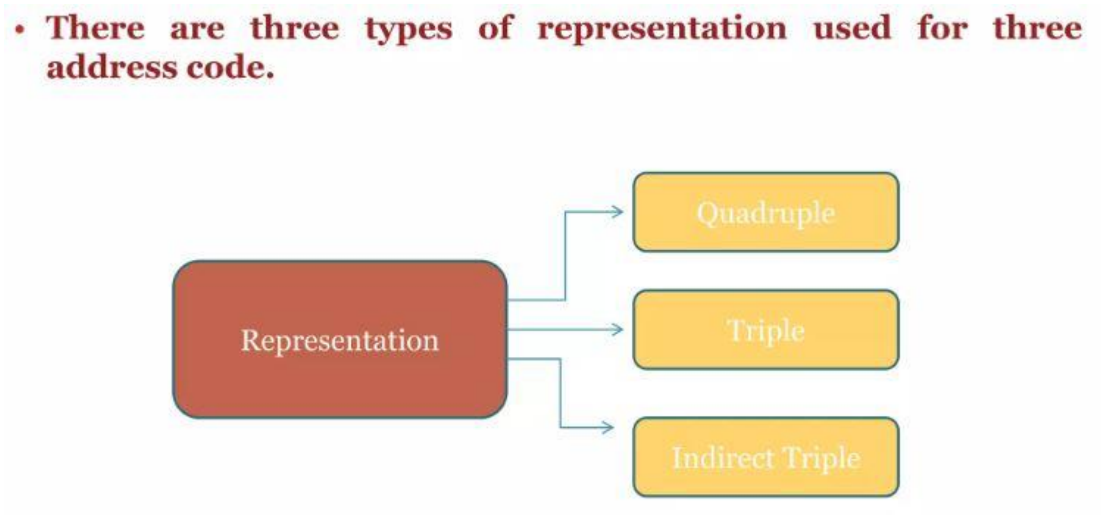
Statement	Code Generated	Register descriptor Register empty	Address descriptor
t:= a - b	MOV a, R0 SUB b, R0	R0 contains t	t in R0
u:= a - c	MOV a, R1 SUB c, R1	R0 contains t R1 contains u	t in R0 u in R1

Explain Triples with an example

- During the translation of a source program into the object code for a target machine, a compiler may generate a middle-level language code, which is known as intermediate code.



- **Three address code** is a type of intermediate code which is easy to generate and can be easily converted to machine code.



- There are three types of representation used for three address code.

Optimization:

- Three address code is often used as an intermediate representation of code during optimization phases of the compilation process.
- The three address code allows the compiler to analyze the code and perform optimizations that can improve the performance of the generated code.

Code generation:

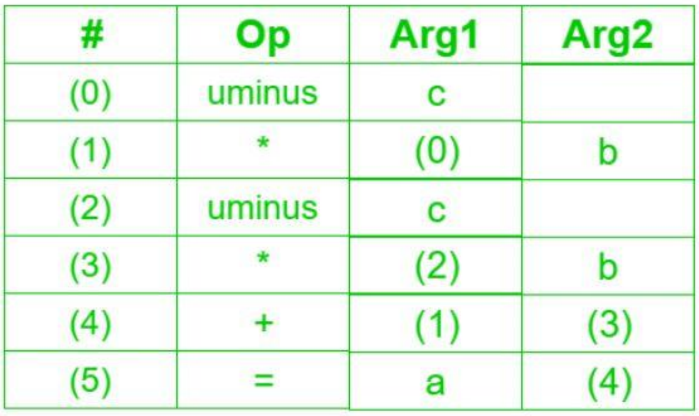
- Three address code can also be used as an intermediate representation of code during the code generation phase of the compilation process.
- The three address code allows the compiler to generate code that is specific to the target platform, while also ensuring that the generated code is correct and efficient.

Triples :

- The triples have three fields to implement the three address code. The field of triples contains the name of the operator, the first source operand and the second source operand.

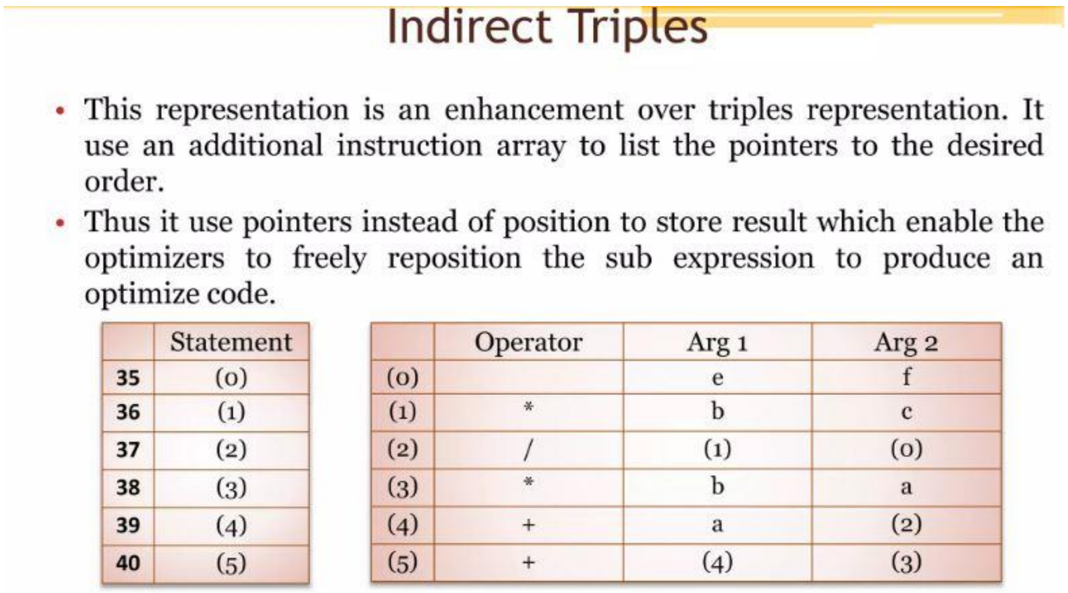
Operator
Source 1
Source 2

Example – Consider expression a = b * – c + b * – c



#	Op	Arg1	Arg2
(0)	uminus	c	
(1)	*	(0)	b
(2)	uminus	c	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	a	(4)

Triples representation



Indirect Triples

- This representation is an enhancement over triples representation. It use an additional instruction array to list the pointers to the desired order.
- Thus it use pointers instead of position to store result which enable the optimizers to freely reposition the sub expression to produce an optimize code.

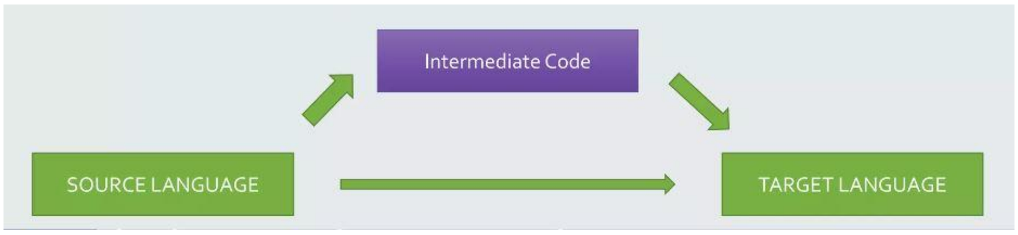
	Statement
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
40	(5)

	Operator	Arg 1	Arg 2
(0)		e	f
(1)	*	b	c
(2)	/	(1)	(0)
(3)	*	b	a
(4)	+	a	(2)
(5)	+	(4)	(3)

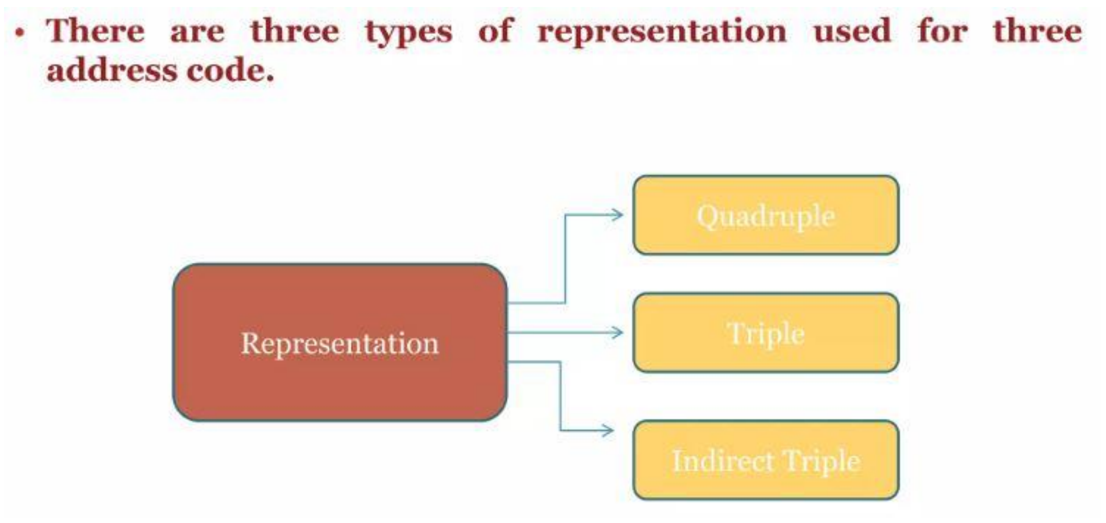
Disadvantage – Moving a statement that define a temporary value requires us to change all references to that statement in arg1 and arg2 arrays. This problem makes triple difficult to use in an optimizing compiler.

Explain Quadruples with an example

- During the translation of a source program into the object code for a target machine, a compiler may generate a middle-level language code, which is known as intermediate code.



- **Three address code** is a type of intermediate code which is easy to generate


• There are three types of representation used for three address code.
Representation
Quadruple
Triple
Indirect Triple

and can be easily converted to machine code.

Optimization:

- Three address code is often used as an intermediate representation of code during optimization phases of the compilation process.
- The three address code allows the compiler to analyze the code and perform optimizations that can improve the performance of the generated code.

Code generation:

- Three address code can also be used as an intermediate representation of code during the code generation phase of the compilation process.
- The three address code allows the compiler to generate code that is specific to the target platform, while also ensuring that the generated code is correct and efficient.

Quadruples :

- In compiler design, quadruples are a form of intermediate code representation. Each quadruple consists of four fields, capturing the essential components of an operation in a program.
- These fields include the operation code (op), two operands (arg1 and arg2), and the result.

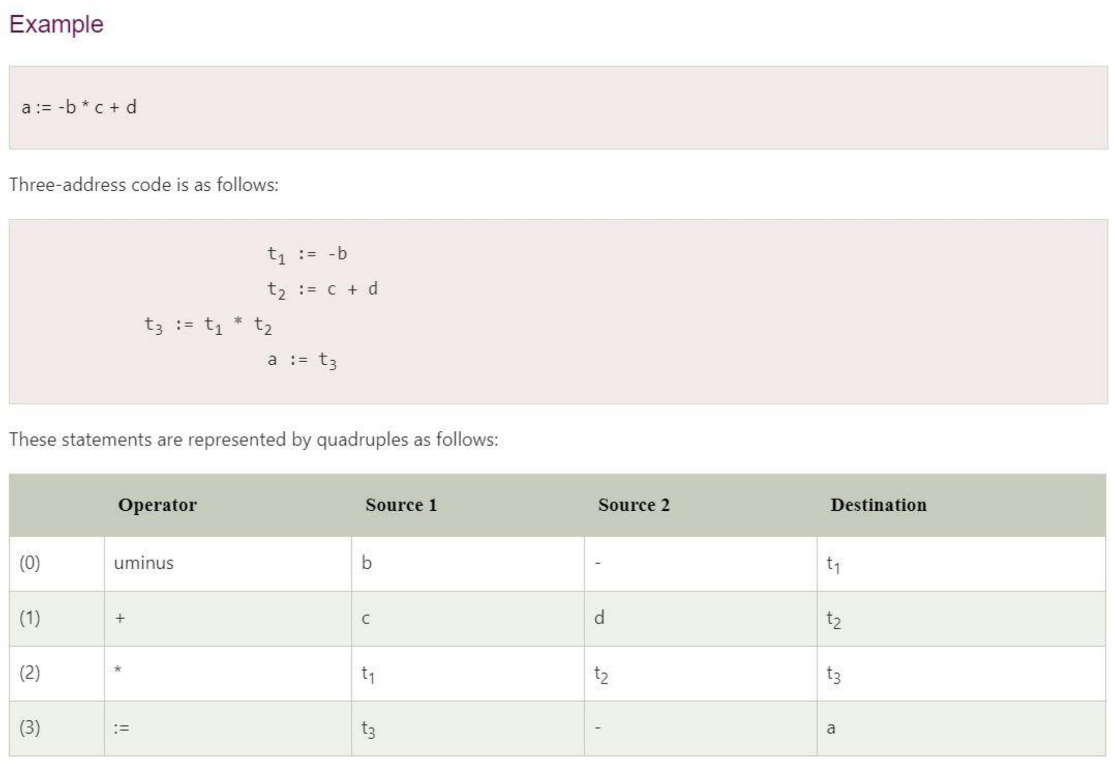
Operator
Source 1
Source 2
Destination

A typical quadruple is represented as (op, arg1, arg2, result), where:

- **op:** Represents the internal code of the operator.
- **arg1 and arg2:** Denote the two operands used in the operation.
- **result:** Represents the result of the expression.

Implementation of Quadruples:

- Enum Type for Operations: An enumerated type is often used to represent the operations (op).
- Pointers to Symbol Table Entries: Symbol table entries are used to represent operands, providing information about the variables used in the code.


Example

```
a := -b * c + d
```

Three-address code is as follows:

$t_1 := -b$

$t_2 := c + d$

$t_3 := t_1 * t_2$

$a := t_3$

These statements are represented by quadruples as follows:

	Operator	Source 1	Source 2	Destination
(0)	uminus	b	-	t_1
(1)	+	c	d	t_2
(2)	*	t_1	t_2	t_3
(3)	:=	t_3	-	a

Explain different parameter passing mechanisms with example

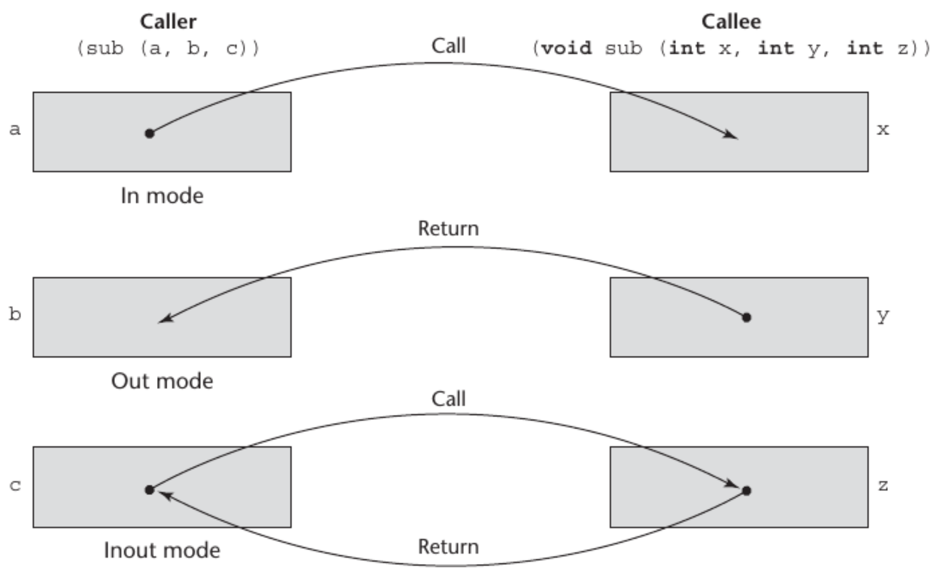
- Parameter passing methods in programming languages determine how parameters are transmitted between called and calling subprograms.

Parameter passing depends on model of subprogram.There are two models for parameter passing-

1. Semantics Models of Parameter Passing
2. Implementation Models of Parameter Passing

Semantics Models of Parameter Passing: Formal parameters are characterized by one of three distinct semantics models:

1. They can receive data from the corresponding actual parameter, called in mode.
2. They can transmit data to the actual parameter, called out mode.
3. They can do both, called inout mode.



Implementation Models of Parameter Passing: This model consist the following ways of parameter passing.

1. **Pass by value:** Value of actual parameter in read only mode is transmitted to formal parameters.

Example :

```
#include <stdio.h>
  void funExample(int a, int b)
{
    a += b;
    printf("In func, a = %d b = %d\n", a, b);
}
int main(void)
{
    int x = 5, y = 7;
    funExample(x, y);
    printf("In main, x = %d y = %d\n", x, y);
    return 0;
}
```

2. **Pass by reference:** Reference/address of actual parameter is transmitted to formal parameters.

```
 #include <stdio.h>
  void swapNum(int* i, int* j)
{
    int temp = *i;
    *i = *j;
    *j = temp;
}
  int main(void)
{
    int a = 15, b = 100;
   swapNum(&a, &b);
    printf("a is %d and b is %d\n", a, b);
    return 0;
}
```

3. **Pass-by-Result:** When a parameter is passed by result, no value is transmitted to the subprogram.

4. **Pass-by-Value-Result:** Pass-by-value-result is an implementation model for inout-mode parameters. Pass-by-value-result is sometimes called pass-by-copy, because the actual

parameter is copied to the formal parameter at subprogram entry and then

copied back at subprogram termination.

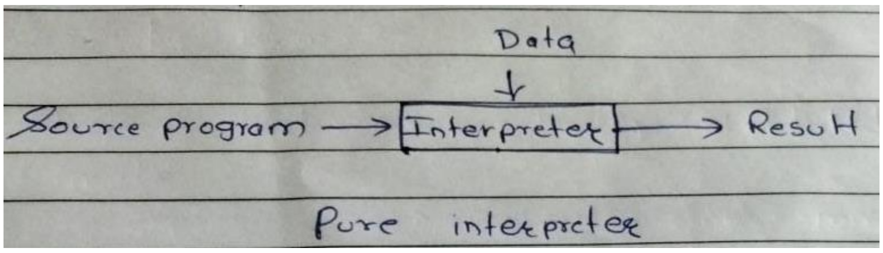
5.Pass-by-Name: Pass-by-name is an inout-mode parameter transmission method. In it parameters are passed by name. Implementing a pass-by-name parameter requires a subprogram to be passed to the called subprogram to evaluate the address or value of the formal parameter.

Explain pure and impure interpreters.

- In the realm of programming language interpreters, the design and functionality can vary based on the treatment of the source program during the interpretation process.
- Two primary approaches, known as pure interpreters and impure interpreters, define how the interpreter handles the source code and performs analysis.

Pure Interpreter:

- A pure interpreter is a type of interpreter that retains the source program in its original form throughout the interpretation process.
- This means that the interpreter directly works with the source code without any preliminary transformation or preprocessing.
- The primary characteristic of a pure interpreter is its immediate execution of source code statements without the creation of an intermediate representation (IR) or any significant analysis performed in advance.
- Pure interpreters Eliminates most of the analysis during interpretation except type analysis.



Advantages of Pure Interpreter:

Simplicity:

- Pure interpreters are generally simpler in design as they operate directly on the source code.
- They don't require an additional compilation step or the generation of intermediate code.
- Immediate execution of source code allows for dynamic behavior and responsiveness during interpretation.

Disadvantages of Pure Interpreter:

Analysis Overheads:

- The lack of preprocessing or intermediate representation may result in higher analysis overheads.
- Type analysis and other checks need to be performed directly on the source, impacting efficiency.
- Reduced Efficiency:

- The absence of an intermediate representation can make pure interpreters less efficient compared to approaches that involve preprocessed code or intermediate code generation.

Impure Interpreter:

- An impure interpreter is a type of interpreter that performs some preliminary processing on the source program before the actual interpretation begins.
- Unlike pure interpreters, impure interpreters often involve a preprocessing step that transforms the source code into an intermediate representation (IR) or another form that facilitates more efficient analysis during interpretation.
- This speeds up interpretation since the code component of the IR that is the IC, can be analyzed more resourcefully than the source form of the program.

Features of Impure Interpreter:

1. **Preliminary Processing:**

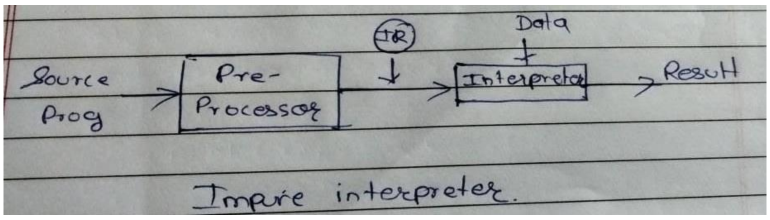
- Impure interpreters perform some level of preliminary processing on the source code before interpreting it.
- This processing step aims to reduce the analysis overhead during interpretation.

2. **Intermediate Representation (IR):**

- The source program is often transformed into an intermediate representation (IR) during preprocessing.
- The IR is an abstract or machine-level representation that is easier to analyze and interpret efficiently.

Advantages of Impure Interpreter:

- The use of an intermediate representation allows for more efficient analysis during interpretation.
- Analysis tasks can be performed on the structured IR, reducing the analysis overhead compared to direct source code interpretation.


Source Prog
Pre-Processor
IR
Data
Interpreter
Result
Impure interpreter.