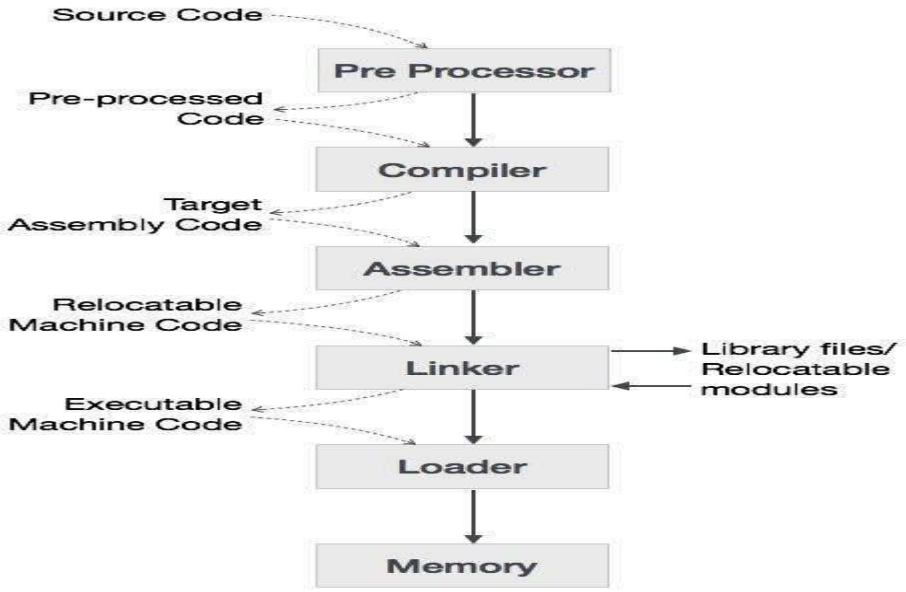
Q1 Write in brief.

a. A **preprocessor** is a program that processes its input data to produce output that is used as input to another program. The output is said to be a preprocessed form of the input data, which is often used by some subsequent programs like compilers. A preprocessor, generally considered as a part of compiler, is a tool that produces input for compilers. It deals with macro-processing, augmentation, file inclusion, language extension, etc.

Function of a preprocessor



A Preprocessor mainly performs three tasks on the HLL code :

- **Removing comments :** It removes all the comments. A comment is written only for the humans to understand the code. So, it is obvious that they are of no use to a machine. So, preprocessor removes all of them as they are not required in the execution and won't be executed as well.
- **File inclusion :** Including all the files from library that our program needs. In HLL we write **#include** which is a directive for the preprocessor that tells it to include the contents of the library file specified.
- **Macro expansion :** Macros can be called as small functions that are not as overhead to process. If we have to write a function (having a small definition) that needs to be called recursively (again and again), then we should prefer a macro over a function.

b. A **syntax error** is an error in the syntax of a sequence of characters or tokens that is intended to be written in a particular programming language.

For compiled languages, syntax errors are detected at compile-time. A program will not compile until all syntax errors are corrected. For interpreted languages, however, a syntax error may be detected during program execution, and an interpreter's error messages might not differentiate syntax errors from errors of other kinds.

Error handling refers to the anticipation, detection, and resolution of programming, application, and communications errors. Specialized programs, called error handlers, are available for some applications. The best programs of this type forestall errors if possible, recover from them when they occur without terminating the application, or (if all else fails) gracefully terminate an affected application and save the error information to a log file.

In programming, a development error is one that can be prevented. Such an error can occur in syntax or logic. Syntax errors, which are typographical mistakes or improper use of special characters, are handled by rigorous proofreading. Logic errors, also called bugs, occur when executed code does not produce the expected or desired result. Logic errors are best handled by meticulous program debugging. This can be an ongoing process that involves, in addition to the traditional debugging routine, beta testing prior to official release and customer feedback after official release.

A run-time error takes place during the execution of a program, and usually happens because of adverse system parameters or invalid input data. An example is the lack of sufficient memory to run an application or a memory conflict with another program. On the Internet, run-time errors can result from electrical noise, various forms of malware or an exceptionally heavy demand on a server. Run-time errors can be resolved, or their impact minimized, by the use of error handler programs, by vigilance on the part of network and server administrators, and by reasonable security countermeasures on the part of Internet users.

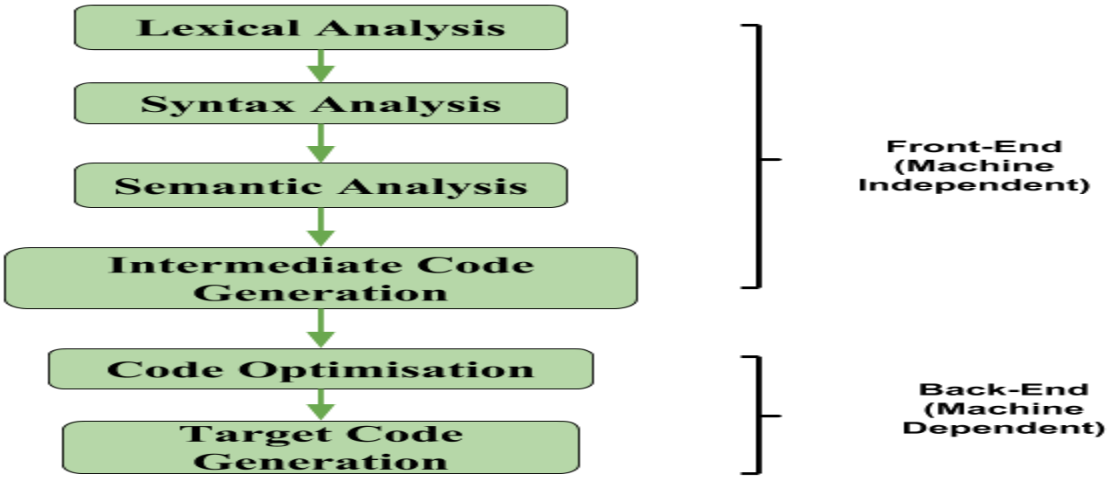
c.A **shift-reduce parser** is a class of efficient, table-driven bottom-up parsing methods for computer languages and other notations formally defined by a grammar. The parsing methods most commonly used for parsing programming languages, LR parsing and its variations, are shift-reduce methods.[1] The precedence parsers used before the invention of LR parsing are also shift-reduce methods. All shift-reduce parsers have similar outward effects, in the incremental order in which they build a parse tree or call specific output actions.

A grammar that is generated to define the mathematical operators is called **operator grammar** with some restrictions on grammar. An **operator precedence grammar** is a context-free grammar that has the property that no production has either an empty right-hand side (null productions) or two adjacent non-terminals in its right-hand side.

Operator precedence parser –
An operator precedence parser is a one of the bottom-up parser that interprets an operator-precedence grammar. This parser is only used for operator grammars. *Ambiguous grammars are not allowed* in case of any parser except operator precedence parser.

d. The benefits of using machine independent intermediate code are:

- Because of the machine independent intermediate code, portability will be enhanced.For ex, suppose, if a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required. Because, obviously, there were some modifications in the compiler itself according to the machine specifications.
- Retargeting is facilitated
- It is easier to apply source code modification to improve the performance of source code by optimising the intermediate code.



e. **Symbol Table** is an important data structure created and maintained by the compiler in order to keep track of semantics of variable i.e. it stores information about scope and binding information about names, information about instances of various entities such as variable and function names, classes, objects, etc.

- It is built in lexical and syntax analysis phases.
- The information is collected by the analysis phases of compiler and is used by synthesis phases of compiler to generate code.
- It is used by compiler to achieve compile time efficiency.
- It is used by various phases of compiler as follows :-
    1. **Lexical Analysis:** Creates new table entries in the table, example like entries about token.
    2. **Syntax Analysis:** Adds information regarding attribute type, scope, dimension, line of reference, use, etc in the table.
    3. **Semantic Analysis:** Uses available information in the table to check for semantics i.e. to verify that expressions and assignments are semantically correct(type checking) and update it accordingly.
    4. **Intermediate Code generation:** Refers symbol table for knowing how much and what type of run-time is allocated and table helps in adding temporary variable information.
    5. **Code Optimization:** Uses information present in symbol table for machine dependent optimization.
    6. **Target Code generation:** Generates code by using address information of identifier present in the table.

Symbol Table entries – Each entry in symbol table is associated with attributes that support compiler in different phases.

Items stored in Symbol table:

- Variable names and constants
- Procedure and function names
- Literal constants and strings
- Compiler generated temporaries
- Labels in source languages

Information used by compiler from Symbol table:

- Data type and name
- Declaring procedures
- Offset in storage
- If structure or record then, pointer to structure table.
- For parameters, whether parameter passing by value or by reference
- Number and type of arguments passed to function
- Base Address

f. In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:

- The output code must not, in any way, change the meaning of the program.
- Optimization should increase the speed of the program and if possible, the program should demand less number of resources.
- Optimization should itself be fast and should not delay the overall compiling process.

Efforts for an optimized code can be made at various levels of compiling the process.

- At the beginning, users can change/rearrange the code or use better algorithms to write the code.
- After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.

- While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

Q2 a.

b. The tasks of the **Error Handling** process are to detect each error, report it to the user, and then make some recover strategy and implement them to handle error. During this whole process processing time of program should not be slow. An **Error** is the blank entries in the symbol table.

Types or Sources of Error – There are two types of error: run-time and compile-time error:

1. A **run-time error** is an error which takes place during the execution of a program, and usually happens because of adverse system parameters or invalid input data. The lack of sufficient memory to run an application or a memory conflict with another program and logical error are example of this. Logic errors, occur when executed code does not produce the expected result. Logic errors are best handled by meticulous program debugging.
2. **Compile-time errors** rises at compile time, before execution of the program. Syntax error or missing file reference that prevents the program from successfully compiling is the example of this.

Classification of Compile-time error –

1. **Lexical** : This includes misspellings of identifiers, keywords or operators
2. **Syntactical** : missing semicolon or unbalanced parenthesis
3. **Semantical** : incompatible value assignment or type mismatches between operator and operand
4. **Logical** : code not reachable, infinite loop.

Finding error or reporting an error – Viable-prefix is the property of a parser which allows early detection of syntax errors.

- **Goal:** detection of an error as soon as possible without further consuming unnecessary input
- **How:** detect an error as soon as the prefix of the input does not match a prefix of any string in the language.
- **Example:** for(**;**), this will report an error as for have two semicolons inside braces.

C. The LR parser is a non-recursive, shift-reduce, bottom-up parser. It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique. LR parsers are also known as LR(k) parsers, where L stands for left-to-right scanning of the input stream; R stands for the construction of right-most derivation in reverse, and k denotes the number of lookahead symbols to make decisions.

There are three widely used properties available for constructing an LR parser:

- SLR(1) – Simple LR Parser:
    - Works on smallest class of grammar
    - Few number of states, hence very small table
    - Simple and fast construction
- LR(1) – LR Parser:
    - Works on complete set of LR(1) Grammar
    - Generates large table and large number of states
    - Slow construction
- LALR(1) – Look-Ahead LR Parser:
    - Works on intermediate size of grammar

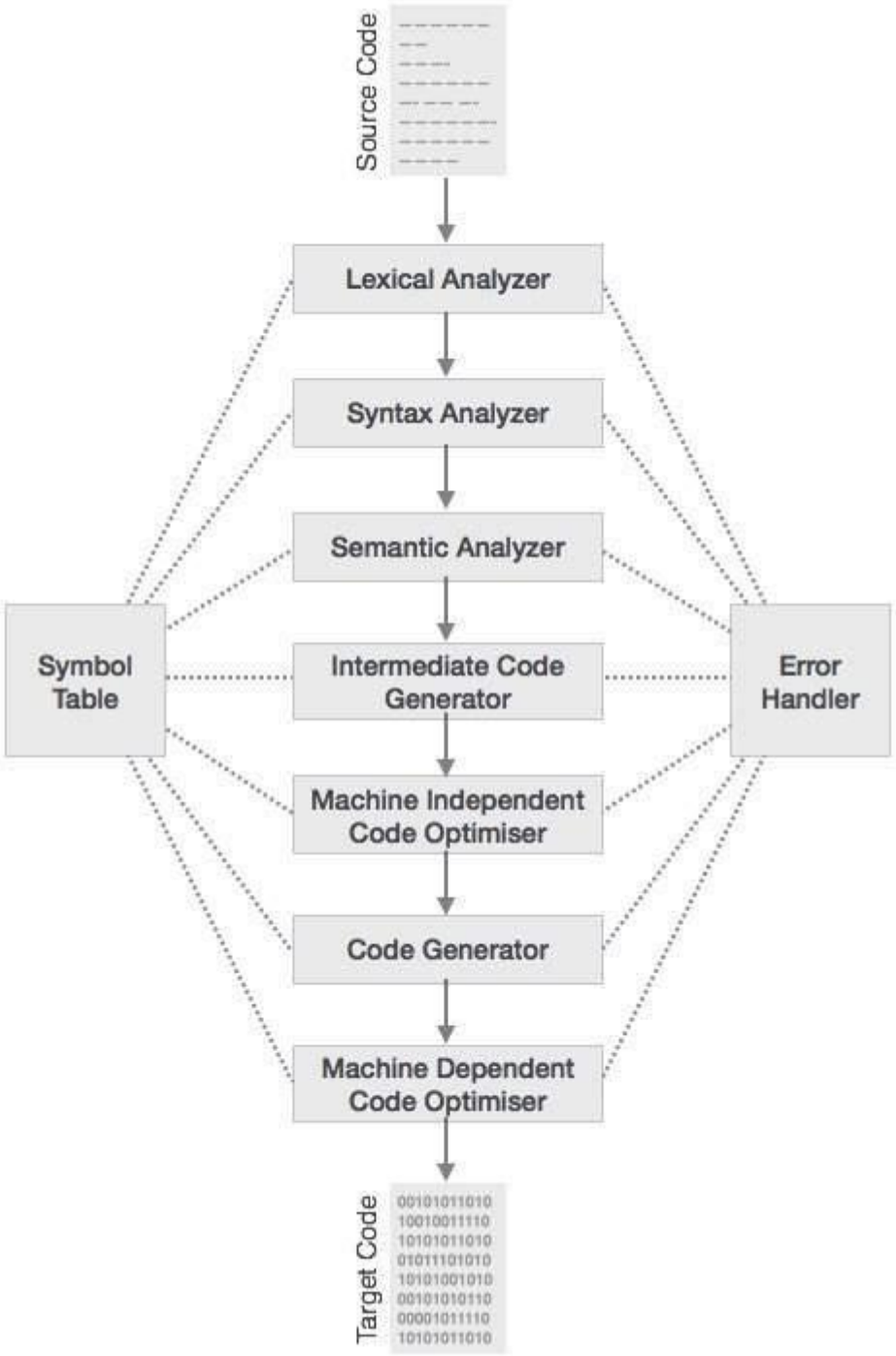
d. **Semantic analysis** or **context sensitive analysis** is a process in compiler construction, usually after parsing, to gather necessary semantic information from the source code. It usually includes type checking, or makes sure a variable is declared before use which is impossible to describe in the extended Backus–Naur form and thus not easily detected during parsing.

In that part we need to check for the type of errors.

- Any used variable must have been declared first in its intended scope.
- Catch multiple declarations of the same variable within the same scope.
- Catch unequivalent types. For simplicity, we all assume that this error happens whenever A = B occurs and both A and B are of different types. Note that sometimes different types do not mean an error as they can be changed through coersion or casting. But we are not considering this here. Also remember that a type of a function is the type of the result it returns.

e. **Global optimization** refers to finding the optimal value of a given function among all possible solution whereas **local optimization** finds the optimal value within the neighboring set of candidate solution

Q3(a). The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.



Lexical Analysis

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

```
<token-name, attribute-value>
```

Syntax Analysis

The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

Semantic Analysis

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

Intermediate Code Generation

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

Code Optimization

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code

lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

Code Generation

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

Symbol Table

It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

b. **Token:** Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are,

1) Identifiers 2) keywords 3) operators 4) special symbols 5)constants

Pattern: A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.

Lexeme: A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

Example:

Description of token

Token	lexeme	pattern
const	const	const
if	if	if
relation	<,<=,= ,< >,>=,>	< or <= or = or < > or >= or letter followed by letters & digit

i	pi	any numeric constant
nun	3.14	any character b/w “and “except"
literal	"core"	pattern

Q4(a). **peephole optimization** is a kind of optimization performed over a very small set of instructions in a segment of generated code. The set is called a "peephole" or a "window". It works by recognizing sets of instructions that can be replaced by shorter or faster sets of instructions.

Common techniques applied in peephole optimization:[1]

- Null sequences – Delete useless operations.
- Combine operations – Replace several operations with one equivalent.
- Algebraic laws – Use algebraic laws to simplify or reorder instructions.
- Special case instructions – Use instructions designed for special operand cases.
- Address mode operations – Use address modes to simplify code.

characteristics of peephole optimization

a. **Redundant instruction elimination.**
b. **Flow of control optimization**
c. **Algebraic simplifications**
d. **Use of machine idioms**

b. Source codes generally have a number of instructions, which are always executed in sequence and are considered as the basic blocks of the code. These basic blocks do not have any jump statements among them, i.e., when the first instruction is executed, all the instructions in the same basic block will be executed in their sequence of appearance without losing the flow control of the program.

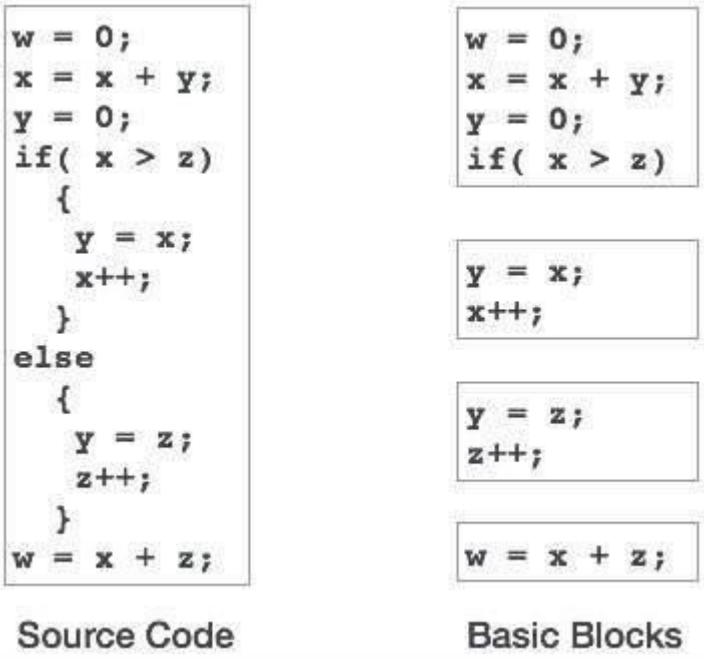
A program can have various constructs as basic blocks, like IF-THEN-ELSE, SWITCH-CASE conditional statements and loops such as DO-WHILE, FOR, and REPEAT-UNTIL, etc.

Basic block identification

We may use the following algorithm to find the basic blocks in a program:

- Search header statements of all the basic blocks from where a basic block starts:
  - First statement of a program.
  - Statements that are target of any branch (conditional/unconditional).
  - Statements that follow any branch statement.
- Header statements and the statements following them form a basic block.
- A basic block does not include any header statement of any other basic block.

Basic blocks are important concepts from both code generation and optimization point of view.



Basic blocks play an important role in identifying variables, which are being used more than once in a single basic block. If any variable is being used

more than once, the register memory allocated to that variable need not be emptied unless the block finishes execution.

Q5(a). **copy propagation** is the process of replacing the occurrences of targets of direct assignments with their values.[1] A direct assignment is an instruction of the form `x = y`, which simply assigns the value of `y` to `x`.

From the following code:

```
y = x
z = 3 + y
```

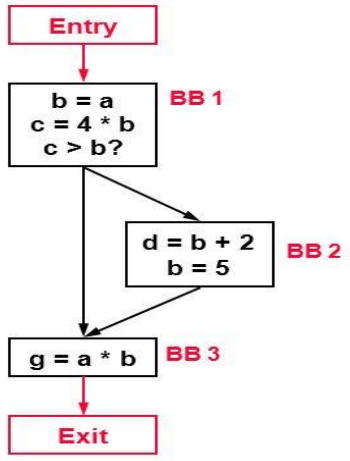
Copy propagation would yield:

```
z = 3 + x
```

Copy propagation often makes use of reaching definitions, use-def chains and def-use chains when computing which occurrences of the target may be safely replaced. If all upwards exposed uses of the target may be safely modified, the assignment operation may be eliminated.

Copy Propagation

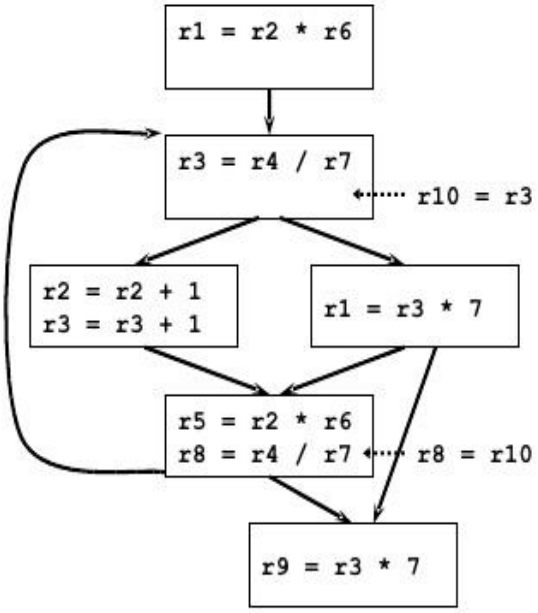
- A ***copy instruction*** is an instruction in the form x = y.
- ***Copy propagation*** replaces later uses of x with uses of y provided intervening instructions do not change the value of either x or y.
- Benefit: saves computations, reduces space; enables other transformations.



b. **Common subexpression elimination** (CSE) is a compiler optimization technique of finding redundant expression evaluations, and replacing them with a single computation . This saves the time overhead resulted by evaluating the expression for more than once . We will have a look into this phenomenon by considering a simple code and again taking a walk through its assembly code .

12

Global: Common Subexpression Elimination (CSE)



- Goal: eliminate recomputations of an expression
- Rules:
  1. X and Y have the same opcode and X dominates Y
  2. src(X) = src(Y) for all srcs
  3. For all srcs, no def of a src on any path between X and Y (excluding Y)
  4. Insert rx = dest(X) immediately after X for new register rx
  5. Replace Y with move dest(Y) = rx

c.