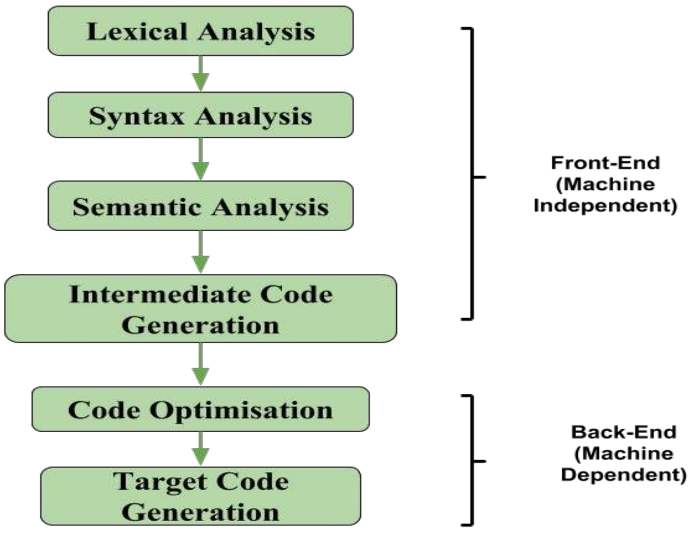
UNIT V: Code Generation: Issues in the Design of a Code Generator, Object Code Forms, Code Generation Algorithm, Register Allocation and Assignment.

Introduction


Lexical Analysis
Syntax Analysis
Semantic Analysis
Intermediate Code Generation
Code Optimisation
Target Code Generation
Front-End (Machine Independent)
Back-End (Machine Dependent)

Code generation can be considered as the final phase of compilation. Through post code generation, optimization process can be applied on the code, but that can be seen as a part of code generation phase itself.
Code generator converts the intermediate representation of source code into a form that can be readily executed by the machine. A code generator is expected to generate the correct code. Designing of the code generator should be done in such a way that it can be easily implemented, tested, and maintained.
The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language. We have seen that the source code written in a higher-level language is transformed into a lower-level language that results in a lower-level object code, which should have the following minimum properties:

- It should carry the exact meaning of the source code.
- It should be efficient in terms of CPU usage and memory management.

Q) Explain various issues that affect the efficiency of generated code

Issues in the Design of a Code Generator

In the code generation phase, various issues can arise:

1. Input to the code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation.

6. Evaluation order

1.Input to code generator – The input to the code generator is the intermediate code generated by the front end, along with information in the symbol table that determines the run-time addresses of the data objects denoted by the names in the intermediate representation. Intermediate codes may be represented mostly in quadruples, triples, indirect triples, Postfix notation, syntax trees, DAGs, etc. The code generation phase just proceeds on an assumption that the input is free from all syntactic and state semantic errors, the necessary type checking has taken place and the type-conversion operators have been inserted wherever necessary.

- The input to the code generator contains the intermediate representation of the source program and the information of the symbol table. The source program is produced by the front end.
- Intermediate representation has the several choices:
  a) Postfix notation
  b) Syntax tree
  c) Three address code.
- We assume front end produces low-level intermediate representation i.e. values of names in it can directly manipulated by the machine instructions.
- The code generation phase needs complete error-free intermediate code as an input requires.

2. Target program:

The target program is the output of the code generator. The output may be absolute machine language, relocatable machine language, or assembly language.

The output can be:

1. **Assembly language:** It allows subprogram to be separately compiled.
2. **Relocatable machine language:** It makes the process of code generation easier.
3. **Absolute machine language:** It can be placed in a fixed location in memory and can be executed immediately.

3. Memory management

- During code generation process the symbol table entries must be mapped to actual p addresses and levels must be mapped to instruction address.
- Mapping name in the source program to address of data is co-operating done by the front end and code generator.
- Local variables are stack allocation in the activation record while global variables are in static area.

4. Instruction selection**:**

- Nature of instruction set of the target machine should be complete and uniform.
- When you consider the efficiency of target machine then the instruction speed and machine idioms are important factors.
- The quality of the generated code can be determined by its speed and size.

Example:

The Three address code is:

1. a:= b + c
2. d:= a + e

Inefficient assembly code is:

1. MOV b, R0 R0→b
2. ADD c, R0 R0 c + R0
3. MOV R0, a a → R0
4. MOV a, R0 R0→ a
5. ADD e, R0 R0 → e + R0
6. MOV R0, d d → R0

5. Register allocation

Register can be accessed faster than memory. The instructions involving operands in register are shorter and faster than those involving in memory operand.

The following sub problems arise when we use registers:

Register allocation: In register allocation, we select the set of variables that will reside in register.

Register assignment: In Register assignment, we pick the register that contains variable.

Certain machine requires even-odd pairs of registers for some operands and result.

For example:

Consider the following division instruction of the form:

1. D x, y

Where,

x is the dividend even register in even/odd register pair

y is the divisor

Even register is used to hold the reminder.

Old register is used to hold the quotient.

6. Evaluation order

The efficiency of the target code can be affected by the order in which the computations are performed. Some computation orders need fewer registers to hold results of intermediate than others.

Disadvantages in the design of a code generator:

Limited flexibility: Code generators are typically designed to produce a specific type of code, and as a result, they may not be flexible enough to handle a wide range of inputs or generate code for different target platforms. This can limit the usefulness of the code generator in certain situations.

Maintenance overhead: Code generators can add a significant maintenance overhead to a project, as they need to be maintained and updated alongside the code they generate. This can lead to additional complexity and potential errors.

Debugging difficulties: Debugging generated code can be more difficult than debugging hand-written code, as the generated code may not always be easy to read or understand. This can make it harder to identify and fix issues that arise during development.

Performance issues: Depending on the complexity of the code being generated, a code generator may not be able to generate optimal code that is as performant as hand-written code. This can be a concern in applications where performance is critical.

Learning curve: Code generators can have a steep learning curve, as they typically require a deep understanding of the underlying code generation framework and the programming languages being used. This can make it more difficult to onboard new developers onto a project that uses a code generator.

Over-reliance: It's important to ensure that the use of a code generator doesn't lead to over-reliance on generated code, to the point where developers are no longer able to write code manually when necessary. This can limit the flexibility and creativity of a development team and may also result in lower quality code overall.

Q)write the algorithm for simple code generator

The simple code generator algorithm generates target code for a sequence of three-address statements. The code generator algorithm works by considering individually all the basic blocks. It uses the next-use information to decide on whether to keep the computation in the register or move it to a variable so that the register could be reused. The computation of next-use information is explained in the previous module. We assume that for each operator in the input statement there is a target-language operator. It uses new function getreg() to assign registers to variables. The algorithm for code generation initially checks if operands to three-address code are available in registers. After computing the results of two operations, the results are kept in registers till the result is required by another computation or register is kept up to a procedure call or end of block to avoid errors.

Code generator is used to produce the target code for three-address statements. It uses registers to store the operands of the three-address statement.

Example:

Consider the three-address statement x: = y + z. It can have the following sequence of codes:

MOV x, R_0

ADD y, R_0

Register and Address Descriptors:

- A register descriptor contains the track of what is currently in each register. The register descriptors show that all the registers are initially empty.
- An address descriptor is used to store the location where current value of the name can be found at run time.

A simple code-generation algorithm

The algorithm takes a sequence of three-address statements as input. For each three-address statement of the form a: = b op c performs the various actions. These are as follows:

1. Invoke a function getreg to find out the location L where the result of computation b op c should be stored.
2. Consult the address description for y to determine y'. If the value of y currently in memory and register both then prefer the register y' . If the value of y is not already in L then generate the instruction **MOV y' , L** to place a copy of y in L.
3. Generate the instruction **OP z' , L** where z' is used to show the current location of z. if z is in both then prefer a register to a memory location. Update the address descriptor of x to indicate that x is in location L. If x is in L then update its descriptor and remove x from all other descriptor.
4. If the current value of y or z have no next uses or not live on exit from the block or in register then alter the register descriptor to indicate that after execution of x : = y op z those register will no longer contain y or z.

```
The Code Generation Algorithm
Algorithm  SimpleCodeGenerator( )
Input : Sequence of 3-address statements from a basic block.
Output: Assembly language code
For each statement x := y op z
1. Set location L = getreg(y, z) to store the result of y op z
2. If y does not belong to L, then generate
       MOV y’,L
where y’ denotes one of the locations where the value of y is available - choose register if possible
3. Generate
        OP z’,L
where z’ is one of the locations of z;
Update register/address descriptor of x to include L
```

4. If y and/or z has no next use and is stored in register, update register descriptors to remove y and/or z

Generating Code for Assignment Statements:

The assignment statement d:= (a-b) + (a-c) + (a-c) can be translated into the following sequence of three address code:

```
 t:= a-b
u:= a-c
 v:= t +u
d:= v+u
```

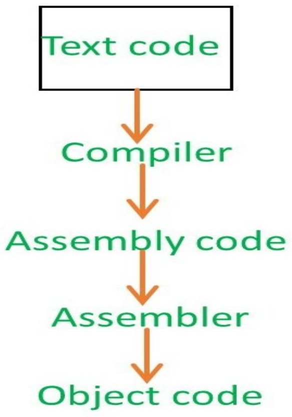
Code sequence for the example is as follows:

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
v:= t + u	ADD R1, R0	R0 contains v R1 contains u	u in R1 v in R1
d:= v + u	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 d in R0 and memory

Q) what are the object code forms in code generation and explain?

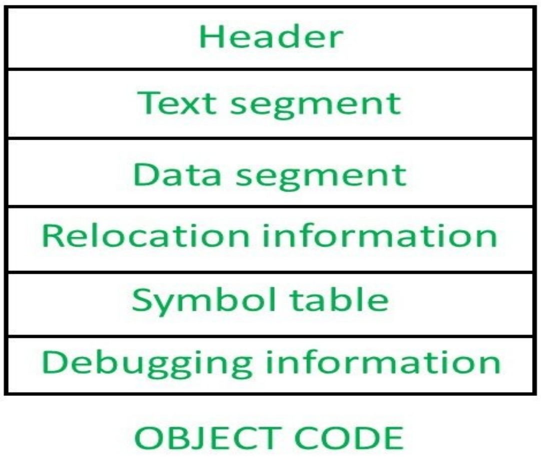
Object Code Forms

object code is the machine-readable code that is generated by the compiler, and it serves as an intermediate step between the source code and the final executable code. Object code files are specific to the target architecture and operating system and are typically stored in a binary file format.



But, when you compile a program, then you are not going to use both compiler and assembler. You just take the program and give it to the compiler and compiler will give you the directly executable code. The compiler is actually combined inside the assembler along with loader and linker. So all the module kept together in the compiler software itself. So when you calling gcc, you are actually not just calling the compiler, you are calling the compiler, then assembler, then linker and loader. Once you call the compiler, then your object code is going to present in Hard-disk.

This object code contains various part –



Object code is the output of a compiler after it has translated the source code into machine code. It is a binary representation of the program that can be executed directly by the computer's CPU.

There are two types of object code forms that a compiler can generate:

Relocatable Object Code: This type of object code can be loaded into any memory location and can be linked with other object files to create an executable file. It contains position-independent code that can be relocated to any memory address.

Executable Object Code: This type of object code is ready to be executed by the CPU. It contains absolute memory addresses that cannot be changed. It cannot be linked with other object files to create an executable file.

In summary, the object code is the output of a compiler that is generated after translating the source code into machine code. It can be either relocatable or executable, depending on the type of object code form generated by the compiler.

After compilation of the source code, the object code is generated, which not only contains machine level instructions but also information about hardware registers, memory address of some segment of the run-time memory (RAM), information about system resources, read-write permissions ..etc.

- The output of Code Generation (CG) is the target language.
- The output may take different forms like absolute machine language, re-locatable machine language or assembly language.

The operations performed in the target code are:

1. Load operation (LD)

- General format is LD dst, addr.
- LD loads the value in location 'addr' into location 'dst'. Here dst is destination.
- For example, LD R1, x ☾ loads the value in location x into register R1.

2. Store operation (ST)

- General format is ST dst, src.
- For example, ST x, R1 ☾ stores the value in register R1 into the location x.

3. Computation operation

- The general form is OP dst, src1, src2 where OP is a operator like ADD, SUB.
- For example, ADD R1, R2, R3 ☾ adds R2 and R3 values and stores into R1.

4. Unconditional jump (BR)

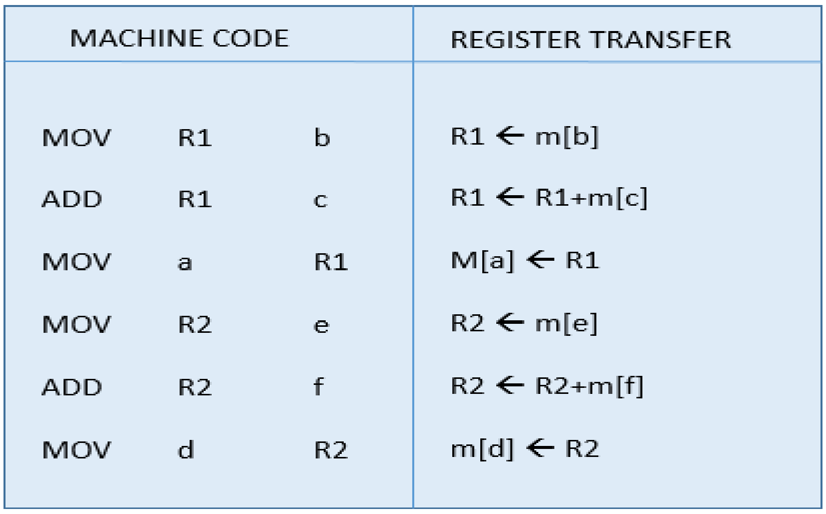
The general form is BR L where BR is branch. This causes control to branch to the machine instruction with label L.

5. Conditional jump

- The general form is Bcond R, L where R is a register, L is label and cond stands for any of the common tests on values in register R.

For example, BGTZ R2, L ☾ This instruction causes a jump to label L if the value in register R2 is greater than zero and allows control to pass to the next machine instruction if not.

The below example performs target codea=b+c; d=e+f;



MACHINE CODE			REGISTER TRANSFER
MOV	R1	b	R1 ← m[b]
ADD	R1	c	R1 ← R1+m[c]
MOV	a	R1	M[a] ← R1
MOV	R2	e	R2 ← m[e]
ADD	R2	f	R2 ← R2+m[f]
MOV	d	R2	m[d] ← R2

Addressing modes in the Target code:

1. Direct addressing
2. Indirect addressing
3. Immediate addressing

4. Indexed addressing

Direct addressing

MOV 1000, R0 ------move the content of data at the location 1000 into R0

Indirect addressing

MOV (R1), R0------- move the content of the data at the location addressed by R1 into R0

Immediate addressing

MOV #1000, R0------move the immediate data 1000 into R0

Indexed addressing

(a). MOV 1000(R1), R0 ---------move the content of data at(R1+1000)into R0

(b). MOV * 1000(R1), R0-------- move the content of the data at a location whose address is at (R1+1000) into R0

Q) Discuss about register allocation and assignment in target code generation?

REGISTER ALLOCATION AND ASSIGNMENT

Registers are the fastest locations in the memory hierarchy. But unfortunately, this resource is limited. It comes under the most constrained resources of the target processor. Register allocation is an NP-complete problem. However, this problem can be reduced to graph colouring to achieve allocation and assignment. Therefore, a good register allocator computes an effective approximate solution to a hard problem.

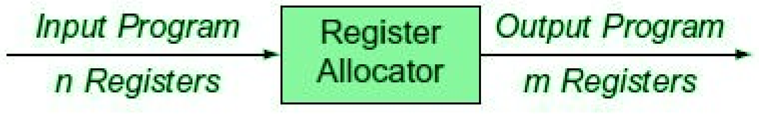


Figure – Input-Output

The register allocator determines which values will reside in the register and which register will hold each of those values. It takes as its input a program with an arbitrary number of registers and produces a program with a finite register set that can fit into the target machine.

Allocation vs Assignment:

Allocation –

Maps an unlimited namespace onto that register set of the target machine.

- **Reg. to Reg. Model:** Maps virtual registers to physical registers but spills excess amount to memory.
- **Mem. to Mem. Model:** Maps some subset of the memory location to a set of names that models the physical register set.

Allocation ensures that code will fit the target machine's reg. set at each instruction.

Assignment –

Maps an allocated name set to the physical register set of the target machine.

- Assumes allocation has been done so that code will fit into the set of physical registers.

- No more than **‘k’** values are designated into the registers, where ‘k’ is the no. of physical registers.

What are the approaches for Register Allocation?
There are two approaches, which are.

- Local allocators: These are based on usage counts.
- Global or intraprocedural allocators: These are based on the concept of graph coloring.

Local Register Allocation and Assignment:
Allocation just inside a basic block is called Local Reg. Allocation. Two approaches for local reg. allocation: Top-down approach and bottom-up approach.
Top-Down Approach is a simple approach based on ‘Frequency Count’. Identify the values which should be kept in registers, and which should be kept in memory.

Algorithm:

1. Compute a priority for each virtual register.
2. Sort the registers into priority order.
3. Assign registers in priority order.
4. Rewrite the code.

Moving beyond single Blocks:

- More complicated because the control flow enters the picture.
- Liveness and Live Ranges: Live ranges consist of a set of definitions and uses that are related to each other as they are i.e. no single register can be common in a such couple of instruction/data.

Following is a way to find out Live ranges in a block. A live range is represented as an interval [i,j], where i is the definition and j is the last use.

Global Register Allocation and Assignment:
1. The main issue of a register allocator is minimizing the impact of spill code.

- Execution time for spill code.
- Code space for spill operation.
- Data space for spilled values.

2. Global allocation can’t guarantee an optimal solution for the execution time of spill code.
3. Prime differences between Local and Global Allocation:

- The structure of a global live range is naturally more complex than the local one.
- Within a global live range, distinct references may execute a different number of times. (When basic blocks form a loop)

4. To make the decision about allocation and assignments, the global allocator mostly uses graph coloring by building an interference graph.
5. Register allocator then attempts to construct a k-coloring for that graph where ‘k’ is the no. of physical registers.

- In case, the compiler can’t directly construct a k-coloring for that graph, it modifies the underlying code by spilling some values to memory and tries again.
- Spilling simplifies that graph which ensures that the algorithm will halt.

6. Global Allocator uses several approaches; however, we'll see top-down and bottom-up allocation's strategies. Subproblems associated with the above approaches.
    - Discovering Global live ranges.
    - Estimating Spilling Costs.
    - Building an Interference graph.

Register allocation by graph coloring.

Global register allocation can be seen as a graph coloring problem.

Basic idea:

- Identify the live range of each variable.
- Build an interference graph that represents conflicts between live ranges (two nodes are connected if the variables they represent are live at the same moment)
- Try to assign as many colors to the nodes of the graph as there are registers so that two neighbors have different colors

Fig Flow graph of an inner loop

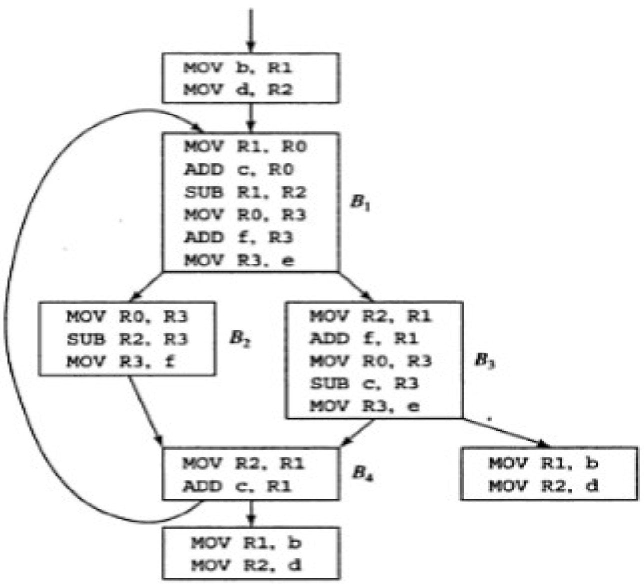


Fig Code sequence using global register assignment.

Q) Give an example to show how DAG is used for register allocation?

DAG for Register Allocation

Code generation from DAG is much simpler than the linear sequence of three address code.

With the help of **DAG one can rearrange sequence of instructions and generate and efficient code**

There exist various algorithms which are used for generating code from DAG.

They are: Code Generation from DAG:-

Rearranging Order

Heuristic Ordering

Labeling Algorithm

Rearranging Order

These address code's order affects the cost of the object code which is being generated.

Object code with minimum cost can be achieved by changing the order of computations.

Example:

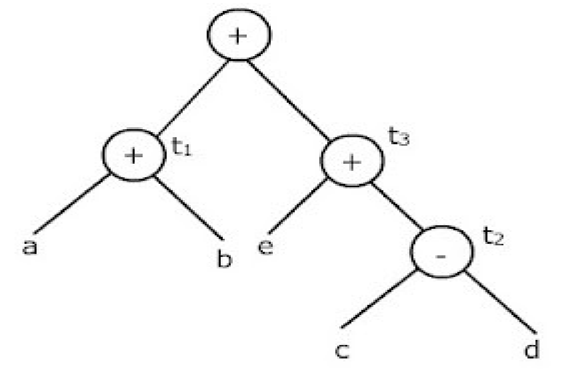
t1:= a + b

t2:= c – d

t3:= e + t2

t4:= t1 + t3

For the expression (a+b) + (e+(c-d)), a DAG can be constructed for the above sequence as shown below



DAG for *(a + b) + (e + (c - d))*

The code is thus generated by translating the three address code line by line

```
MOV a, R0
ADD b, R0
MOV c, R1
SUB d, R1
MOV R0, t          t1:= a+b
MOV e, R0          R1 has c-d
ADD R0, R1           /* R1 contains e + (c – d)*/
MOV t1, R0         /R0 contains a + b*/
ADD R1, R0
MOV R0, t4
```

Now, if the ordering sequence of the three address code is changed

```
t2:= c - d
t3:= e + t2
t1:= a + b
t4:= t1 + t3
```

Then, an improved code is obtained as:

```
MOV c, R0
SUB D, R0
MOV e, R1
ADD R0, R1
MOV a, R0
```

```
        ADD b, R0
        ADD R1, R0
        MOV R0, t4
```

Heuristic Ordering

The algorithm displayed below is for heuristic ordering. It lists the nodes of a DAG such that the node's reverse listing results in the computation order.

```
{
While unlisted interior nodes remain
do
{
select an unlisted node n, all of whose parents have been listed;
List n;
While the leftmost child 'm' of 'n' has no unlisted parents and is not a leaf
do
{
/*since na was just listed, surely m is not yet listed*/
}
{
list m
n =m;
}
}
order = reverse of the order of listing of nodes
}
```

Labeling Algorithm

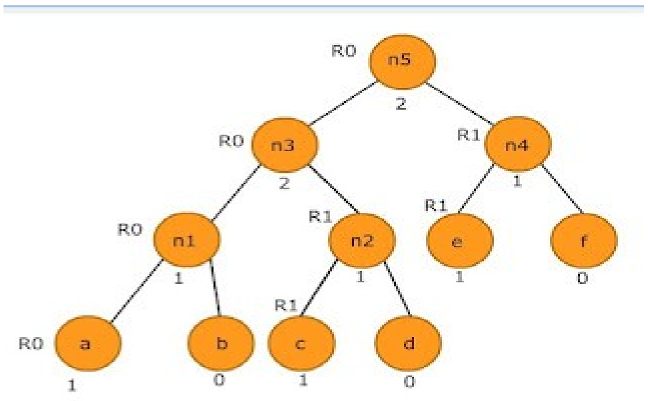
Labeling algorithm deals with the tree representation of a sequence of three address statements

It could similarly be made to work if the intermediate code structure was a parse tree. This algorithm has two parts:

- The first part labels each node of the tree from the bottom up, with an integer that denotes the minimum number of registers required to evaluate the tree, and with no storing of intermediate results.
- The second part of the algorithm is a tree traversal that travels the tree in an order governed by the computed labels in the first part, and which generates the code during the tree traversal

```
if n is a leaf then
        if n is leftmost child of its parents then
                Label(n) = 1
        else label(n) = 0
else
{
        /*n is an interior node*/
        let n1, n2, …..,nk be the children of ordered by label,
        so label(n1) >= label(n2) >= … >= label(nk);
        label(n) = max (label (ni) + i – 1)
}
```

Example:


R0
n5
2
R0
n3
2
R1
n4
1
R0
n1
1
R1
n2
1
R1
e
1
f
0
R0
a
1
b
0
R1
c
1
d
0