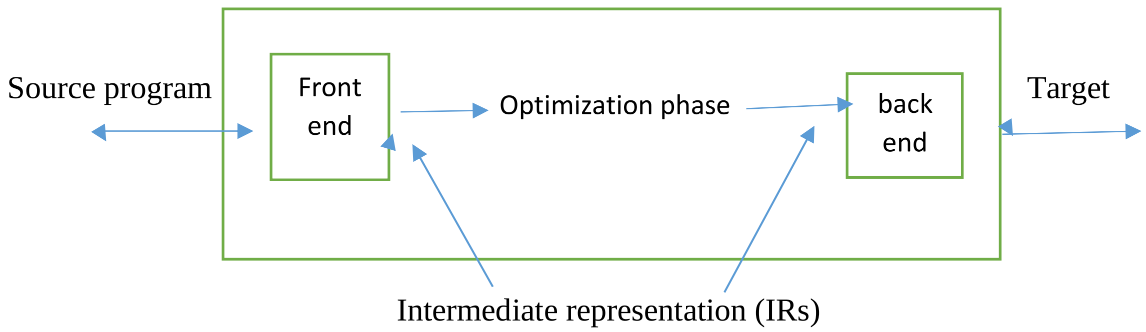
CODE OPTIMIZATION

A compiler performs code optimization to improve the execution efficiency of a program.
Remove unnecessary things in the program .are known as code optimization

It achieves optimization through the following two means:

1. Elimination of redundancies in a program.

2. Rearrangement of computations in a program to make it execute more efficiently. It is axiomatic that code optimization must not change the meaning of a program



Schematic of an optimising compiler

Two practical considerations limit the scope of optimization. First, code optimization seeks to improve a program rather than the algorithm used in a program.

Hence it does not replace an algorithm by a more efficient one. Second, efficient code generation for a specific target machine (eg., by fully exploiting its instruction set) belongs in the back end of a compiler.

Hence it is excluded from the scope of optimization contains a schematic of an optimizing compiler.

It differs from the schematic only in the presence of the optimization phase.

The front end generates an intermediate representation consisting of either triples, quadruples or abstract syntax trees.

The optimization phase analyzes the intermediate representation, performs optimizations and generates an intermediate representation of the optimized program.

The back end of the compiler generates target code from this intermediate representation. Thus, the optimization techniques are independent of both the programming language in which the source program is written and the tr get machine

The compiler extracts clues for code optimization from the structure of a program and the manner in which it manipulates its data.

Code Optimization Techniques

1)Compile time evaluation:

If a compiler can perform certain computations specified in a program during compilation itself, it need not generate code that would perform them during execution of the program.

This way the generated code would execute more efficiently, particularly if such computations occur inside loops.

Constant folding is the primary optimization of this kind. It replaces an expression all of whose operands are constant

s by a single constant.

Accordingly, when the compiler sees an expression such as 3.14157/2, which is /2, on the right hand of the assignment a =3.14157/2 it replaces the assignment by a = 1.570785.

This replacement eliminates a division operation from the target program.

2) Elimination of common subexpressions:

Common subexpressions are those occurrences of expressions in a program that yield the same value.

Csi means common subexpression

ej means elimination

Let CSi designate a set of such occurrences in a program. An expression occurrence ej that is included in CSi can be eliminated if the value of some expression included in CSi would be computed before control would reach the occurrence ej during the program's execution.

The compiler implements this transformation by generating code that would save the value of ej when it is evaluated and use it in the place where ej occurred in the program.

If the program contains if statement and loops, the compiler would have to verify that this condition would be satisfied irrrspective of how control would actually reach the occurrence ej.

Example: a :=b*c compiler →

____

X:= b*c+5.2. **Before optimization**

t ::=b*c; →

a :=t;

________. **After optimization**

X:= t+5.2;

Here CS, contains the two occurrences of b*c because values of b * c are identical in these expressions.

The second occurrence of bee can be eliminated because the first occurrence of b*c would have been executed before the second occurrence would be reached during execution of the program.

Hence the transforms the program to the form shown in the right column.

Now, the value of the first occurrence of bec would be saved in the temporary location t and it would be used in the assignment to x

3)Dead code elimination:

Code that can be omitted from a program without affecting its results is called dead code.

In the statement x:= <exp>, the assignment to x is a dead code if the value it assigns to x is not used anywhere in the program, that is, either x is not used in the program or it is used only after it has been assigned some other value.

<exp> constitutes dead code only if its execution does not produce side effects, ie, only if it does not contain function or procedure calls that can produce side effects.

example

```
for (i=0; i<100;i++)

{

  a=0;

}
```

4)Frequency reduction/code movement:-Execution time of a program can be reduced by moving code that is situated in a part of the program that is executed very frequently to another part of the program that is executed less frequently.

For example, the transformation of loop optimization moves loop invariant code out of a loop and places it prior to loop entry.

example loop optimization

for i:=1 to 100 do	x:= 25*a;
begin	for i := 1 to 100 do
z:=i;	**begin**
x:=25*a;	z := i;
y:=x+z;	y :=x+z;
end;	**end;**
Before optimization	**After optimization**

Here x = 25*a; is loop invariant because the value of a is not changed inside the for loop, so it is moved to a place prior to loop entry.

This way it is computed only once before entering the loop. y: = x*z; is not loop invariant. Hence it cannot be subjected to frequency reduction

5) Strength reduction

The strength reduction optimization replaces a time-consuming operation (a "high strength operation) in an expression by a faster operation (a 'low strength" opera- tion), e.g., replacement of a multiplication by an addition.

Example (Strength reduction) The following program computes i*5 within the loop for i

```
for i :=1 to 10 do
begin
—------
k:=i*5;
—------
end;
```

before optimization

```
itemp :=5;
for i:=1 to 10 do
begin
 —-----
k:= itemp;
—----
itemp:=itemp+5;
end;
```

after optimization

Because the value of i would vary from 1 to 10 in steps of 1 during the program's execution, values of 1*5 in successive iterations would differ by 5.

Hence the expression i*5 occurring within the loop can be replaced by the expression itemp+5 within the loop and an initialization of itemp before entering the loop.

During execution of the optimized program, the variable itemp would track the value of i*5 through repeated additions

Local and global optimization

Optimization of a program is performed in the following two phases:

1. Local optimization: This phase applies optimizing transformations over small sections of a program that consist of a few statements each. It incurs a mod est program analysis cost because only a few statements in the program are analyzed at a time; however, it also has limited benefits.

2. Global optimization: This phase applies optimizing transformations over a complete function or procedure. It incurs a higher program analysis cost but also provides larger benefits.

Local Optimization

The scope of local optimization is a basic block which is an 'essentially sequential section of code segment in the source program.

The cost of local optimization is low because the sequential nature of the basic block simplifies the analysis needed for optimization. The benefits are limited because certain optimizations such as loop optimization are beyond the scope of local optimization.

Basic block- A basic block is a sequence of program statements (s1,s2 ...Sn) such that only sn can be a transfer of control statement and only s1can be the destination of a transfer of control statement.

Thus, a basic block bi, is a section of code that has a single entry point.

If control reaches statement s1 during program execution, all statements s1,s2,..... sn will be executed.

The 'essentially sequential' nature of a basic block simplifies local optimization.discusses this aspect. Use of basic blocks for local optimization also simplifies global optimization. For example, let a basic block bi contains n occurrences of an expression a+b.

After local common subexpression elimination has been performed over bi, global optimization needs to consider elimination of only the first occurrence of a+b-other occurrences of a+b are either not redundant or would have been already eliminated!

local optimization

consider the local optimization

a:=x*y;	t:= x*y
—----	a:=t;
b:=x*y;	—-------
labi: c:=x*y;	b:= t;
	labi : c:=x*y;
before optimization	**After optimization**

Local optimization identifies two basic blocks in the program because presence of the label labi implies the possibility of a control transfer to the statement shown against that label.

The first basic block extends up to the end of the statement b:= x*y; Its optimization leads to elimination of the second occurrence of x*y.

The second basic block contains only the last statement. The occurrence of x*y in it cannot be eliminated because it may be reached without executing the first basic block.

Global Optimization

Compared to local optimization, global optimization requires more program analy sis effort to establish the feasibility of an optimization. Consider global common subexpression elimination.

if some expression x*y occurs in a set of basic blocks SB of program P. its occurrence in a block bj be eliminated if the following two conditions are satisfied for every possible execution of P:

1. Basic block bj is executed only after some block by that is included in SB has been executed.

2. No assignments to x or y have been executed after the last (or only)

SB means set of basic block

evaluation of x*y in block b ensures that x *y would be evaluated before execution reaches block bj, while condition 2 ensures that the value resulting from that evaluation would be equivalent to the value of x*y in block bj.

due to presence conditional statements in the program, different blocks in SB may satisfy condition 1 for different executions of P. Common subexpression elimination is realized by saving the value of x*y in a temporary location in all blocks by which satisfy condition 1 and using the saved value in block b, instead of evaluating x*y.

Note the emphasis on the words 'every possible execution. This requirement is introduced to ensure that the meaning of the program is unaffected by the optimization. In this section we use the word 'always' to imply 'in every possible evaluation''.Unlike local optimization, which is performed over a basic block, global optimization is performed over a function or procedure.

Hence the global optimization phase has to consider the effect of conditional statements and loops while analyzing a program. To simplify this task, it performs two kinds of analysis. It first analyzes a program by using the techniques of control flow analysis to find how control may flow during execution of a program and whether loops exist in it.

then analyzes the manner in which variables are likely to be assigned values and referenced during execution by using the techniques of data flow analysis.

- **Parameter passing mechanisms**

  A programming language specifies the semantics of use of a parameter inside a function, thereby defining the kind of side effects a function can produce on its actual parameters.

  A parameter passing mechanism is an arrangement used in the target code to realize passing and accessing of actual parameters and production of side effects on them.

  This section discusses the side effect producing capability and exe- cution efficiency of three common parameter passing mechanisms.

  Call by value

  the call by value parameter passing mechanism passes values of actual parameters to the called function through the parameter list.

  If any statement in the function modifies the value of a formal parameter, the copied value of the corresponding actual parameter would change but the change would not be reflected on the value of the actual parameter.

  Thus a function cannot produce any side effects on its parameters The compiler can implement call by value as follows:

  It can allocate memory to formal parameters of a function as if they were local variables of the function and generate code at the start of a function that would copy values of actual from the parameter list into the corresponding formal parameters.

This way a formal parameter can be accessed the same way a local variable would be accessed.

This arrangement simplifies code generation for accesses to formal parameters.

The call by value mechanism is efficient for formal parameters that are scalar variables; however, the copy operation would incur substantial overhead for parameters that are arrays It is commonly used for built-in functions of a language because they do not need to produce side effects on parameters and typically have scalar formal parameters parameters

Call by value-result

Call by value-result extends the capabilities of the call by value mechanism by copy ing back the values of formal parameters into corresponding actual parameters while returning control to the calling program.

Thus, if statements in the function modify values of formal parameters, side effects would be realized at return.

This mechanism inherits the simplicity of the call by value mechanism.

Call by reference

Call by reference passes the address of an actual parameter to the called function.

If the actual parameter is an expression in the calling program, the compiler generates code that would evaluate the expression, store its value in a temporary location, and pass the address of the temporary location to the called function.

If a parameter is an array element, the compiler generates code to compute its address and pass it in the parameter list.

The parameter list is thus a list of addresses of actual parameters. To compile a reference to a formal parameter p in a statement in the function, the compiler generates code that would obtain the address of the corresponding actual parameter from

Call by name

Call by name has the effect as if every occurrence of a formal parameter in the body of the called function was substituted by the name of the corresponding actual parameter through string replacement.

Call by name achieves instantaneous side effects.

It also has the implication that the actual parameter that corresponds to a formal parameter may change during the execution of a function, thus providing additional expressive power.

z := d[i]; statement 1

i := i+1; statement 2

x := d[i]+5; statement 3

name substitution implies replacement of the string a by the string d[i] in the body of alpha during the exe- cution of alpha. Thus the call alpha (d[1],x) has the same effect as execution of the statements

When the value of i is modified, the name d[i] of the actual parameter effectively changes. Hence the formal parameter a corresponds to two different elements of d in the first and third statements above.

Call by name is implemented as follows: While compiling a function call, the compiler builds a parameter descriptor for each parameter that is the address of a routine that would compute the address of the corresponding actual parameter.

The routine itself would also be generated at this time

- **pure & Impure Interpreter-**

There are 2 types of interpreter.i)pure ii)impure

- Pure interpreter can be defined as an interpreter that maintains a source program in it's source form till it completely gets interpreted

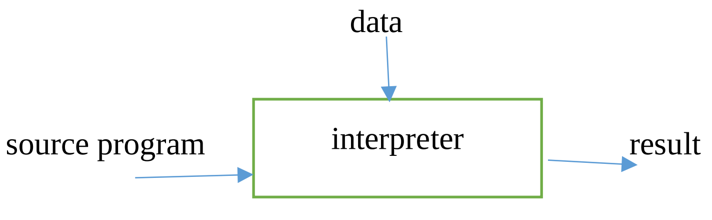
An impure Interpreter define is defined as on interpicked. that does Source program's preliminary processing throughout interpretation to minimize cost of analysis

In the process of pure interpreter the original program of Source Code has to be maintain in the memory. This Causes overhead of analysis when statement is interpreted (line by line evaluate)

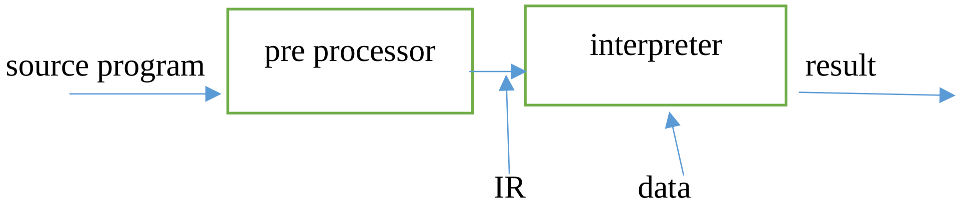
-for-demoing overhead (time, memory, bandwidth.) overhead means time is more required for pure interpreter.

- for decreasing overhead of analysis in the process of interpretation, on impure interpreter does few processing related to Source program.

-The program is converted in Intermediate Representation CIR) during the process of interpretation this is used for Speeding up the interprctation process as a rode componend But since the entire program has to be preprocessed after any modification, it adds 4 fixed overhead at the Star of interpretation



pure interpreter



impure interpreter