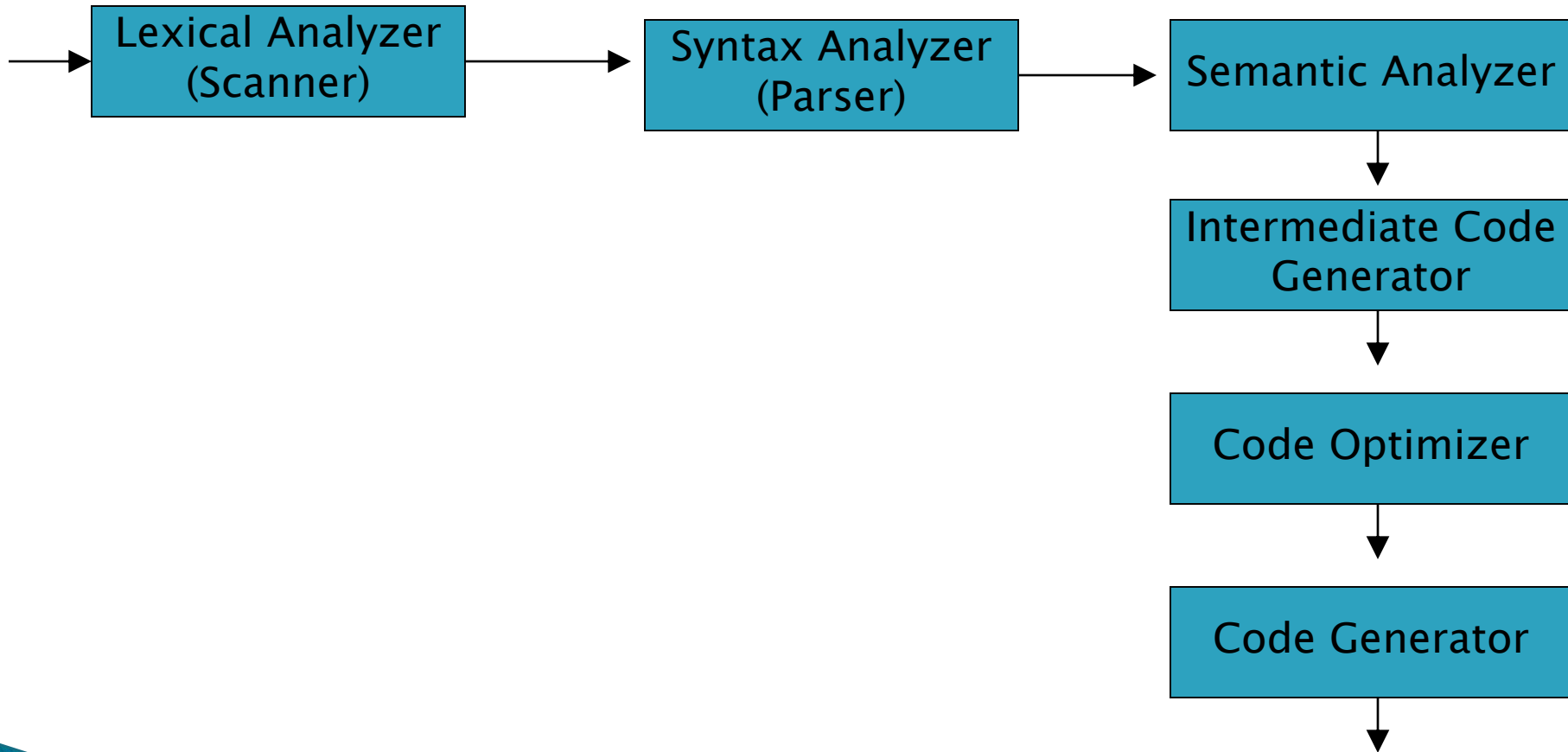


Code Optimization



Code Optimization

- ▶ **Optimization** is a program transformation technique, which tries to improve the **code** by making it consume **less resources (i.e. CPU, Memory)** and deliver high speed.
- ▶ In **optimization**, high-level general programming constructs are replaced by very efficient low-level programming **codes**.

Two goals:

- ▶ **Time Reduction (T)** – reduce the time for the execution of the compiled program
- ▶ **Space reduction (S)** – reduce the space occupied by the running program.

Two levels of CO:

- ▶ Machine Independent optimization (MI)
- ▶ Machine Dependent optimization (MD)

Machine Independent optimization (MI):

e.g. (example from FORTON language):

```
Do L1 I = 1, 1000
```

```
    C = 0.0
```

```
L1 A = D (I)*5
```

- ▶ In this program segment, the statement $C = 0.0$ does something which is not related to the loop and can be done only once outside the loop and reduce the execution time.

$C = 0.0$

DO 11 I = 1, 1000

11 A = D (I)*5

- ▶ This operation of adjusting the code for reducing the execution time is independent of the type of CPU used; it is valid for any type of CPU.

```
do {  
  item = 10; value = value + item;  
} while(value<100);
```

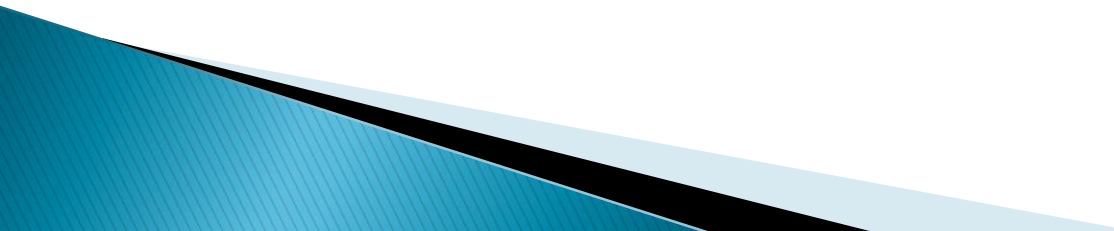
after optimization :

```
Item = 10;  
do {  
  value = value + item;  
} while(value<100)
```


Machine Dependent optimization (MD):

- ▶ Machine-dependent optimization is done after the **target code** has been generated and when the code is transformed according to the target machine architecture.
- ▶ It involves CPU registers and may have absolute memory references rather than relative references.

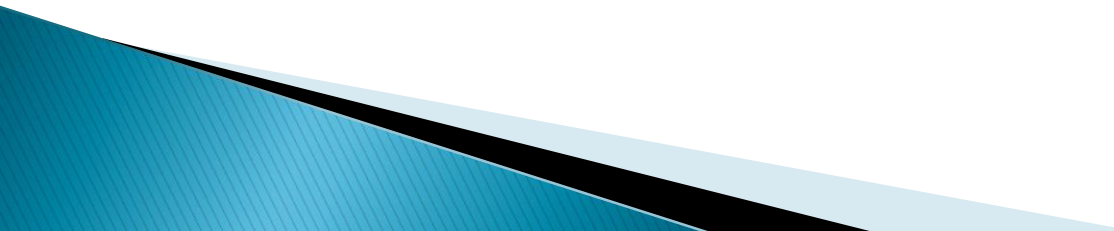
Categories of Code Optimization

- ▶ **Parser optimization** – i.e., optimization actions which are implemented within the parser
 - ▶ **Linear (Peephole) optimization** – those optimization actions which are done at the level of the intermediate code
 - ▶ **Structural optimization** – deals with the total program or a substantial part of it.
- 

Parser Optimization

- ▶ Reduce number of “goto”; (MI, S&T)
- ▶ Use **in-line code** for short functions (e.g. string copy) rather than a function call; (MI, MD, T)
- ▶ Use **shift-left** to multiply by 2 or power of 2; (MI, MD, T) etc.
- ▶ Generally limited to optimization that can be done based on a single production rule in the grammar.

Linear/ Peephole Optimization

- ▶ This kind of optimization is done by one or more additional passes over the entire output of the Parser, in a **linear fashion**. Such optimization is called “peephole” optimization because it inspects and manipulates a small portion of the output of the Parser at a time.
 - ▶ Relatively small and fast
 - ▶ Require only small amount of additional memory
 - ▶ Looks for certain patterns and replaces them with simpler ones
- 

Linear/ Peephole Optimization

Strength Reduction : replace one operation with another requiring less time on any machine (MI)

Operation	Replace by	[opt.]
$x * 4$	$x < < 2$	[T&S]
$x/8$	$x > > 3$	[T&S]
$x\%8$	$x\&7$	[T&S]
multiply by a small int.	Adds or shifts and adds	[T ↓ S ↑]

Constant Folding :

Compiler itself does as much of the calculation as possible. If any expression or part of it contains only constants, then the compiler can easily calculate it at the compile time, even the Parser can do it.

For example,

- ▶ $a + 3 * 7$ is replaced by $a + 21$,

As the parser itself can do the calculations $3*7=21$. This is called Constant Folding.

Constant Propagation :

If the value of a variable is a constant, then replace the variable by the constant

It is not the constant definition, but a variable is assigned to a constant

For example

- $N := 10; C := 2;$

$\text{for } (i:=0; i<N; i++) \{ s := s + i*C; \}$

$\Rightarrow \text{for } (i:=0; i<10; i++) \{ s := s + i*2; \}$

► Dead variable and Dead code :

Dead variables: if a variable is initialized, but then it is not used on any statement (RHS), in any function call, etc., till a certain statement, then between these statements it is a **dead variable**.

- Ineffective statements

- $x := y + 1$ (immediately redefined, eliminate!)
- $y := 5 \quad \Rightarrow \quad y := 5$
- $x := 2 * z \quad \quad \quad x := 2 * z$

Structural Optimization

- ▶ In this type of optimization, the overall structure of the code is analysed. It uses Parse Tree or Parse Matrix.

- ▶ **Elimination of common sub-expression** : [MI, S&T]
- ▶ If the code contains expressions which are repeatedly used, they are calculated only once and each subsequent is replaced by a temporary quantity.
- ▶ For example, $a * b + a * b$

Note that here the sub-expressions was in the same main expression, but a good structural optimizer is able to detect and eliminate sub-expressions used across statements. For example,

```
c = a * b;
```

```
.....
```

```
.....
```

```
d = a * b + a * b;
```

assuming that a, b and c do not change in between.

this will be optimized to:

```
c = a * b
```

```
d = c << 1;
```

- ▶ **Loop unwinding/ Loop unrolling** : [MI,T]
- ▶ Replace a loop by its equivalent linear code.
For example,
 for (i=0; i<5; i++) a[i] = i;
can be replaced by :
 a[0] = 0;
 a[1] = 1; etc..

- Example:

// old loop

```
for(int i=0; i<3; i++) {  
    color_map[n+i] = i;  
}
```

// unrolled version

```
int i = 0;  
colormap[n+i] = i;  
i++;  
colormap[n+i] = i;  
i++;  
colormap[n+i] = i;
```

- ▶ **Replace index with pointers :**

$a[i]$ by $*(a+i)$

$b[i][j]$ by $*(b+j_{\max}*i+j)$

- ▶ which is must faster, because the compiler can take advantage of constant propagation, folding, etc.

- ▶ **Move loop-invariant code out of the loop**
[MI, T]

For example:

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

```
    Array [ i ] = num/denom;
```

- ▶ Replaced by:

```
t1 = num/denom;
```

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

```
    array [ i ] = t1;
```

▶ Loop fusion

- Example

```
for i=1 to N do
  A[i] = B[i] + 1
endfor
for i=1 to N do
  C[i] = A[i] / 2
endfor
for i=1 to N do
  D[i] = 1 / C[i+1]
endfor
```

Before Loop Fusion

```
for i=1 to N do
  A[i] = B[i] + 1
  C[i] = A[i] / 2
  D[i] = 1 / C[i+1]
endfor
```

Is this correct?
Actually, cannot fuse
the third loop

Example :

```
a = 200;
while(a>0)
{
    b = x + y;    // a line loop ni outside avase
    if (a % b == 0)
        printf("%d", a);
}
```

```
int foo(a, b) {  
    a = a - b;  
    b++;  
    a = a * b;  
    return a; }
```

Parser optimization

```
#define foo(a, b) (((a)-(b)) * ((b)+1))
```

```
for (i = 0; i < MAX; i++) /* initialize 2d array to 0's */
    for (j = 0; j < MAX; j++)
        a[i][j] = 0.0;
for (i = 0; i < MAX; i++) /* put 1's along the diagonal */
    a[i][i] = 1.0;
```

loop fusion

New code:

```
for (i = 0; i < MAX; i++)
{
    for (j = 0; j < MAX; j++)
        a[i][j] = 0.0; /* initialize 2d array to 0's */
    a[i][i] = 1.0; /* put 1's along the diagonal */
}
```

Example

Consider the following C code segment.

```
for (i = 0, i < n; i++)  
{  
    for (j = 0; j < n; j++)  
    {  
        if (i % 2)  
        {  
            x += (4 * j + 5 * i);  
            y += (7 + 4 * j);  
        }  
    }  
}
```

- ▶ $4*j$ is common **subexpression elimination**.
- ▶ $5*i$ can be moved out of inner loop. so can be $i\%2$. **loop invariant computation**.
- ▶ $4*j$ as well as $5*i$ can be replaced with $a = -4$; before j loop then $a = a + 4$; where $4*j$ is computed, likewise for $5*i$. **strength reduction**

Constant Folding

Example:

```
int a = 30;
```

```
int b = 10
```

```
int c;
```

```
c = b * 4; // replace b by 10      constant propagation
```

```
if (c > 10) {
```

```
c = c - 10;
```

```
}
```

```
return c * (60 / a); // (60 / a is replaced with 2)
```

constant propagation then constant folding

- ▶ Replacing the expression $4 * 2.14$ by 8.56 is known as _____.
- A. Constant Folding
- B. Constant Propagation
- C. Strength Reduction

Constant Folding



$X = 12.3;$

$Y = X/2.3;$

constant propagation then constant folding


```
c = a * b
```

```
x = a
```

```
--
```

```
--
```

```
-
```

```
d = x * b + 4
```

```
---
```

```
---
```

```
//After Optimization
```

```
c = a * b
```

```
x = a
```

```
till
```

```
d = a * b + 4
```

wrong

```
c=a*b
```

```
d=a*b+4
```

variable optimization

- ▶ `#define k 5`

- ▶ `x = 2 * k`

- ▶ `y = k + 5`

constant propagation, then constant folding
with help of strength optimization

- ▶ This can be computed at compile time and
the values of x and y are :

- ▶ `x = 10`

- ▶ `y = 10`

```
#include <iostream>
using namespace std;
int main() {
    int num;
    num=10;
        cout << "GFG!";
        return 0;
    cout << num; //unreachable code
}
```

//after elimination of unreachable code

```
int main() {
    int num;
    num=10;
        cout << "GFG!";
        return 0;
}
```

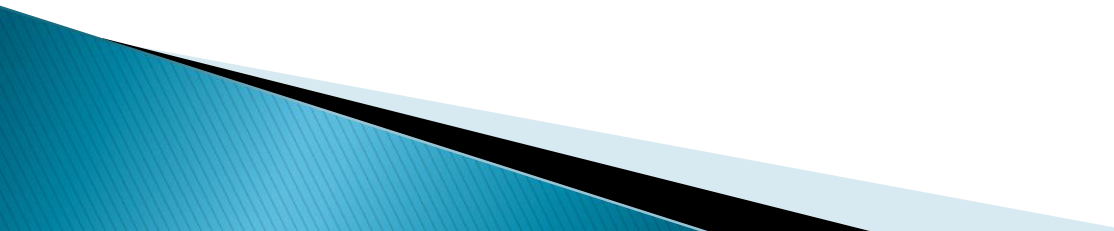
```
for(int k=0;k<10;k++)  
{  
  x = k*2;  
}  
for(int k=0;k<10;k++)  
{  
  y = k+3;  
}
```

```
//After loop jamming  
for(int k=0;k<10;k++)  
{  
  x = k*2;  
  y = k+3;  
}
```

loop fusion

```
#include <stdio.h>
int main(void)
{
    int arr[1000];
    int a = 1, b = 5, c = 25, d = 7;

    for (int i = 0; i < 1000; ++i) {
        arr[i] = (((c % d) * a / b) % d) * i;
    }
    return 0;
}
```



```
#include <stdio.h>
int main(void)
{
    int arr[1000];
    int a = 1, b = 5, c = 25, d = 7;

    // pre calculating the constant expression
    int temp = (((c % d) * a / b) % d);

    for (int i = 0; i < 1000; ++i) {
        arr[i] = temp * i;
    }
    return 0;
}
```

loop invariant code out of loop