CC Lecture 2

Prepared for: 7th Sem, CE, DDU

Prepared by: Niyati J. Buch

Code Optimization

- **Code Optimization** aims at improving the execution efficiency of a program.

- This is achieved in two ways:
  1. **Redundancies** in a program are **eliminated**.
  2. **Computations** in a program are **rearranged** or **rewritten** to make it **execute efficiently**.

- Code optimization must not change the meaning of a program.

Scope of optimization

1. Optimization seeks to improve a program rather than the algorithm used in a program.

   Thus, ***replacement of an algorithm by a more efficient algorithm is beyond the scope of optimization***.

2. Efficient code generation for a specific target machine (e.g. by fully exploiting its instruction set) is also beyond its scope; it belongs in the back end of the compiler.

   The optimization techniques are thus independent of both the PL(programming language) and the target machine.

Two pass schematic for language processing

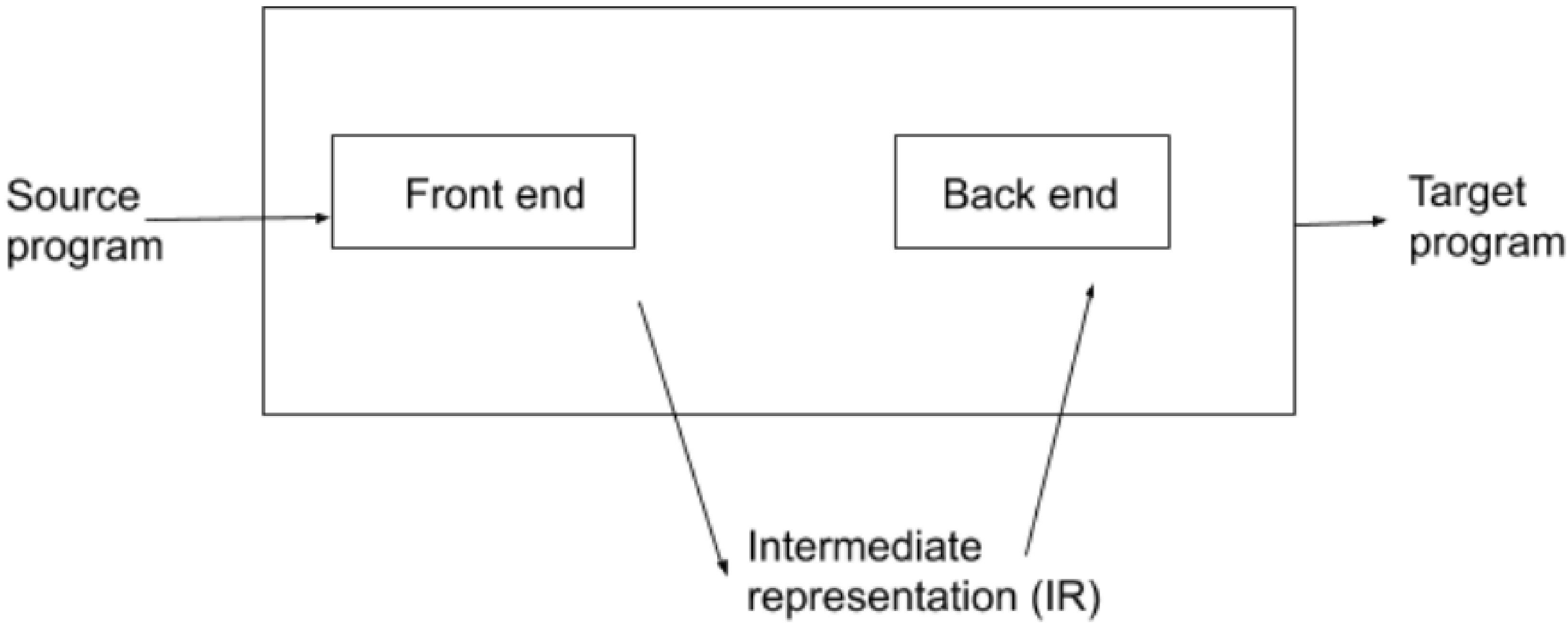


Figure 1 Two pass schematic for language processing

Schematic of an optimizing compiler

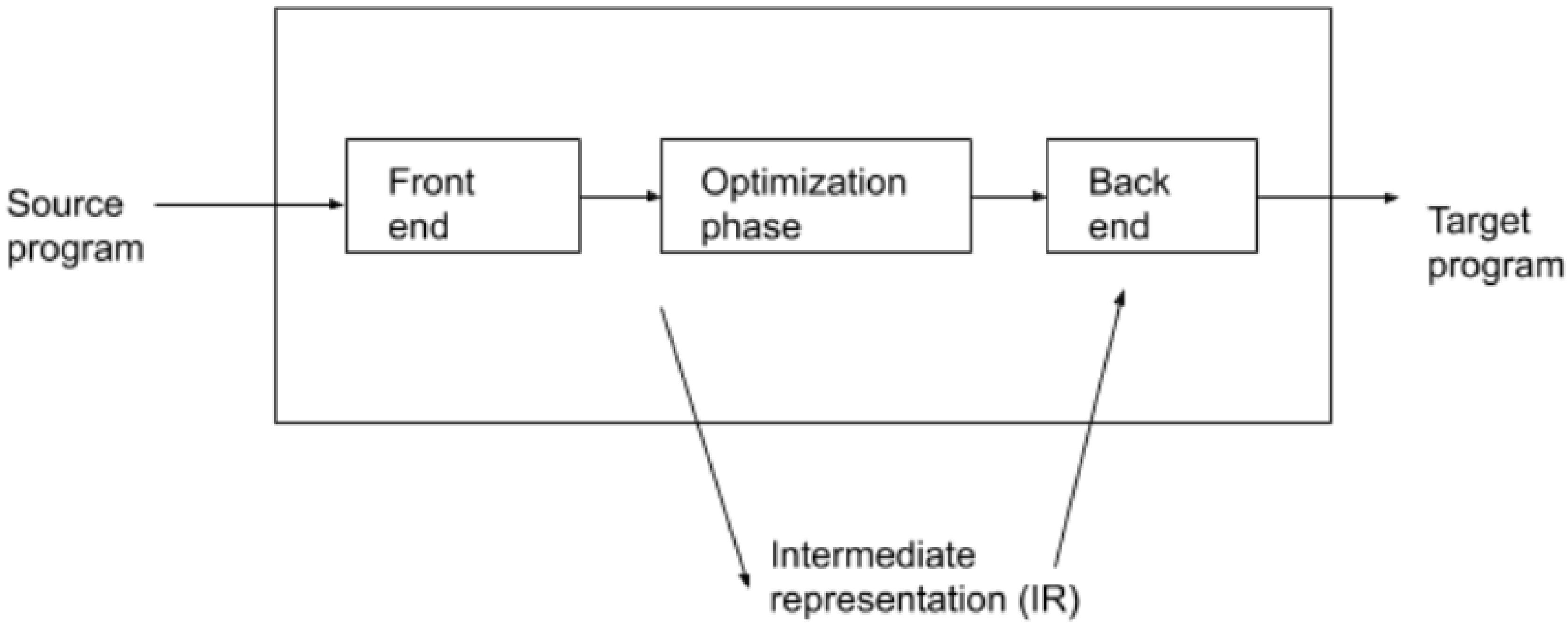


Figure 2 Schematic of an optimizing compiler

Optimizing Transformations

- An ***optimizing transformation*** is a rule for rewriting a segment of a program to improve its execution efficiency without affecting its meaning.

- Optimizing transformations are classified into ***local* and *global*** transformations depending on whether they are applied over small segments of a program consisting of a few source statements or over large segments consisting of loops or function bodies.

- The reason for this distinction is the difference in the costs and benefits of the optimizing transformations.

Few of the optimizing transformations

1. Compile time evaluation
2. Elimination of common subexpression
3. Dead code elimination
4. Frequency reduction
5. Strength reduction

Compile time evaluation

- Execution efficiency can be improved by performing certain actions specified in a program during compilation itself.
- This eliminates the need to perform them during execution of the program, thereby reducing the execution time of the program.

- ***Constant folding*** is the main optimization of this kind.
- When all operands in an operation are constants, the operation can be performed at compilation time.
- The result of the operation, also a constant, can replace the original evaluation in the program.

Compile time evaluation

- Example

  An assignment a:=3.14157 / 2 can be replaced by a:=1.570785, thereby eliminating a division operation.

Elimination of common subexpressions

- **Common subexpressions** are occurrences of expressions yielding the same value.

  (Such expressions are called **equivalent expressions**.)

- Let CS_i designate a set of common subexpressions.
- It is possible to eliminate an occurrence $e_j \in CS_i$ if, no matter how the evaluation of e_j is reached during the execution of the program, the value of some $e_k \in CS_i$ would have been already computed.
- Provision is made to save this value and use it at the place of occurrence of e_j

Elimination of common subexpressions

a := b * c	→	t := b * c
---		a := t
x := b * c + 5.2		---
		x := t + 5.2

- Here CS_i contains two occurrences of b*c.
- The second occurrence of b*c can be eliminated because the first occurrence of b*c is always evaluated before the second occurrence is reached during the execution of the program.
- The value computed at the first occurrence is saved in **t**.
- This value is used in the assignment to x.

Dead code elimination

- The code which can be omitted from a program without affecting its result is called ***dead code***.
- Dead code is detected by checking whether the value assigned in an assignment statement is used anywhere in the program.
-

  An assignment x := <exp> constitutes dead code if the value assigned to x is not used in the program, no matter how control flows after executing this assignment.
- **Note** that <exp> constitute dead code only if its execution does not produce side effects, i.e. only if it does not contain function calls.

Frequency reduction

- Execution time of a program can be reduced by moving code from a part of a program which is executed very frequently to another part of the program which is executed fewer times.

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

Frequency reduction

- Here, **x := 25 * a** is loop invariant. Hence in the optimized program it is computed only once before entering the **for** loop.
- **y := x + z** is not loop invariant. Hence it cannot be subjected to frequency reduction.

for i := 1 to 100 do
begin
  z := i
  x := 25 * a
  y := x+z
End
	→	**x := 25 * a**
		for i := 1 to 100 do
		begin
		  z := i
		  y := x+z
		end

Strength reduction

- The strength reduction optimization replaces the occurrence of a time consuming operation (a **high strength** operation) by an occurrence of a faster operation (a **low strength** operation).
- e.g. replacement of a multiplication by an addition.

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

→

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

Strength reduction

- **Note** that strength reduction optimization is **not** performed on operations involving **floating point operands** because finite precision of floating point arithmetic cannot guarantee equivalence of results after strength reduction.

Practice…

1. r = 20; area = (22/7) * r * r;
2. a = b + c; p = q + r; d = c + r; x = b + c;
3. x = (y – (50/100));
4. ```
   for(i=0; i<n ;i++){
       sum = sum + i;
       a = x *  y;
   }
   ```
5. q = 10; if(q == 0) { p = r + s; }
6. a = b * 4;
   ```

Three Address Code

- In **three-address code**, there is **at most one operator** on the right side of an instruction; that is, no built-up arithmetic expressions are permitted.
- Thus, a source-language expression like **x + y * z** might be translated into the sequence of three-address instructions

  t1 = y * z

  t2 = x + t1

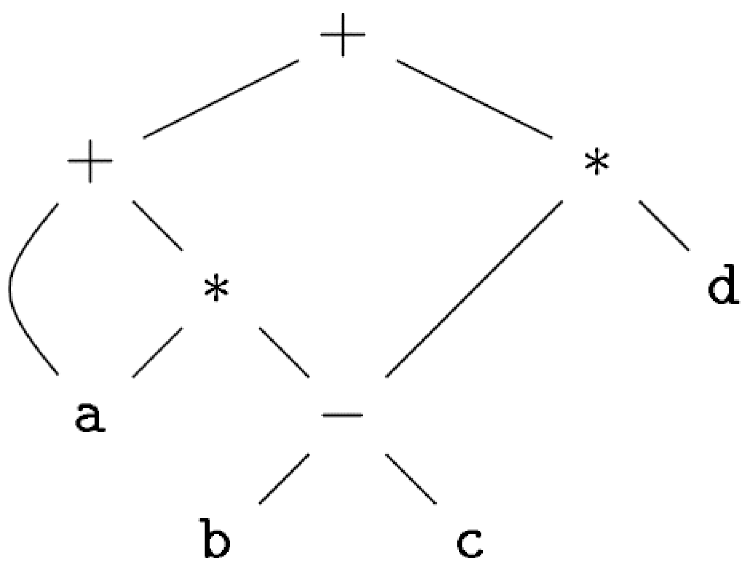
  where t1 and t2 are compiler-generated temporary names.

Benefits

- Desirable for target-code generation and optimization
  - multi-operator arithmetic expressions and of nested flow-of-control statements are unravelled
- The code can be rearranged easily
  - use of names for the intermediate values computed by a program

Three-address code is a linearized representation of a syntax tree or a DAG in which explicit names correspond to the interior nodes of the graph.

DAG (Directed Acyclic Graph)



Three address code

t1 = b - c

t2 = a * t1

t3 = a + t2

t4 = t1 * d

t5 = t3 + t4

Three address code = Address + Instructions

Address in Three address code

1. **A name.**
   - For convenience, we allow source-program names to appear as addresses in three-address code.
   - In an implementation, a source name is replaced by a pointer to its symbol-table entry, where all information about the name is kept.
2. **A constant.**
   - In practice, a compiler must deal with many different types of constants and variables.
   - Type conversions within expressions can be both implicit as well as explicit.

Address in Three address code

3. **A compiler-generated temporary.**
   - It is useful, especially in optimizing compilers, to create a distinct name each time a temporary is needed.
   - These temporaries can be combined, if possible, when registers are allocated to variables.

Symbolic labels

- Symbolic labels will be used by instructions that alter the flow of control.
- A **symbolic label** represents the index of a three-address instruction in the sequence of instructions.
- Actual indexes can be substituted for the labels, either by making a separate pass or by "backpatching".

Three-address instruction forms

1. **Assignment instructions** of the form **x = y op z**,
   – where op is a binary arithmetic or logical operation,
   – and x, y, and z are addresses.
2. **Assignments** of the form **x = op y**,
   – where op is a unary operation.
   – Essential unary operations include unary minus, logical negation, and conversion operators that, for example, convert an integer to a floating-point number.
3. **Copy instructions** of the form **x = y**,
   – where x is assigned the value of y.
4. **An unconditional jump goto L**.
   – The three-address instruction with label L is the next to be executed.

Three-address instruction forms

5. **Conditional jumps** of the form **if x goto L** and **ifFalse x goto L**.
   - These instructions execute the instruction with label L next if x is true and false, respectively.
   - Otherwise, the following three-address instruction in sequence is executed next, as usual.
6. **Conditional jumps** such as **if x relop y goto L**,
   - which apply a **relational operator** (<, ==, >=, etc.) to x and y, and execute the instruction with label L next if x stands in relation relop to y.
   - If not, the three-address instruction following if x relop y goto L is executed next, in sequence.

Three-address instruction forms

7. **Procedure calls and returns** are implemented using the following instructions for procedure and function call
   - **param x for parameters**
   - **call p, n**
   - **y = call p, n**

e.g.

param x1

param x2

...

param xn

call p, n

- generated as part of a call of the procedure p(x1; x2; : : : ; xn).

Three-address instruction forms

- The integer n, indicating the number of actual parameters in **“call p, n,”** is not redundant because calls can be nested.

- That is, some of the first param statements could be parameters of a call that comes after p returns its value; that value becomes another parameter of the later call.

Three-address instruction forms

8. **Indexed copy instructions** of the form **x = y[i]** and **x[i]=y**.
   - The instruction x = y[i] sets x to the value in the location i memory units beyond location y.
   - The instruction x[i]=y sets the contents of the location i units beyond x to the value of y.

Three-address instruction forms

9. **Address and pointer assignments** of the form **x = &y**, **x = *y**, and *** x = y**.
   - The instruction x = &y sets the r-value of x to be the location (l-value) of y. Presumably y is a name, perhaps a temporary, that denotes an expression with an l-value such as A[i][j], and x is a pointer name or temporary.
   - In the instruction x = * y, presumably y is a pointer or a temporary whose r-value is a location. The r-value of x is made equal to the contents of that location.
   - Finally, * x = y sets the r-value of the object pointed to by x to the r-value of y.

```
void quicksort(int m, int n){
/* recursively sorts a[m] through a[n] */
int i, j; int v, x;
if (n <= m) return;
    /* fragment begins here */
    i = m-1; j = n; v = a[n];
    while (1) {
        do i = i+1; while (a[i] < v);
        do j = j-1; while (a[j] > v);
        if (i >= j) break;
            x = a[i]; a[i] = a[j]; a[j] = x; /* swap a[i], a[j] */
    }
    x = a[i]; a[i] = a[n]; a[n] = x; /* swap a[i], a[n] */
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1,n);
}
```

Generating Three address code for given code fragment

- **i = m-1;**
- **j = n;**
- **v = a[n];**

(1) i = m-1

(2) j = n

(3) t1 = 4 * n

(4) v = a[t1]

Here, it is assumed that int occupies 4 bytes.

Generating Three address code for given code fragment

```
do i = i + 1;
   while (a[i] < v);
```

```
(5) i = i + 1
(6) t2 = 4 * i
(7) t3 = a[t2]
(8) if t3<v goto (5)
```

```
do j = j - 1;
   while (a[j] > v);
```

```
(9) j = j - 1
(10) t4 = 4 * j
(11) t5 = a[t4]
(12) if t5>v goto (9)
```

Generating Three address code for given code fragment

if (i >= j) break;	(13) if i>=j goto **(23)**
x = a[i];	(14) t6 = 4 * i (15) x = a[t6]
a[i] = a[j];	(16) t7 = 4 * i (17) t8 = 4 * j (18) t9 = a[t8] (19) a[t7] = t9
a[j] = x; **/* swap a[i], a[j] */**	(20) t10 = 4 * j (21) a[t10] = x
} //closing of while(1)	(22) goto (5)

Generating Three address code for given code fragment

x = a[i];	(23) t11 = 4*i (24) x = a[t11]
a[i] = a[n];	(25) t12 = 4*i (26) t13 = 4*n (27) t14 = a[t13] (28) a[t12] = t14
a[n] = x; **/* swap a[i], a[n] */**	(29) t15 = 4*n (30) a[t15] = x

Three Address Code

```
(1) i = m-1
(2) j = n
(3) t1 = 4*n
(4) v = a[t1]
(5) i = i+1
(6) t2 = 4*i
(7) t3 = a[t2]
(8) if t3<v goto (5)
(9) j = j-1
(10) t4 = 4*j
(11) t5 = a[t4]
(12) if t5>v goto (9)
(13) if i>=j goto (23)
(14) t6 = 4*i
(15) x = a[t6]
(16) t7 = 4*i
(17) t8 = 4*j
(18) t9 = a[t8]
(19) a[t7] = t9
(20) t10 = 4*j
(21) a[t10] = x
(22) goto (5)
(23) t11 = 4*i
(24) x = a[t11]
(25) t12 = 4*i
(26) t13 = 4*n
(27) t14 = a[t13]
(28) a[t12] = t14
(29) t15 = 4*n
(30) a[t15] = x
```