CT3 Compiler design updated 230502 174457

Compiler Design (SRM Institute of Science and Technology)

CT3 Compiler design:

1.What is three address code? What is its type? How is it implemented?

Three address code is a type of intermediate code which is easy to generate and can be easily converted to machine code. It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by compiler. The compiler decides the order of operation given by three address code.

Three address code is used in compiler applications:

Optimization: Three address code is often used as an intermediate representation of code during optimization phases of the compilation process. The three address code allows the compiler to analyze the code and perform optimizations that can improve the performance of the generated code.

Code generation: Three address code can also be used as an intermediate representation of code during the code generation phase of the compilation process. The three address code allows the compiler to generate code that is specific to the target platform, while also ensuring that the generated code is correct and efficient.

Debugging: Three address code can be helpful in debugging the code generated by the compiler. Since three address code is a low-level language, it is often easier to read and understand than the final generated code. Developers can use the three address code to trace the execution of the program and identify errors or issues that may be present.

Example-2: Write three address code for following code

```
for(i = 1; i<=10; i++){
  a[i] = x * 5;
 }
```

$i = 1$

$L : t_1 = x * 5$

$t_2 = \&a$

$t_3 = sizeof(int)$

$t_4 = t_3 * i$

$t_5 = t_2 + t_4$

$*t_5 = t_1$

$i = i + 1$

$if\ i<=10\ goto\ L$

Implementation of Three Address Code –

There are 3 representations of three address code namely

1. Quadruple
2. Triples
3. Indirect Triples

1. Quadruple – It is a structure which consists of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

Advantage –

- Easy to rearrange code for global optimization.
- One can quickly access value of temporary variables using symbol table.

Disadvantage –

- Contain lot of temporaries.
- Temporary variable creation increases time and space complexity.
-

2. Triples – This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So, it consist of only three fields namely op, arg1 and arg2.

Disadvantage –
1.Temporaries are implicit and difficult to rearrange code.
2.It is difficult to optimize because optimization involves moving intermediate code. When a triple is moved, any other triple referring to it must be updated also. With help of pointer one can directly access symbol table entry.

4. **Indirect Triples –** This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

2.Explain in detail about the issues in code generation with examples

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 following issue arises during the code generation phase:

Design Issues

In the code generation phase, various issues can arises:

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

6. Evaluation order

1. Input to the code generator

- 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 can be:

a) **Assembly language:** It allows subprogram to be separately compiled.

b) **Relocatable machine language:** It makes the process of code generation easier.

c) **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 have to be mapped to actual p addresses and levels have to 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.

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.

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.

3.What is back patching?

Backpatching is basically a process of fulfilling unspecified information. This information is of labels. It basically uses the appropriate semantic actions during the process of code generation. It may indicate the address of the Label in goto statements while producing TACs for the given expressions. Here basically two passes are used because assigning the positions of these label statements in one pass is quite challenging. It can leave these addresses unidentified in the first pass and then populate them in the second round.

Backpatching is the process of filling up gaps in incomplete transformations and information.

Need for Backpatching:

Backpatching is mainly used for two purposes:

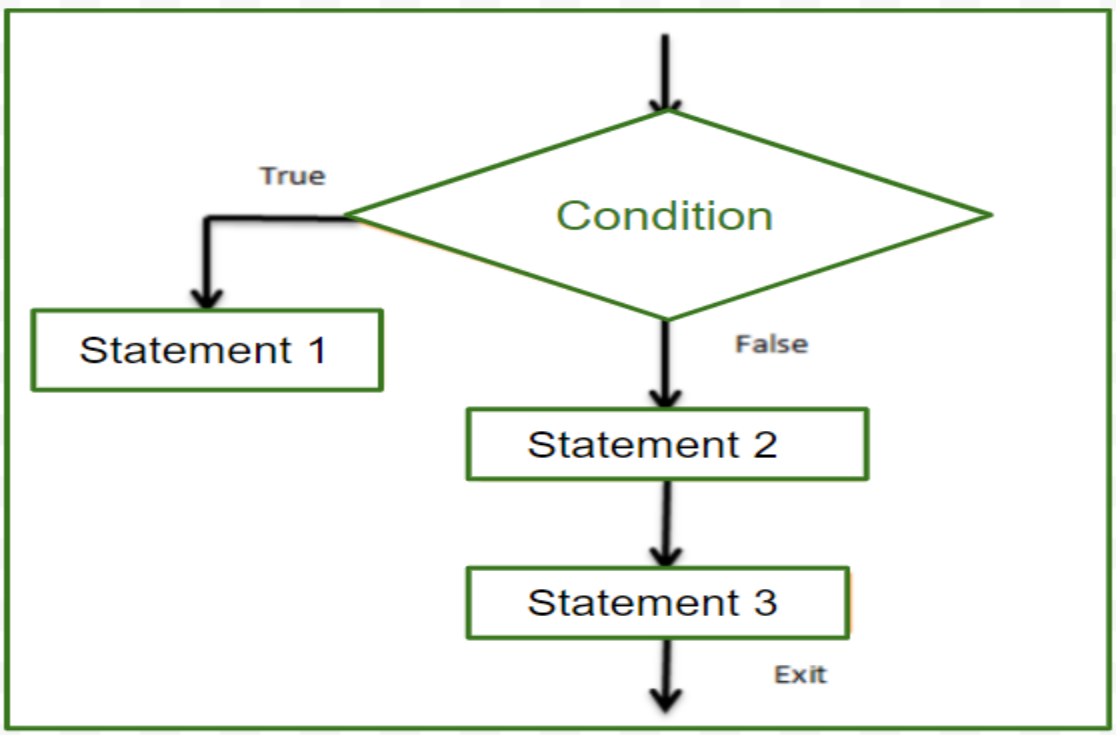
1. Boolean expression:

Boolean expressions are statements whose results can be either true or false. A boolean expression which is named for mathematician George Boole is an expression that evaluates to either true or false. Let's look at some common language examples:

- My favorite color is blue. → true
- I am afraid of mathematics. → false
- 2 is greater than 5. → false

2. Flow of control statements:

The flow of control statements needs to be controlled during the execution of statements in a program. For example:



3. Labels and Gotos:

The most elementary programming language construct for changing the flow of control in a program is a label and ***goto***. When a compiler encounters a statement like ***goto L***, it must check that there is exactly one statement with label ***L*** in the scope of this ***goto*** statement. If the label has already appeared, then the symbol table will have an entry giving the compiler-generated label for the first three-address instruction associated with the source statement labeled ***L***

Applications of Backpatching:

- Backpatching is used to translate flow-of-control statements in one pass itself.
- Backpatching is used for producing quadruples for boolean expressions during bottom-up parsing.
- It is the activity of filling up unspecified information of labels during the code generation process.
- It helps to resolve forward branches that have been planted in the code

The flow of Control Statements:

Control statements are those that alter the order in which statements are executed. If, If-else, Switch-Case, and while-do statements are examples. Boolean expressions are often used in computer languages to

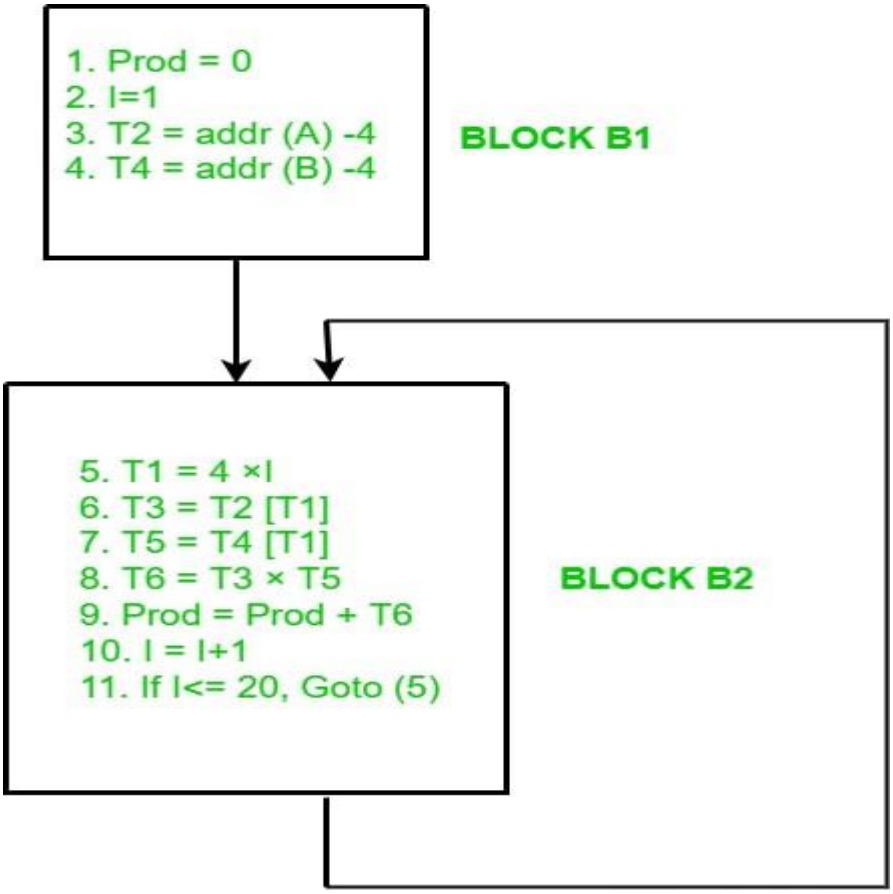
- **Alter the flow of control:** Boolean expressions are conditional expressions that change the flow of control in a statement. The value of such a Boolean statement is implicit in the program's position. For example, if (A) B, the expression A must be true if statement B is reached.
- **Compute logical values:** During bottom-up parsing, it may generate code for Boolean statements via a translation mechanism. A non-terminal marker M in the grammar establishes a semantic action that takes the index of the following instruction to be formed at the appropriate moment.

Flow Graph in Code Generation

Flow Graph:

A flow graph is simply a directed graph. For the set of basic blocks, a flow graph shows the flow of control information. A control flow graph is used to depict how the program control is being parsed among the blocks. A flow graph is used to illustrate the flow of control between basic blocks once an intermediate code has been partitioned into basic blocks. When the beginning instruction of the Y block follows the last instruction of the X block, an edge might flow from one block X to another block Y.

Let's make the flow graph of the example that we used for basic block formation:



Basic block structure

Optimization is applied to the basic blocks after the intermediate code generation phase of the compiler. Optimization is the process of transforming a program that improves the code by consuming fewer resources and delivering high speed. In optimization, high-level codes are replaced by their equivalent efficient low-level codes. Optimization of basic blocks can be machine-dependent or machine-independent. These transformations are useful for improving the quality of code that will be ultimately generated from basic block.

There are two types of basic block optimizations:

1. Structure preserving transformations
2. Algebraic transformations

1. Structure-Preserving Transformations

Structure preserving transformations can be achieved by the following methods:

1. Common sub-expression elimination
2. Dead code elimination
3. Renaming of temporary variables
4. Interchange of two independent adjacent statements

2. Algebraic Transformations

In the case of algebraic transformation, we basically change the set of expressions into an algebraically equivalent set.

For example, and expression

```
x:= x + 0
```

```
or x:= x *1
```

This can be eliminated from a basic block without changing the set of expressions.

How would you solve the issues in the design of code generators?

1. Use appropriate algorithms: The selection of appropriate algorithms for different code generation tasks, such as instruction selection and register allocation, can significantly affect the quality of generated code. For example, graph-coloring-based register allocation algorithms can produce high-quality results but may have high computational complexity. Therefore, it is essential to choose algorithms that balance performance and quality.
2. Handle complex language constructs: Many programming languages have complex constructs, such as closures, exception handling, and dynamic dispatch, that are challenging to translate efficiently into machine code. To address this issue, code generators can use specialized techniques, such as trampolines, to handle these constructs efficiently.
3. Optimize generated code: The generated code can be optimized for performance by using techniques such as dead-code elimination, loop unrolling, and common subexpression elimination. These optimizations can significantly improve the performance of generated code, but they may also increase the complexity of the code generator.
4. Use machine learning: Machine learning techniques can be used to train models that can generate high-quality code automatically. For example, neural networks can learn to generate code by observing input-output pairs, which can reduce the complexity of the code generator and improve the quality of generated code.
5. Use formal methods: Formal methods can be used to prove the correctness of the code generator and ensure that the generated code meets the specification. Formal methods can also help to identify potential bugs and improve the quality of generated code.

Data flow analysis in Compiler

Data flow analysis is a technique used in compilers and other software analysis tools to derive information about how data is being used and propagated through a program. It is used to analyze the control flow of a program and identify variables that are used or defined at various points in the program. Data flow

analysis is essential for many compiler optimizations, such as dead code elimination, common subexpression elimination, and constant propagation.

Some of the common types of data flow analysis performed by compilers include:

1. Reaching Definitions Analysis: This analysis tracks the definition of a variable or expression and determines the points in the program where the definition “reaches” a particular use of the variable or expression. This information can be used to identify variables that can be safely optimized or eliminated.

2. Live Variable Analysis: This analysis determines the points in the program where a variable or expression is “live”, meaning that its value is still needed for some future computation. This information can be used to identify variables that can be safely removed or optimized.

3. Available Expressions Analysis: This analysis determines the points in the program where a particular expression is “available”, meaning that its value has already been computed and can be reused. This information can be used to identify opportunities for common subexpression elimination and other optimization techniques.

4. Constant Propagation Analysis: This analysis tracks the values of constants and determines the points in the program where a particular constant value is used. This information can be used to identify opportunities for constant folding and other optimization techniques.

Data flow analysis can have a number of advantages in compiler design, including:

1. Improved code quality: By identifying opportunities for optimization and eliminating potential errors, data flow analysis can help improve the quality and efficiency of the compiled code.
2. Better error detection: By tracking the flow of data through the program, data flow analysis can help identify potential errors and bugs that might otherwise go unnoticed.
3. Increased understanding of program behavior: By modeling the program as a graph and tracking the flow of data, data flow analysis can

help programmers better understand how the program works and how it can be improved.

7.Describe in detail about the stack allocation in memory management?

Stack allocation is a popular technique used for managing memory in computer programs. It is a type of automatic memory management where variables are allocated and deallocated from a stack data structure.

In the stack allocation technique, the stack is a region of memory that is used to store local variables and function call frames. When a function is called, a new frame is pushed onto the stack, which contains the local variables for that function. When the function returns, its frame is popped from the stack, and the local variables are deallocated.

The stack is managed by the operating system, which ensures that it has sufficient space to store all the function call frames and local variables. The size of the stack is usually fixed at program startup and cannot be dynamically resized during runtime.

The advantages of using stack allocation include:

1. Efficiency: Stack allocation is a very fast memory allocation technique since all that is required is to adjust the stack pointer to allocate or deallocate a variable.
2. Simplicity: Stack allocation is a simple and straightforward technique that does not require complex memory management algorithms.
3. Safety: Stack allocation is a safe memory allocation technique since it ensures that memory is automatically deallocated when it is no longer needed.

However, there are also some limitations to using stack allocation:

1. Limited memory: Since the size of the stack is fixed at program startup, there is a limited amount of memory available for stack allocation.
2. Stack overflow: If too much memory is allocated on the stack, it can cause a stack overflow error, which can crash the program.
3. Lifetime restrictions: Variables allocated on the stack have a limited lifetime and cannot be used outside of their scope.

What are different storage allocation strategies? Explain.

Storage Allocation

The different ways to allocate memory are:

1. Static storage allocation
2. Stack storage allocation
3. Heap storage allocation

Static storage allocation

- In static allocation, names are bound to storage locations.
- If memory is created at compile time then the memory will be created in static area and only once.
- Static allocation supports the dynamic data structure that means memory is created only at compile time and deallocated after program completion.
- The drawback with static storage allocation is that the size and position of data objects should be known at compile time.
- Another drawback is restriction of the recursion procedure.

Stack Storage Allocation

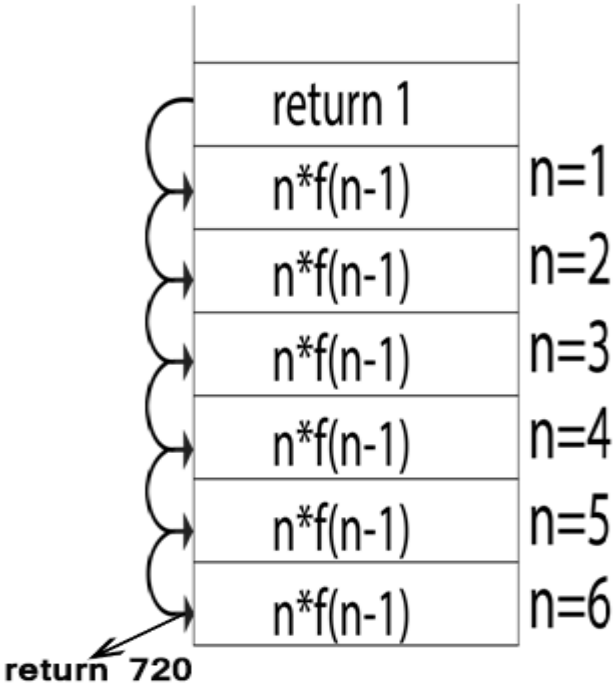
- In static storage allocation, storage is organized as a stack.
- An activation record is pushed into the stack when activation begins and it is popped when the activation end.
- Activation record contains the locals so that they are bound to fresh storage in each activation record. The value of locals is deleted when the activation ends.
- It works on the basis of last-in-first-out (LIFO) and this allocation supports the recursion process.

Heap Storage Allocation

- Heap allocation is the most flexible allocation scheme.
- Allocation and deallocation of memory can be done at any time and at any place depending upon the user's requirement.
- Heap allocation is used to allocate memory to the variables dynamically and when the variables are no more used then claim it back.
- Heap storage allocation supports the recursion process.

Example:

```
1. fact (int n)
2. {
3.    if (n<=1)
4.       return 1;
5.    else
6.       return (n * fact(n-1));
7. }
8. fact (6)
```



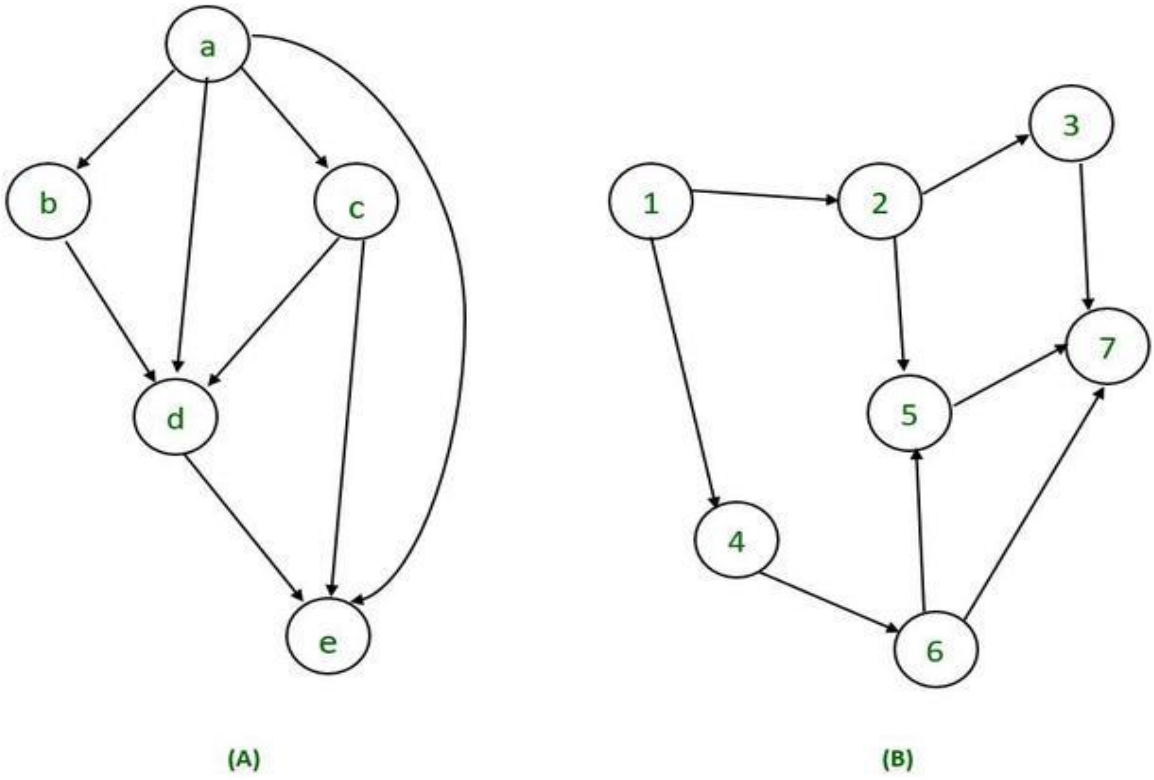
Directed Acyclic graph in Compiler Design (with examples)

Directed Acyclic Graph :

The Directed Acyclic Graph (DAG) is used to represent the structure of basic blocks, to visualize the flow of values between basic blocks, and to provide optimization techniques in the basic block. To apply an optimization technique to a basic block, a DAG is a three-address code that is generated as the result of an intermediate code generation.

- Directed acyclic graphs are a type of data structure and they are used to apply transformations to basic blocks.
- The Directed Acyclic Graph (DAG) facilitates the transformation of basic blocks.
- DAG is an efficient method for identifying common sub-expressions.
- It demonstrates how the statement's computed value is used in subsequent statements.

Examples of directed acyclic graph :



(A) (B)

Directed Acyclic Graph Characteristics :

- The graph's leaves each have a unique identifier, which can be variable names or constants.
- The interior nodes of the graph are labelled with an operator symbol.
- In addition, nodes are given a string of identifiers to use as labels for storing the computed value.
- Directed Acyclic Graphs have their own definitions for transitive closure and transitive reduction.
- Directed Acyclic Graphs have topological orderings defined.

Application of Directed Acyclic Graph:

- Directed acyclic graph determines the subexpressions that are commonly used.
- Directed acyclic graph determines the names used within the block as well as the names computed outside the block.
- Determines which statements in the block may have their computed value outside the block.
- Code can be represented by a Directed acyclic graph that describes the inputs and outputs of each of the arithmetic operations performed within the code; this representation allows the compiler to perform common subexpression elimination efficiently.
- Several programming languages describe value systems that are linked together by a directed acyclic graph. When one value changes, its successors are recalculated; each value in the DAG is evaluated as a function of its predecessors.

Peephole Optimization in Compiler Design:

Peephole optimization is a type of code Optimization performed on a small part of the code. It is performed on a very small set of instructions in a segment of code.
The small set of instructions or small part of code on which peephole optimization is performed is known as ***peephole*** *or* ***window****.*
It basically works on the theory of replacement in which a part of code is replaced by shorter and faster code without a change in output. The peephole is machine-dependent optimization.

Objectives of Peephole Optimization:

The objective of peephole optimization is as follows:

1. To improve performance
2. To reduce memory footprint
3. To reduce code size

Peephole Optimization Techniques:

A. Redundant load and store elimination: In this technique, redundancy is eliminated.

Initial code:

```
y = x + 5;
i = y;
z = i;
w = z * 3;
```

Optimized code:

```
y = x + 5;
w = i * 3;

//* We've removed two redundant variables i & z whose value were just being copied from one another.
```

B. Constant folding: The code that can be simplified by the user itself, is simplified.

Initial code:

```
x = 2 * 3;
```

Optimized code:

```
x = 6;
```

C. Strength Reduction: The operators that consume higher execution time are replaced by the operators consuming less execution time.

Initial code:

```
y = x * 2;
```

Optimized code:

```
y = x + x;    or     y = x << 1;
```

Initial code:

```
y = x / 2;
```

Optimized code:

```
y = x >> 1;
```

D. Null sequences/ Simplify Algebraic Expressions : Useless operations are deleted.

```
a := a + 0;

a := a * 1;

a := a/1;

a := a - 0;
```

E. Combine operations: Several operations are replaced by a single equivalent operation.

F. Deadcode Elimination: A part of the code which can never be executed, eliminating it will improve processing time and reduces set of instruction.

Code Optimization in Compiler Design

The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (i.e. CPU, Memory) so that faster-running machine code will result. Compiler optimizing process should meet the following objectives :

- The optimization must be correct, it must not, in any way, change the meaning of the program.
- Optimization should increase the speed and performance of the program.
- The compilation time must be kept reasonable.
- The optimization process should not delay the overall compiling process.

Types of Code Optimization: The optimization process can be broadly classified into two types :

1. **Machine Independent Optimization:** This code optimization phase attempts to improve the **intermediate code** to get a better target code as the output. The part of the intermediate code which is transformed here does not involve any CPU registers or absolute memory locations.
2. **Machine Dependent Optimization:** 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. Machine-dependent optimizers put efforts to take maximum **advantage** of the memory hierarchy.

Code Optimization is done in the following different ways:

1. Compile Time Evaluation:

- C

```
(i)   A = 2*(22.0/7.0)*r
      Perform 2*(22.0/7.0)*r at compile time.
(ii)  x = 12.4
      y = x/2.3
      Evaluate x/2.3 as 12.4/2.3 at compile time.
```

2. Variable Propagation:

- C

```
//Before Optimization
c = a * b
x = a
till
d = x * b + 4


//After Optimization
c = a * b
x = a
till
d = a * b + 4
```

3. Constant Propagation:

- If the value of a variable is a constant, then replace the variable with the constant. The variable may not always be a constant.

Example:

- C

```
(i)  A = 2*(22.0/7.0)*r
     Performs 2*(22.0/7.0)*r at compile time.
(ii)  x = 12.4
      y = x/2.3
      Evaluates x/2.3 as 12.4/2.3 at compile time.
(iii) int k=2;
      if(k) go to L3;
      It is evaluated as :
      go to L3 ( Because k = 2 which implies condition is always true)
```

4. Constant Folding:

- Consider an expression : a = b op c and the values b and c are constants, then the value of a can be computed at compile time.

Example:

- C

```
#define k 5
x = 2 * k
y = k + 5

This can be computed at compile time and the values of x and y are :
 x = 10
 y = 10
```

Note: Difference between Constant Propagation and Constant Folding:

- In Constant Propagation, the variable is substituted with its assigned constant where as in Constant Folding, the variables whose values can be computed at compile time are considered and computed.

5. Copy Propagation:

- It is extension of constant propagation.
- After a is assigned to x, use a to replace x till a is assigned again to another variable or value or expression.

- It helps in reducing the compile time as it reduces copying.

Example :

- C

```
//Before Optimization
c = a * b
x = a
till
d = x * b + 4


//After Optimization
c = a * b
x = a
till
d = a * b + 4
```

6. Common Sub Expression Elimination:

- In the above example, a*b and x*b is a common sub expression.

7. Dead Code Elimination:

- Copy propagation often leads to making assignment statements into dead code.
- A variable is said to be dead if it is never used after its last definition.
- In order to find the dead variables, a data flow analysis should be done.

Example:

- C

```
c = a * b
x = a
till
d = a * b + 4

//After elimination :
```

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

8. Unreachable Code Elimination:

- First, Control Flow Graph should be constructed.
- The block which does not have an incoming edge is an Unreachable code block.
- After constant propagation and constant folding, the unreachable branches can be eliminated.

- C++

```cpp
#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;
 }
```

9. Function Inlining:

- Here, a function call is replaced by the body of the function itself.
- This saves a lot of time in copying all the parameters, storing the return address, etc.

10. Function Cloning:

- Here, specialized codes for a function are created for different calling parameters.
- **Example:** Function Overloading

11. Induction Variable and Strength Reduction:

- An induction variable is used in the loop for the following kind of assignment i = i + constant. It is a kind of Loop Optimization Technique.
- Strength reduction means replacing the high strength operator with a low strength.

Examples:

- C

```
Example 1 :
Multiplication with powers of 2 can be replaced by shift left operator which is less
expensive than multiplication
a=a*16
// Can be modified as :
a = a<<4

Example 2 :
i = 1;
while (i<10)
{
   y = i * 4;
}


//After Reduction
i = 1
t = 4
{
  while( t<40)
  y = t;
  t = t + 4;
}
```

Loop Optimization Techniques:

1. Code Motion or Frequency Reduction:

- The evaluation frequency of expression is reduced.
- The loop invariant statements are brought out of the loop.

Example:

- C

```
a = 200;
 while(a>0)
 {
     b = x + y;
     if (a % b == 0)
     printf(“%d”, a);
   }


//This code can be further optimized as
a = 200;
b = x + y;
while(a>0)
 {
     if (a % b == 0}
     printf(“%d”, a);
   }
```

2. Loop Jamming:

- Two or more loops are combined in a single loop. It helps in reducing the compile time.

Example:

- C

```
// Before loop jamming
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;
}
```

3. Loop Unrolling:

- It helps in optimizing the execution time of the program by reducing the iterations.
- It increases the program's speed by eliminating the loop control and test instructions.

Example:

```
//Before Loop Unrolling

for(int i=0;i<2;i++)
{
  printf("Hello");
}

//After Loop Unrolling

printf("Hello");
printf("Hello");
```

Advantages of Code Optimization:

Improved performance: Code optimization can result in code that executes faster and uses fewer resources, leading to improved performance.
Reduction in code size: Code optimization can help reduce the size of the generated code, making it easier to distribute and deploy.

Increased portability: Code optimization can result in code that is more portable across different platforms, making it easier to target a wider range of hardware and software.
Reduced power consumption: Code optimization can lead to code that consumes less power, making it more energy-efficient.
Improved maintainability: Code optimization can result in code that is easier to understand and maintain, reducing the cost of software maintenance.

Disadvantages of Code Optimization:

Increased compilation time: Code optimization can significantly increase the compilation time, which can be a significant drawback when developing large software systems.
Increased complexity: Code optimization can result in more complex code, making it harder to understand and debug.
Potential for introducing bugs: Code optimization can introduce bugs into the code if not done carefully, leading to unexpected behavior and errors.
Difficulty in assessing the effectiveness: It can be difficult to determine the effectiveness of code optimization, making it hard to justify the time and resources spent on the process.

Discuss in detail about the four principle uses of registers in code generation.

Registers are high-speed storage areas used by a processor to store data and intermediate results during the execution of a program. In code generation, the effective use of registers can significantly improve the performance and efficiency of the generated code. The four principle uses of registers in code generation are:

1. Register allocation: The first and most important use of registers is to store frequently accessed data and intermediate results that are likely to be reused later in the program. The register allocation algorithm assigns registers to variables and expressions to minimize the number of memory accesses needed during program execution. Register allocation is a critical step in optimizing the performance of a program.

2. Argument passing: When a function or method is called, its arguments must be passed to the function. Registers are often used to pass the arguments since they are faster than memory access. The number of registers used for argument passing depends on the architecture of the processor.
3. Return value storage: After a function or method is executed, it returns a value to the calling code. Registers are used to store the return value since they are faster than memory access. Again, the number of registers used for return value storage depends on the architecture of the processor.
4. Instruction operands: Registers are used to hold the operands of instructions. In many cases, the operands are stored in registers to minimize the number of memory accesses needed during program execution. This helps to improve the performance of the program.