Code Generation

What is code generation? Explain basic blocks & flow graphs?

Code Generation:

Code generation is the final phase of compilation where high-level source code is transformed into lower-level object code, typically in assembly language. The generated code should preserve the exact semantics of the source code while being efficient in terms of CPU usage and memory management.

Directed Acyclic Graph (DAG):

A Directed Acyclic Graph (DAG) is a graphical representation that depicts the structure of basic blocks and facilitates optimization. In a DAG:

- Leaf nodes represent identifiers, names, or constants.
- Interior nodes represent operators and the results of expressions or identifiers where values are stored.
- DAGs provide easy transformation of basic blocks and optimize the flow of values among them.

Peephole Optimization:

Peephole optimization works locally on code to transform it into an optimized form. It analyzes a bunch of statements and performs optimizations such as redundant instruction elimination, removing unreachable code, and optimizing the flow of control.

Basic Blocks:

Basic blocks are straight-line code sequences with no inward or outward branches except for entry and exit points, respectively. They comprise statements that execute sequentially without halting. Basic block construction involves partitioning a sequence of three-address instructions. The algorithm identifies leaders within the intermediate code, which are the starting points of basic blocks.

Flow Graph:

A flow graph is a directed graph that illustrates the flow of control among basic blocks. It depicts how program control is parsed between blocks, with edges representing control flow from one block to another. Flow graphs help visualize the execution flow of a program and aid in understanding control dependencies and optimization opportunities.

Transformations on Basic Blocks:

Basic blocks can undergo transformations to optimize code without changing the set of expressions computed. Structure-preserving transformations include common sub-expression elimination, dead code elimination, and renaming of temporary variables. Algebraic transformations involve replacing expressions with algebraically equivalent forms to simplify the code.

Explain three address codes with example

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

1. **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.
2. **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.
3. **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.
4. **Language translation:** Three address code can also be used to translate code from one programming language to another. By translating code to a common intermediate representation, it becomes easier to translate the code to multiple target languages.

General Representation

```
a = b op c
```

Where a, b or c represents operands like names, constants or compiler generated temporaries and op represents the operator

Example-1: Convert the expression a * – (b + c) into three address code.

$t_1 = b + c$

$t_2 = \text{uminus } t_1$

$t_3 = a * t_2$

Example-2: Write three address code for following code

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

$i = 1$

$L : t_1 = x * 5$

$t_2 = \&a$

$t_3 = \text{sizeof(int)}$

$t_4 = t_3 * i$

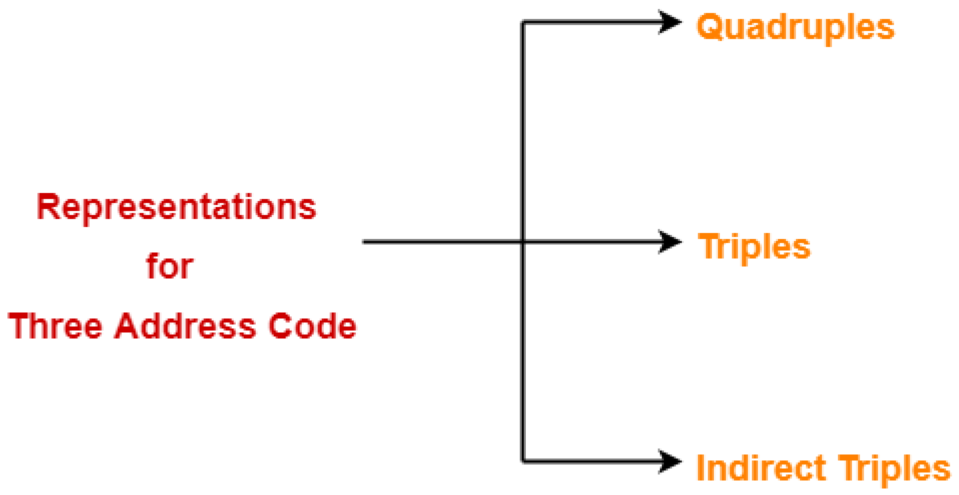
$t_5 = t_2 + t_4$

$*t_5 = t_1$

$i = i + 1$

if $i <= 10$ goto L

Explain quadruples & triples with examples.



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.

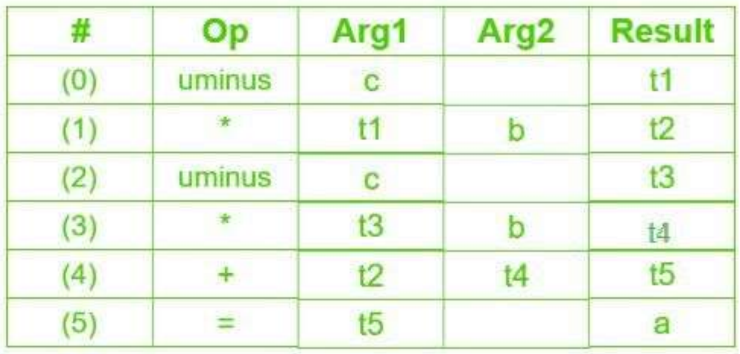
Disadvantage –

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

Example – Consider expression a = b * – c + b * – c. The three address code is:

t1 = uminus c   (Unary minus operation on c)
t2 = b * t1
t3 = uminus c (Another unary minus operation on c)
t4 = b * t3

t5 = t2 + t4
a = t5 (Assignment of t5 to a)



#	Op	Arg1	Arg2	Result
(0)	uminus	c		t1
(1)	*	t1	b	t2
(2)	uminus	c		t3
(3)	*	t3	b	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

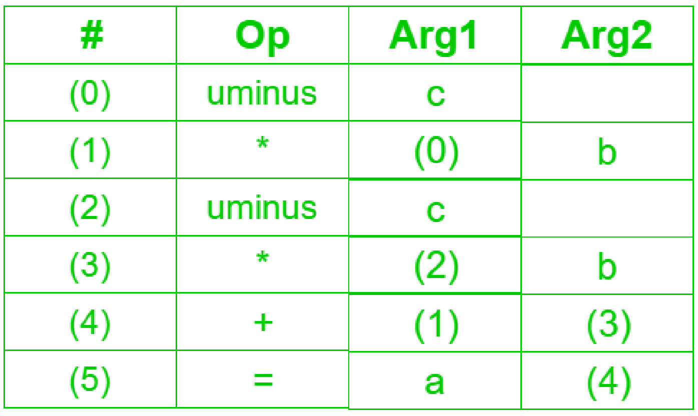
Quadruple representation

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 –

- Temporaries are implicit and difficult to rearrange code.
- 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.

Example – Consider expression a = b * – c + b * – c

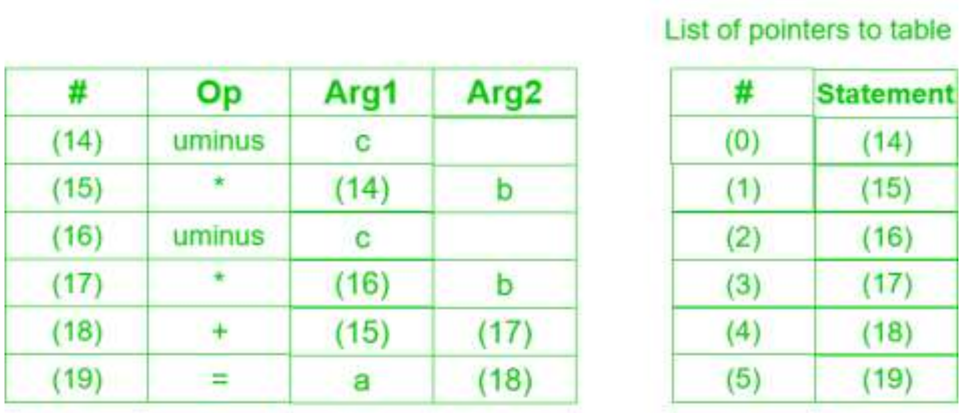


#	Op	Arg1	Arg2
(0)	uminus	c	
(1)	*	(0)	b
(2)	uminus	c	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	a	(4)

Triples representation

3. 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.

Example – Consider expression a = b * – c + b * – c



#	Op	Arg1	Arg2
(14)	uminus	c	
(15)	*	(14)	b
(16)	uminus	c	
(17)	*	(16)	b
(18)	+	(15)	(17)
(19)	=	a	(18)

List of pointers to table

#	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Indirect Triples representation

Explain quadruples & triples 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.

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 determining runtime addresses of data objects.
   - Intermediate codes may vary in representation, such as quadruples, syntax trees, etc.
2. **Target Program:**
   - The output of the code generator can be absolute machine language, relocatable machine language, or assembly language.
   - Each output format has its advantages and drawbacks in terms of execution, linking, and loading.
3. **Memory Management:**
   - Mapping names in the source program to data object addresses is done by both the front end and the code generator.
   - The symbol table assists in determining relative addresses for names.
4. **Instruction Selection:**
   - Efficient instruction selection improves program efficiency.
   - Selection involves considering instruction completeness, uniformity, speeds, and machine idioms.
   - Designing optimal instruction sequences requires knowledge of instruction costs, which can be challenging to predict accurately.
5. **Register Allocation Issues:**
   - Efficient register allocation is crucial for faster computations.

- It involves two subproblems: register allocation and register assignment.
- Register allocation decides which sets of variables reside in registers at each program point.
- Register assignment selects specific registers to access variables.

6. **Evaluation Order:**
    - The code generator determines the order in which instructions will be executed.
    - The order of computations impacts target code efficiency.
    - Finding the best computational order is challenging and often an NP-complete problem.

Approaches to Address Code Generation Issues:

- Code generators must prioritize generating correct, maintainable, testable, and efficient code.
- Design goals include correctness, ease of maintenance, testability, and efficiency.

Disadvantages in Code Generator Design:

1. **Limited Flexibility:**

    Code generators may be designed for specific code types or target platforms, limiting their flexibility to handle various inputs or generate code for different platforms.

2. **Maintenance Overhead:**

    Maintaining and updating code generators alongside generated code can add complexity and potential errors to a project.

3. **Debugging Difficulties:**

    Debugging generated code may be challenging due to readability and understanding issues, potentially making issue identification and resolution harder.

4. **Performance Issues:**

   Code generators may not always produce optimal code, impacting performance, especially in performance-critical applications.

5. **Learning Curve:**

   Code generators typically require a deep understanding of the underlying framework and programming languages, leading to a steep learning curve for developers.

6. **Over-Reliance:**

   Over-reliance on generated code can limit developers' ability to write manual code when necessary, impacting flexibility and creativity.

Explain Run time storage management & next use information

Run-Time Storage Management:

The run-time environment of a program manages memory and maintains essential information required for executing the program. It includes structures such as activation records, program counters, stack pointers, and frame pointers.

Types of Runtime Environments:

1. **Fully Static:**
    - Suitable for languages without pointers or dynamic allocation and no support for recursive function calls.
    - Each procedure has only one activation record allocated before execution.
    - Variables are accessed directly via fixed addresses.
    - Minimal bookkeeping overhead, mainly storing return addresses.
2. **Stack-Based:**
    - Activation records are allocated and deallocated from the stack portion of memory during function calls and returns.
    - Stack grows and shrinks with the chain of function calls.
3. **Fully Dynamic:**
    - Used in functional languages like Lisp, ML, etc.
    - Activation records are deallocated only when all references to them have disappeared.
    - Requires a memory manager (garbage collector) for dynamic allocation and deallocation.

Activation Record:

- A contiguous block of storage managed by the run-time environment.
- Allocated when a procedure is entered and deallocated when exited.

- Contains temporary data, local data, machine status, access links, control links, actual parameters, and returned values.

Next-Use Information:

- Determines whether a variable value in a register will be referenced subsequently.
- If a variable's value computed at statement i is used in statement j without intervening assignments, it's live at statement i.
- To determine liveness and next-use information for each statement in a basic block:
    1. Start from the last statement and scan backward.
    2. Attach current liveness and next-use information of variables.
    3. Set variables to "not live" and "no next use" and update next-use information for referenced variables.
    4. Record next uses of variables into quadruples and mark them as dead if no subsequent use.

Example:

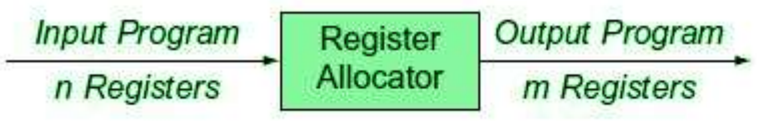
- Quadruple: **x := y op z;**
- Record next uses of x, y, z into quadruple.
- Mark x dead if its previous value has no next use.
- Next use of y is the current statement; next use of z is also the current statement; y and z are live.

Next-use information helps optimize register allocation by reusing registers when variables are no longer needed, improving program efficiency.

Explain issues in register allocation.

Issues in Register Allocation:

1. **Limited Resource:**
    - Registers are the fastest memory locations but limited in number, posing a challenge for allocation.
    - Register allocation is an NP-complete problem, making it computationally intensive.

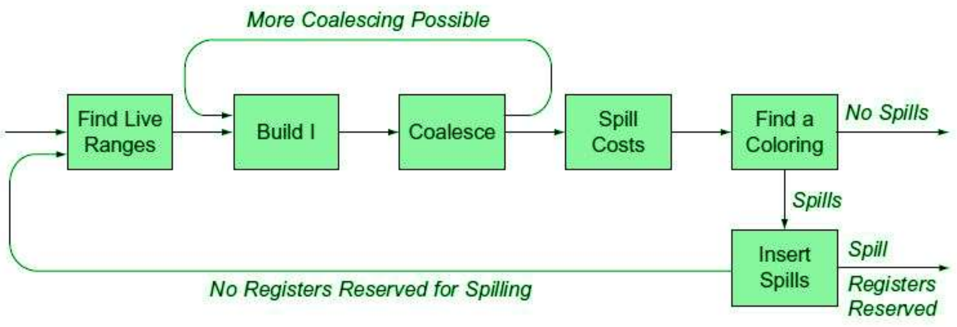


2. **Allocation vs. Assignment:**
    - **Allocation:** Maps an unlimited namespace onto the finite register set of the target machine.
        - **Reg. to Reg. Model**: Maps virtual registers to physical registers but spills excess to memory.
        - **Mem. to Mem. Model**: Maps some memory locations to a set of names modeling physical registers.
    - **Assignment:** Maps allocated names to physical registers, ensuring code fits into the available registers.
3. **Local Register Allocation and Assignment:**
    - Allocation within a basic block.
    - **Approaches:** Top-down (based on frequency count) and bottom-up.
    - **Challenges:** Liveness and live ranges, determining when values are in use.
4. **Global Register Allocation and Assignment:**
    - Deals with live ranges across multiple basic blocks.
    - **Issues:**

- Minimizing the impact of spill code on execution time, code space, and data space.
- Complexity due to loop structures and varying execution frequencies.

- **Approach:** Graph coloring using interference graphs.
    - Attempt to construct a k-coloring for the graph, where 'k' is the number of physical registers.
    - Spilling values to memory simplifies the graph and aids in halting the algorithm.
- **Subproblems:** Discovering global live ranges, estimating spilling costs, building interference graphs.

5. **Handling Spills:**
    - When no color is found for live ranges, spilling is necessary but not always optimal.
    - **Strategies:** Live range splitting to reduce interferences, coalescing copies to reduce interference graph complexity.
6. **Top-Down vs. Bottom-Up Allocation:**
    - **Top-Down:** Priority-based ordering of constrained nodes, with spill and iterate philosophy for handling spills.
    - **Bottom-Up:** Constructs an order where most nodes are colored in an unconstrained graph, potentially lessening spill code usage.



Register allocation is a complex process involving trade-offs between compile time, execution time, code size, and memory usage. Effective allocation and assignment strategies are essential for optimizing program performance on target architectures.

Explain code generation form DAGs.

Generating Code From DAGs:

Code generation from Directed Acyclic Graphs (DAGs) offers advantages over linear representations like three-address statements. Rearranging the order of computations becomes more intuitive with DAGs.

Rearranging the Order:

Consider the following basic block with three-address statements:

1. **t1 : = a + b**
2. **t2 : = c + d**
3. **t3 : = e - t2**
4. **t4 : = t1 - t3**

The generated code sequence might not be optimal due to the fixed order:

```
MOV a , R0
ADD b , R0
MOV c , R1
ADD d , R1
MOV R0 , t1
MOV e , R0
SUB R1 , R0
MOV t1 , R1
SUB R0 , R1
MOV R1 , t4
```

However, by rearranging the basic block, we can optimize the code:

```
MOV c , R0
ADD d , R0
MOV a , R0
SUB R0 , R1
MOV a , R0
ADD b , R0
SUB R1 , R0
MOV R0 , t4
```

This rearrangement eliminates redundant instructions, improving efficiency.

Heuristic Ordering for DAGs:

A heuristic ordering algorithm prioritizes evaluating a node's leftmost argument first. The algorithm iterates through unlisted interior nodes, listing them based on their parents' availability.

Algorithm:

1. Iterate while unlisted interior nodes remain.
2. Select an unlisted node **n** whose parents are listed.
3. List **n**.
4. Iterate while the leftmost child **m** of **n** has no unlisted parents and is not a leaf.
5. List **m** and update **n** to **m**.

Example: Initially, the only node with no unlisted parents is 1 so set n=1 at line (2) and list 1 at line (3). Now, the left argument of 1, which is 2, has its parents listed, so we list 2 and set n=2 at line (6). Now, at line (4) we find the leftmost child of 2, which is 6, has an unlisted parent 5. Thus we select a new n at line (2), and node 3 is the only candidate. We list 3 and proceed down its left chain, listing 4, 5 and 6. This leaves only 8 among the interior nodes so we list that. The resulting list is 1234568 and the order of evaluation is 8654321.

Code sequence:

```
t8 : = d + e

t6 : = a + b

t5 : = t6 - c

t4 : = t5 * t8

t3 : = t4 - e

t2 : = t6 + t4

t1 : = t2 * t3
```

This will yield an optimal code for the DAG on machine whatever be the number of registers.

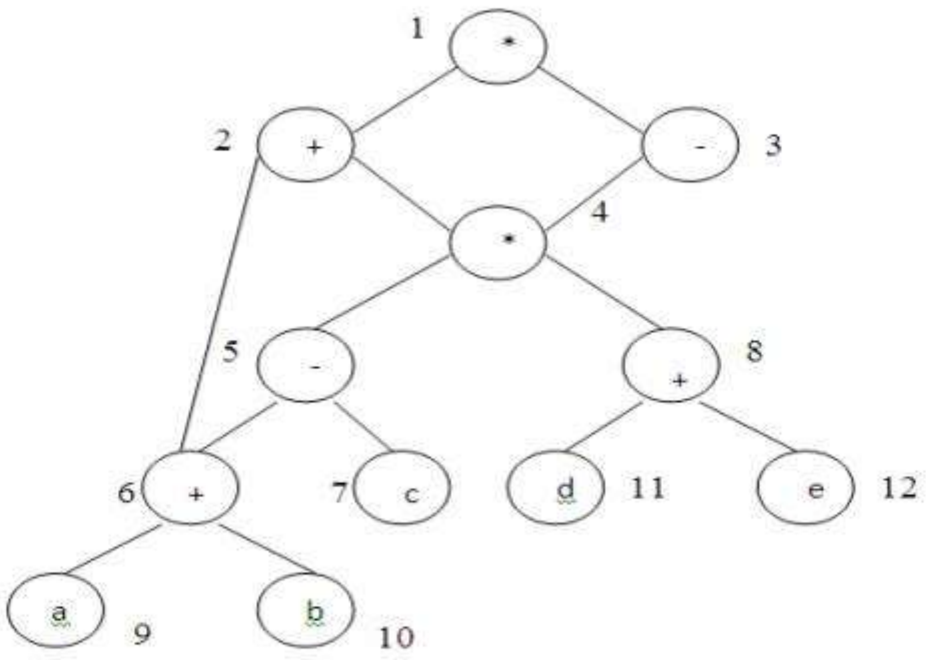


Fig. 4.7 A DAG