ASSIGNMENT - III

AUTOMATA THEORY AND COMPILER DESIGN

18AMC302T

By

S. MUTHURAM (927622BAL029)

P. M. NAVEEN KARTHICK (927622BAL033)

T. SIDDARTH (927622BAL041)

K. UDHAYAKUMAR (927622BAL051)

C. YADHAVARAMANAN (927622BAL055)

B.TECH ARTIFICIAL INTELLIGENCE AND

MACHINE LEARNING

III YEAR V SEMESTER

1. By taking the example of factorial program explain how activation record will look like for every recursive call in case of factorial (3). Distinguish between the source text of a procedure and its activation at run time.

In the case of a factorial program, each recursive call creates a new activation record (also known as a stack frame) to store the data specific to that call. Let's walk through this using the example of calculating the factorial of 3 (factorial(3)).

Factorial Program Example:

Here's a simple recursive program to calculate the factorial of a number \(n\):

```
def factorial(n):
  if n == 0:
    return 1
  else:
    return n * factorial(n - 1)
```

An activation record is created each time "factorial" is called recursively. Each activation record contains:

- **Return address** – Where the function should continue execution after it finishes.
- **Parameters** – The parameters for the current function call.
- **Local variables** – Any variables declared within the function.
- **Return value** – The result the function will eventually return.

For factorial(3), this program will involve the following recursive calls:

- factorial(3) calls factorial(2)
- factorial(2) calls factorial(1)
- factorial(1) calls factorial(0)

Activation Records for Each Call:

Let's visualize the stack of activation records as each call is made and then unwound.

1. factorial(3)

- **Return address:** Location to return after factorial(3) completes.
- **Parameter:** n = 3
- **Local variables:** None (but stores result of 3 factorial(2) once factorial(2) returns).
- **Return value:** Waiting for the result of factorial(2).

2. factorial(2)

- **Return address:** Location to return after factorial(2) completes (back to factorial(3)).
- **Parameter:** n = 2
- **Local variables:** None (but stores result of 2 * factorial(1) once factorial(1) returns).
- **Return value:** Waiting for the result of factorial(1).

3. factorial(1)

- **Return address:** Location to return after factorial(1) completes (back to factorial(2)).

- **Parameter:** n = 1
- **Local variables:** None (but stores result of 1 * factorial(0) once factorial(0) returns).
- **Return value:** Waiting for the result of factorial(0).

4. factorial(0)

- **Return address:** Location to return after factorial(0) completes (back to factorial(1)).
- **Parameter:** n = 0
- **Local variables:** None.
- **Return value:** Returns 1 directly (base case).

When factorial(0) returns 1, each previous call (from factorial(1) up to factorial(3)) resumes and completes its multiplication until the final result is returned to the original call.

Distinction between Procedure Source Text and Runtime Activation:

1. Source Text of the Procedure: The source text is the actual code written for the `factorial` function. It defines the logic of how to calculate the factorial but does not specify how each recursive call will be managed in memory at runtime.

2.Activation at Runtime: Each call to the `factorial` function results in a new activation record on the call stack at runtime. These records track the current state of each call independently, with its own parameters and pending return address. This mechanism allows recursion to work by isolating each call's context, enabling the function to return to the correct place once the recursive call completes.

2. Explain in detail stack based allocation of space. Justify use of activation trees and activation records in stack based allocation. Write a simple program to implement a sort procedure. Draw the activation tree when the numbers 9, 8, 7, 6, 5, 4, 3, 2, 1 are sorted. What is the largest number of activation records that ever appear together on the stack?

Stack-based allocation is a method of managing memory during program execution, particularly in languages that support recursive function calls. It relies on a structure called the call stack, where each function call generates a new activation record (or stack frame) to hold the necessary information for that function execution, such as parameters, local variables, and return addresses.

Stack-Based Allocation Works

- **Push on Call:** When a function is called, a new activation record is pushed onto the stack. This record contains all necessary data specific to that function instance.
- **Pop on Return:** When a function finishes executing, its activation record is popped off the stack. The stack then returns control to the calling function.
- **LIFO Order:** Since the stack operates in a last-in, first-out (LIFO) manner, the most recent function call is completed first, which makes it suitable for managing recursive calls.

Activation Trees and Activation Records in Stack-Based Allocation

1. Activation Records: These hold all data related to a single call to a procedure, including:

- The parameters passed to the function.
- The local variables within the function.
- The return address, indicating where to resume after the function completes.
- Temporary data or saved registers, depending on the function's needs.

2. Activation Trees: An activation tree is a conceptual representation showing the hierarchy of function calls, especially in recursive functions. Each node in the tree represents an activation record, and each branch shows the calling relationship between functions. The depth of this tree represents the recursive depth and is proportional to the maximum stack usage during execution.

By visualizing function calls with an activation tree, we can easily understand the sequence of recursive calls, and by analysing it, we can determine the maximum number of activation records that may appear simultaneously on the stack.

Sort Program

Below is an example of a recursive "QuickSort" function to sort an array of numbers. QuickSort is a recursive algorithm that partitions the array into smaller sub-arrays and recursively sorts them.

```
def quicksort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        quicksort(arr, low, pi - 1)  # Sort elements before partition
        quicksort(arr, pi + 1, high)  # Sort elements after partition
def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1
```

Example:

```
arr = [9, 8, 7, 6, 5, 4, 3, 2, 1]
quicksort(arr, 0, len(arr) - 1)
print(arr)
```

Activation Tree for Sorting [9, 8, 7, 6, 5, 4, 3, 2, 1].

To illustrate the activation tree, let's follow the sequence of recursive calls made by `quicksort` on this array:

- **Root Call:** quicksort(arr, 0, 8) with pivot = 1
- This root call splits into two recursive calls:
  - ✓ quicksort(arr, 0, pivot_index - 1)
  - ✓ quicksort(arr, pivot_index + 1, 8)

The activation tree shows each recursive call branching out as QuickSort further subdivides the array into smaller sections:

```
                 quicksort(0, 8)
                  /        \
          quicksort(0, 6)    quicksort(8, 8)
             /     \
     quicksort(0, 4)  quicksort(6, 6)
          /     \
    quicksort(0, 2)  quicksort(4, 4)
        /     \
    quicksort(0, 0) quicksort(2, 2)
```

Maximum Number of Activation Records on the Stack

The maximum number of activation records on the stack corresponds to the maximum depth of recursion for the given array. In the worst case for QuickSort (already sorted or reverse sorted), the partitioning divides the array into one large part and one small part repeatedly.

For the array `[9, 8, 7, 6, 5, 4, 3, 2, 1]`, the maximum depth of recursion would be 8, as each recursive call partitions the array with only one element shifting to the sorted side at each level. So, the largest number of activation records that appear together on the stack is 8. This worst-case scenario represents an unbalanced recursive call structure. However, with optimal partitioning, the recursive depth would be approximately "$\log_2(n)$".

3. Create DAG and three – address code for the following C program.

```
 i = 1;
s = 0;
while ( i<= 10)
{ s = s+ a[i] [i];
i = i + 1; }
```

To create a Directed Acyclic Graph (DAG) and generate Three-Address Code (TAC) for the given C program, we'll go through each step and construct the DAG for the expressions used, followed by the TAC representation.

Program

```
i = 1;

s = 0;

while (i <= 10) {

    s = s + a[i][i];

    i = i + 1; }
```

Step 1: DAG Construction

1. Initialization of `i` and `s`:

- `i = 1` assigns the constant value `1` to `i`.
- `s = 0` assigns the constant value `0` to `s`.

2. Loop Condition `i <= 10`: The condition checks if `i` is less than or equal to `10`.

3. Inside the Loop: Updating `s` and `i`:

- `s = s + a[i][i]` updates `s` by adding `a[i][i]` to it. Here, `a[i][i]` is the element in array `a` at position `(i, i)`.
- `i = i + 1` increments `i` by `1`.

DAG Representation

In the DAG, nodes represent unique expressions or variables. We reuse nodes if the same expression appears more than once. Here's how the DAG would look:

```
 (s)           (10)

 |              |

 |              |

 +--------+       +----->(<=)----------------+

 |     |           |            |

(0)    (+) <---+        |            |

      |    |       |            |

      |    |       |            |

     (s)   (a[i][i])  (i) <---(1)      loop

           |        \   /         |

           |         +---+          |

          [i][i]                exit

             |

            (i) <---+
```

1. s = 0: Represents `s` being initialized to `0`.

2. Condition i <= 10: Checks whether `i` is less than or equal to `10` to control the loop.

3. s = s + a[i][i]: Updates `s` by adding `a[i][i]`.

4. i = i + 1: Increments `i` by `1`.

Three-Address Code (TAC)

Using the DAG structure, we can generate TAC for each step of the program.

Initializations

```
t1 = 1        // Temporary variable t1 holds the value 1

i = t1        // Assigns i = 1

t2 = 0        // Temporary variable t2 holds the value 0

s = t2        // Assigns s = 0
```

Loop (while i <= 10)

1. Condition: Check if `i <= 10`

```
L1:   t3 = 10        // Temporary variable t3 holds the value 10

      t4 = i <= t3     // t4 holds the result of (i <= 10)

      if t4 == 0 goto L2   // If t4 is false (0), exit loop (goto L2)
```

2. Loop Body: Updating s with s + a[i][i].

```
t5 = a[i][i]     // Load value of a[i][i] into temporary variable t5

t6 = s + t5      // Update s by adding t5 (i.e., s + a[i][i])

s = t6           // Assign the result back to s
```

Incrementing i:

```
t7 = i + 1       // Increment i by 1

i = t7           // Assign the incremented value back to i

goto L1          // Go back to the start of the loop
```

3. Exit Loop:

```
L2:   (exit loop)      // Exit point of the loop
```

Final Three-Address Code

Putting it all together, the TAC is as follows:

```
t1 = 1
i = t1
t2 = 0
s = t2
L1:   t3 = 10
     t4 = i <= t3
     if t4 == 0 goto L2
     t5 = a[i][i]
     t6 = s + t5
     s = t6
     t7 = i + 1
     i = t7
     goto L1
L2:   // Exit loop
```

Largest Number of Activation Records

In this code, there is only one loop and no nested function calls, so the maximum number of activation records on the stack at any point is 1 (for the main function or calling scope). Each iteration of the loop reuses the same space for the variables without requiring additional stack frames, since no further functions are called recursively.

4.Considering the diverse data types and the need for both performance and efficiency, which storage allocation strategy (or combination of strategies) would you recommend for managing the platform's data? Discuss the advantages and disadvantages of your chosen strategy in terms of fragmentation, access speed, and space utilization. How would you address potential challenges associated with your recommended approach?

For managing data efficiently on a platform that handles diverse data types, a combination of stack-based, heap-based, and static allocation strategies would provide the best balance between performance, efficiency, and memory utilization. Let's discuss each of these strategies in terms of fragmentation, access speed, and space utilization, along with addressing potential challenges.

1. Static Allocation

- **Description:** In static allocation, memory for variables is allocated at compile time. Global variables and constants often use this strategy.

- ❖ **Advantages:**
  - ✓ Low Fragmentation as the memory is allocated once and persists throughout the program's runtime, there's minimal risk of fragmentation.
  - ✓ High Access Speed addresses are fixed, accessing static memory is fast and predictable.
  - ✓ Efficient for Constant Data best suited for data that does not change in size or require dynamic resizing.
- ❖ **Disadvantages:**
  - ✓ Limited Flexibility not suitable for data structures requiring dynamic resizing.
  - ✓ High Space Utilization memory is allocated regardless of usage patterns, which can lead to wasted space if not all static variables are used throughout the runtime.
  - ✓ Use Case strategy is ideal for configuration parameters, constants, and other data elements with a predictable, fixed size. For platform-wide settings or shared constants, static allocation ensures high-speed access without overhead.

2. Stack-Based Allocation

- ❖ **Description:** Stack-based allocation is used for function-local variables, particularly in recursive and re-entrant functions. It follows a Last-In-First-Out (LIFO) model.
- ❖ **Advantages:**
  - ✓ **Fast Access and Deallocation:** Stack allocation is very efficient because memory is automatically reclaimed as each function call ends, with no need for explicit deallocation.
  - ✓ **Minimal Fragmentation:** Since the stack grows and shrinks in a predictable order, there's no external fragmentation.
- ❖ **Disadvantages:**
  - ✓ Limited Lifespan variables allocated on the stack are local to function calls, limiting their lifespan to the scope of that function.
  - ✓ Fixed Size Limitation stack memory is usually limited in size, which can restrict the size of allocated data, making it unsuitable for large, dynamic data structures.
  - ✓ Use Case of Stack-based allocation is suitable for short-lived, small variables (like local counters, function parameters, or temporary buffers). It allows efficient function execution and is well-suited for managing small data types and temporary data structures without risk of memory leaks.

3. Heap-Based Allocation

- ❖ **Description:** Heap-based allocation is used for data that needs a dynamic lifespan, allowing for memory to be allocated and deallocated as needed.
- ❖ **Advantages:**
  - ✓ **Dynamic Sizing:** Ideal for data structures like arrays, lists, and trees where the size may vary.
  - ✓ **Extended Lifespan:** Data persists across function calls and can be explicitly freed when no longer needed.
- ❖ **Disadvantages:**
  - ✓ Fragmentation Risk heap memory is prone to fragmentation, especially with frequent allocations and deallocations of varying sizes.
  - ✓ Slower Access Speed memory access can be slower than stack or static memory due to indirect addressing.
  - ✓ Use Case of Heap allocation is crucial for larger or more complex data structures that need flexibility, such as user-defined objects, arrays, and buffers for streaming data. Heap allocation

is the best choice for data that grows or shrinks at runtime, such as collections of user data or file buffers.

Recommended Combination Strategy and Justification

Recommended Approach:

Using a combination of all three strategies allows efficient and flexible management of diverse data types on the platform. Here's a suggested approach:

- ❖ **Static Allocation:** For constant or global configuration data that is not expected to change and is frequently accessed.
- ❖ **Stack-Based Allocation:** For temporary or local data within functions to reduce overhead and take advantage of the stack's efficient memory management.
- ❖ **Heap-Based Allocation:** For dynamically-sized data structures or objects with unpredictable lifetimes.

Advantages:

- ❖ **Fragmentation Control:** By restricting the heap usage to large, flexible data structures only, the risk of fragmentation is contained.
- ❖ **Access Speed:** Static and stack-based allocations ensure fast access for frequently used or temporary data, with slower access limited to dynamic structures.
- ❖ **Efficient Space Utilization:** The combination ensures that static and stack space is used optimally, reducing reliance on the fragmented heap space.

Challenges and Solutions:

1. Heap Fragmentation: To address heap fragmentation, a memory pooling strategy can be employed. This involves allocating blocks of memory in advance for frequently used data sizes, which reduces fragmentation and accelerates allocation.

2. Garbage Collection: Explicit deallocation can be challenging to manage, particularly in heap-based structures. Incorporating a garbage collection mechanism (or using smart pointers in languages that support them) can help automate memory management and prevent leaks.

3. Stack Overflow: For large or deeply recursive data, the stack may become overburdened. Optimizing recursive functions through techniques like tail recursion or transforming recursion to iteration (where possible) can help mitigate stack overflow.

This hybrid strategy leverages the strengths of each storage allocation type, achieving a balance between performance, flexibility, and efficient memory usage on the platform.

5. (i.) Describe in detail about optimization of basic blocks.

(ii.) Construct the DAG for the following Basic block & explain it

1. t1: = 4 * i

2. t2:= a [t1]

3. t3: = 4 * i

4. t4:= b [t3]

5. t5:=t2*t4

6. t6:=Prod+t5

7. Prod:=t6

8. t7:=i+1

Optimization of basic blocks is a critical part of compiler optimization, aimed at improving code efficiency by eliminating redundant computations, reducing execution time, and minimizing memory usage. A basic block is a sequence of instructions with a single entry point and a single exit point, meaning that all instructions in the block execute sequentially without branching.

Block Optimization Techniques

- **Constant Folding:** This involves evaluating constant expressions at compile time rather than runtime. For example, if an expression like 4 * 5 appears, it is replaced by 20 directly in the code.
- **Common Subexpression Elimination (CSE)**: If an expression is calculated multiple times within a block, it is computed only once, and subsequent references are replaced with the stored result. This reduces redundant computations.
- **Copy Propagation:** If a variable is assigned the value of another variable (e.g., b = a), further uses of b can be replaced by a, eliminating the copy operation.
- **Dead Code Elimination:** Any code that doesn't affect the program's outcome (e.g., variables that are assigned but never used) is removed, reducing memory usage.
- **Strength Reduction:** Expensive operations are replaced with equivalent, less costly operations. For example, x * 2 can be replaced by x << 1 (left shift) in some languages to speed up execution.
- **Algebraic Simplification:** Simplifying algebraic expressions, such as replacing x + 0 with x, to improve efficiency.

DAG (Directed Acyclic Graph)

The DAG represents expressions within a basic block, helping in identifying common subexpressions, performing copy propagation, and tracking the dependencies among variables and operations. Each unique subexpression is represented as a node in the DAG, and the same subexpression is referenced by a single node, eliminating redundant computations.

Constructing the DAG

Let's construct the DAG for the following basic block:

1. t1 := 4 * i

2. t2 := a[t1]

3. t3 := 4 * i

4. t4 := b[t3]

5. t5 := t2 * t4

6. t6 := Prod + t5

7. Prod := t6

8. t7 := i + 1

Step-by-Step DAG Construction

- **Instruction 1:** t1 := 4 * i
  - ✓ Create a node for the multiplication 4 * i.
  - ✓ t1 is assigned this node.
- **Instruction 2:** t2 := a[t1]
  - ✓ Create a node for the array access a[t1], with t1 as the index.
  - ✓ Assign t2 to this node.
- **Instruction 3:** t3 := 4 * i
  - ✓ Since 4 * i is the same expression as in Instruction 1, we reuse the node from Instruction 1.
  - ✓ Assign t3 to this existing node (no new node is created).
- **Instruction 4:** t4 := b[t3]
  - ✓ Create a node for the array access b[t3], with t3 as the index.
  - ✓ Assign t4 to this node.
- **Instruction 5:** t5 := t2 * t4
  - ✓ Create a node for the multiplication t2 * t4.
  - ✓ Assign t5 to this node.
- **Instruction 6:** t6 := Prod + t5
  - ✓ Create a node for the addition Prod + t5.
  - ✓ Assign t6 to this node.
- **Instruction 7:** Prod := t6
  - ✓ Update Prod to reference the node for t6.
- **Instruction 8:** t7 := i + 1
  - ✓ Create a node for the addition i + 1.
  - ✓ Assign t7 to this node.

Final DAG Structure

The DAG for this basic block is as follows:

```
  (4 * i)          (Prod)    (1)
     |               |        |
     |               |        |
  t1/t3             t6         |
  /   \              \       /
a[t1]   b[t3]       Prod + t5   i + 1
  |       |            |
  t2      t4           t7
   \      /
  (t2 * t4)
      |
      t5
```

Explanation of the DAG Nodes

- **(4 * i):** This node represents the common subexpression 4 * i, used by both t1 and t3.
- **a[t1] and b[t3]:** These nodes represent the array accesses a[t1] and b[t3], where t1 and t3 are indices.
- **t2 * t4:** This node represents the multiplication of t2 and t4.
- **Prod + t5:** This node represents the addition of Prod and t5.
- **i + 1:** This node represents the increment of i by 1.

Optimized Three-Address Code (TAC)

The DAG allows us to remove redundant calculations and generate an optimized TAC:

```
1. t1 := 4 * i        // Common subexpression used by both t1 and t3

2. t2 := a[t1]        // Access a[t1]

3. t4 := b[t1]        // Use t1 instead of recomputing 4 * i

4. t5 := t2 * t4      // Calculate t2 * t4

5. t6 := Prod + t5    // Add t5 to Prod

6. Prod := t6         // Update Prod

7. t7 := i + 1        // Increment i by 1
```

Advantages of DAG Optimization

- **Eliminates Redundant Computation:** The common subexpression 4 * i is only computed once, reducing execution time.
- **Reduced Code Size:** By avoiding duplicate expressions, the TAC generated is more concise.
- **Improves Performance:** With fewer instructions to execute, the optimized code executes faster.
- **Easier Code Maintenance:** The optimized code is cleaner and easier to understand, especially for complex expressions.