CSA1455

Compiler design for Lexical Analysis

Assignment - 4

Name: Sai Lokesh Malabothu

Reg.No: 192365023

Branch: CSE - cyber security

Date: 27 - 02 - 2025

Serial. No: 27

Managing Scope and Lifetime in Code

1. Scope Rules

Scope defines the Region of the code where a variable is accessible. In Intermediate code address code, abstract syntax tree or byte code. These variables follows Scope Rules similar to high level languages:

* Lexical Scope
* Dynamic Scope

2. Static vs Dynamic Lifetime

* Static Lifetime

→ Allocated at Compile time and Persists throughout Program execution.

→ Example: Global variables, static variables in c/c++.

Dynamic Life time:

* Created and destroyed at Run time.
* Managed using stack.
* Example: Function - Local variables.

3. Memory Allocation in Intermediate Code

* Stack Allocation

```
Func:
Push BP
Mov BP, SP
Sub SP, 8
...
Mov SP, BP
Pop BP
ret
```

* Heap Allocation

```
t1 = call malloc (16)
*t = 10
```

* Static Data Segment Allocation

```
global_var: .word 42
```

a. How does Intermediate code handle variable scope?

Intermediate code plays a crucial Role in Maintaining variable scope by translating high level language Rules into a structured format that compiler or Interpeter can Manage.

1. Scope Management

* Scope defines the visibility and life time of a variable. variable scope is managed through:

* Lexical scope

* Dynamic scope

2. Scope Implementation techniques

a) Symbol Tables

→ name

→ type

→ Memory location

→ scope

b) stack based scope Handling

c) Heap based scope Handling

d) Register Allocation for Temporary variables

3. Scope Rules

a) Global scope

b) Local scope

c) Block scope

d) Temporary scope

t1 = a + b

B. Explain the difference between static and dynamic variable lifetime?

The Life time of a variable Refers to the duration for which it exist in memory during program Execution. variables can have static or dynamic lifetime based on how and when they are allocated.

Feature	static lifetime	Dynamic lifetime
Memory allocation	At compile time	At Run time
Storage	Data Segment	Stack (or) Heap
Scope	Global / Function static	Local
Deallocation	Never	Automatic/Manual
Persists blw calls?	Yes	No
Example	static int x = 5;	int *P = Malloc (sizeof(int));

Conclusion:

* Static Lifetime variables are allocated once and Persist throughout Execution.

* Dynamic Lifetime variables are allocated at Runtime and exist only for a specific Execution Content.

C. How does Memory Allocation differ between Local & Global variables?

Memory Allocation for local and Global variables differs in terms of where they are stored, when they are Allocated, and how long they persist during Program Execution.

Feature	Stack	Data Segment
Storage	Stack	Data Segment
Allocation	At Run time	At Compile time
Deallocation	Automatically	Never
Scope	Limited	Accessible
Lifetime	Short	Long
Access Speed	faster	Slightly Slower
Example	int x = 10;	int g = 100;

Conclusion:

* Local variables use the stack, are temporary, and are automatically managed.

* Global variables use the data segment, persist throughout execution, and must be managed carefully to avoid unwanted modifications.

D. Generate TAC for a block-scoped variable declaration.

Example Code:

```
void func() {
    int a = 5;
    {
        int b = 10;
        a = a+b;
    }
}
```

TAC Generation:

```
Func:
 Push BP
 Mov BP, SP
  Sub SP, 8
Mov [BP-4], 5
Mov [BP-8], 10
 Mov eax, [BP-4]
 add eax, [BP-8]
 Mov [BP-4], eax

 Mov SP, BP
  Pop BP
  ret
```

Explanation:

* Stack Allocation
* Variable scope
* Block scope management.

E. Discuss how garbage Collection manages Memory in dynamic scope.

Garbage Collection automatically manages Memory by Reclaiming dynamically allocated objects that are no longer accessible.

Garbage Collection operates by identifying and Reclaiming Memory that is no longer in use.

Example:

```
a = [1,2,3]
b = a
del a
del b
```

Mark & sweep Algorithm

```
class Demo {
public static void main(String Args[]){
    Demo obj = new demo();
    obj = null;
  }
}
```

Memory Management in dynamic scope

Example (Python)

```
import gc
class Node:
    def __int__(self, value):
        self.value = value
        self.next = None
n1 = node(10)
n2 = node(20)
n1.next = n2
n2.next = n1
del n1, n2
gc.collect()
```

Conclusion:

* GC simplify Memory Management but comes with a Runtime overhead, which Modern optimizations minimize effectively.