

System Programming and Compiler Construction

<https://github.com/Dare-marvel/>

MSE MARCH 2020

Illustrate the design of an absolute loader with the help of an example

5.2 DESIGN OF AN ABSOLUTE LOADER

We introduce the general topic of loader design by presenting a design of an absolute loader.

With an absolute loading scheme the programmer and the assembler perform the tasks of allocation, relocation, and linking. Therefore, it is only necessary for the loader to read cards of the object deck and move the text on the cards into the absolute locations specified by the assembler.

There are two types of information that the object deck must communicate from the assembler to the loader. First, it must convey the machine instructions that the assembler has created along with the assigned core locations. Second, it must convey the entry point of the program, which is where the loader is to transfer control when all instructions are loaded. Assuming that this information is transmitted on cards, a possible format is shown in Figure 5.10.

Note that in the card format shown the instructions are stored on the card as one core byte per column. For each of the 256 possible contents of an eight-bit

byte there is a corresponding punched card code (e.g., hexadecimal 00 is a column punched with five holes, 12-0-1-8-9, whereas a hexadecimal F1 is a column with a single punch in row 1). Thus, when a card is read, it is stored in core as 80 contiguous bytes.

Text cards (for instructions and data)

<i>Card column</i>	<i>Contents</i>
1	Card type = 0 (for text card identifier)
2	Count of number of bytes (1 byte per column) of information on card
3-5	Address at which data on card is to be put
6-7	Empty (could be used for validity checking)
8-72	Instructions and data to be loaded
73-80	Card sequence number

Transfer cards (to hold entry point to program)

<i>Card column</i>	<i>Contents</i>
1	Card type = 1 (transfer card identifier)
2	Count = 0
3-5	Address of entry point
6-72	Empty
73-80	Card sequence number

FIGURE 5.10 Card formats for an absolute loader

The algorithm for an absolute loader is quite simple. The object deck for this loader consists of a series of text cards terminated by a transfer card. Therefore, the loader should read one card at a time, moving the text to the location specified on the card, until the transfer card is reached. At this point the assembled instructions are in core, and it is only necessary to transfer to the entry point specified on the transfer card. A flowchart for this process is illustrated in Figure 5.11.

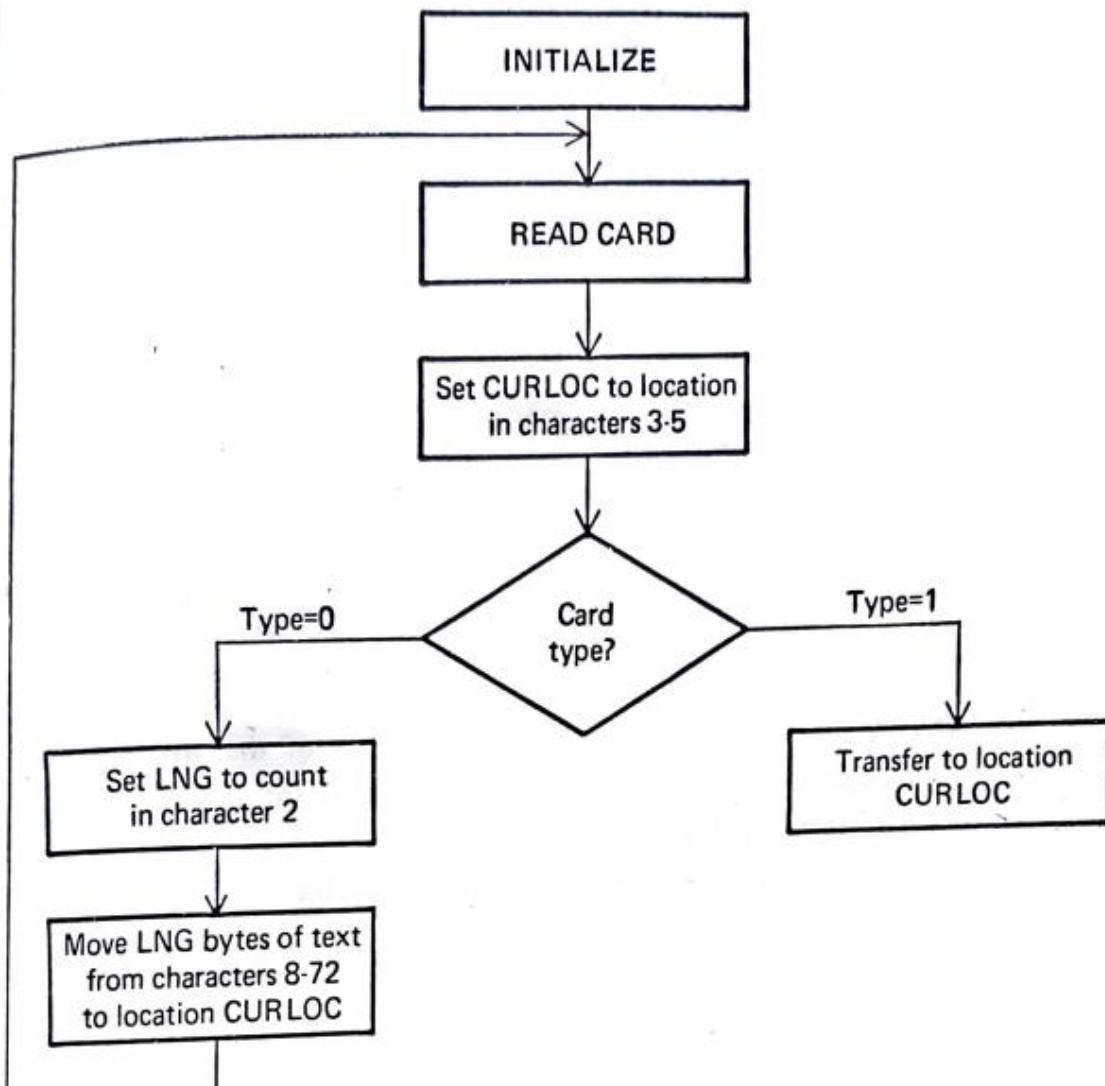


FIGURE 8.11 Absolute loader

MSE MARCH 2018

Advantages and disadvantages of absolute loader

Advantages & Disadvantages

Advantages:

1. It is simple to design & implement.
2. This scheme allows multiple programs or the source programs written in different languages.
3. Respective language converter will convert it to the common object file.
4. The process of execution is efficient.
5. No linking & relocation required.

Disadvantages:

1. Programmer should specify address to assembler for the program where it should be loaded.
2. If modification done, update the address of program.
3. For that, it is necessary for a programmer to know the memory management

Advantages:

1. It is simple to implement.
2. This scheme allows multiple programs or the source programs written in different languages. If there are multiple programs written in different languages then the respective language assembler will convert it to the language and common object file can be prepared with all the ad resolution.
3. The task of loader becomes simpler as it simply obeys the instruction regarding where to place the object code to the main memory.
4. The process of execution is efficient.

Disadvantages:

1. In this scheme, it's the programmer's duty to adjust all the inter-segment addresses and manually do the linking activity. For that, it is necessary for a programmer to know the memory management.
2. If at all any modification is done to some segment the starting address of immediate next segments may get changed the programmer has to take care of this issue and he/she needs to update the corresponding starting address on any modification in the source.

Advantages

1. **Simplicity:** Absolute loaders are simple and easy to design and implement.
2. **Efficiency:** They are efficient in their operation.

3. **Memory Utilization:** Since more core memory is available to the user, there is no memory limit.
4. **Allocation:** The loader allocates memory for the program in the main memory2.
5. **Linking:** It combines two or more separate object programs or modules and supplies necessary information2.
6. **Loading:** It brings the object program into the main memory for execution.

Disadvantages

1. **Address Specification:** The programmer must specifically tell the assembler the address where the program is to be loaded. This can be a complex task.
2. **Subroutine Libraries:** It's difficult to use subroutine libraries with an absolute loader because when subroutines are referenced, the programmer must specify their address whenever they are called.
3. **Relocation:** There are some address-dependent locations in the program, and these address constants must be modified to fit the available space2. This can be done by the loader and this is known as relocation2.
4. **Loading:** The loader performs loading; hence, the assembler must provide the loader with the object program2.
5. **Execution:** At the point of execution, the software will begin to run after it has loaded2. The object program is the sequence of the object records2.

With respect to direct linking loaders, what is the use of GEST and LESA?

In the context of a direct linking loader, GEST (Global External Symbol Table) and LESA (Local External Symbol Array) play crucial roles¹:

1. **GEST (Global External Symbol Table):** This is used to store each external symbol and its corresponding assigned core address¹. An external symbol is a symbol that is defined in one program module but can be used in other modules. The GEST helps in managing these symbols and ensuring that they are correctly linked during the loading process.
2. **LESA (Local External Symbol Array):** This is used to establish correspondence between ESD ID (External Symbol Dictionary Identifier) used in ESD & RLD (Relocation & Linkage Directory) cards¹. This helps in managing the local symbols that are defined and used within the same program module.

These two components work together to ensure that all symbols, both local and external, are correctly managed and linked when the program is loaded into memory. This is crucial for the correct execution of the program.

- v. *Global External Symbol Table (GEST)*: It is used to store each external symbol and its corresponding core address.

External symbol (8 bytes) character	Assigned core (4 bytes) address decimal
“PG1bbbbbb”	104
“PG1ENT1b”	124

- vi. *Local external symbol array (LESA)*: The external symbol is used for relocation and linking purpose. This is used to identify the RLD card by means of an ID number rather than the symbols name. The ID number must match an SD or ER entry on the ESD card

Assigned core address of corresponding symbol [4 bytes]
104
124
134
....
....

This technique saves space and also increases the processing speed.

Write a short note on input buffering in lexical analysis with example

Sure, input buffering is a crucial concept in lexical analysis, which is a part of the compiler design¹.

Input Buffering in Lexical Analysis¹²: Lexical analysis scans the input string from left to right one character at a time to identify tokens. It uses two pointers to scan tokens¹³:

- **Begin Pointer (bptr)**: It points to the beginning of the string to be read³.
- **Look Ahead Pointer (lptr)**: It moves ahead to search for the end of the token³.

Input buffering allows a compiler to check if a sequence of characters matches any known tokens². For example, a token like “if” could be matched against a list of pre-defined tokens for that language, allowing the language parser to understand this keyword².

Example¹: Consider the statement `int a, b;`. Initially, both the pointers point to the first character of the input string. The look ahead pointer moves ahead to search for the end of the lexeme. As soon as a blank space is encountered, it indicates the end of the lexeme. In this example, as soon as the look ahead pointer encounters a blank space, the lexeme “int” is identified¹.

Buffering Techniques¹: Input buffering is used to improve performance and reduce overhead. The basic idea behind input buffering is to read a block of input from the source code into a buffer, and then process that buffer before reading the next block¹. There are two methods used in this context¹:

- [One Buffer Scheme](#): Only one buffer is used to store the input string¹.
- [Two Buffer Scheme](#): The buffer is divided into two halves⁴. One pointer marks the beginning of the token being discovered. A look ahead pointer scans ahead of the beginning point, until the token is discovered⁴.

Remember, these are general concepts and methods. The actual implementation can vary depending on the specific needs of the compiler and the characteristics of the source code being compiled.

What are the different storage allocation strategies in compiler design? explain stack storage allocation strategy in detail.

In compiler design, there are mainly three types of storage allocation strategies1:

1. [Static Allocation](#): In static allocation, storage for all data objects is laid out at compile time1. The address of these identifiers will be the same throughout1. The memory will be allocated in a static location once it is created at compile time1.
2. [Heap Allocation](#): Heap allocation is used where the Stack allocation lacks1. If we want to retain the values of the local variable after the activation record ends, which we cannot do in stack allocation, here LIFO scheme does not work for the allocation and de-allocation of the activation record1. Heap is the most flexible storage allocation strategy we can dynamically allocate and de-allocate local variables whenever the user wants according to the user needs at run-time1.
3. [Stack Allocation](#): Stack is commonly known as Dynamic allocation1. Dynamic allocation means the allocation of memory at run-time1. Stack is a data structure that follows the LIFO principle so whenever there is multiple activation record created it will be pushed or popped in the stack as activations begin and ends1. Local variables are bound to new storage each time whenever the activation record begins because the storage is allocated at runtime every time a procedure or function call is made1. When the activation record gets popped out, the local variable values get erased because the storage allocated for the activation record is removed1.

Now, let's delve deeper into the Stack Allocation Strategy1:

Stack allocation is a runtime storage management mechanism for the compiler whereby activation records are pushed and popped onto the stack as activations begin and end2. An activation record contains the locals so that they are bound to fresh storage in each activation record3. The value of locals is deleted when the activation ends3. It works on the basis of last-in-first-out (LIFO) and this allocation supports the recursion process3.

For example, consider a recursive function to calculate the factorial of a number:

```
int fact(int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return (n * fact(n - 1));  
}
```

In this example, each recursive call to `fact` creates a new activation record on the stack. This record contains the local variable `n` and the return address. When `fact` is called with `n = 6`, six activation records are created on the stack. [As each call to `fact` completes, its activation record is popped from the stack, and control returns to the calling function3.](#)

Remember, these are general concepts and methods. The actual implementation can vary depending on the specific needs of the compiler and the characteristics of the source code being compiled.

Explain the importance and contents of symbol table with reference to RUN TIME storage management

In the context of runtime storage management, a symbol table plays a crucial role in tracking and managing information about variables, functions, and other program entities during the execution of a program. The symbol table is a data structure that stores metadata about the symbols used in the program, helping the compiler or interpreter manage memory and resolve references efficiently at runtime.

Here are the key aspects of the symbol table with reference to runtime storage management:

Variable Information:

- The symbol table contains information about variables, including their names, data types, memory locations, and scope.
- During runtime, the symbol table is consulted to allocate memory for variables and to retrieve their values when needed.

Function Information:

- For functions and procedures, the symbol table stores details such as function names, return types, parameter types, and memory addresses.
- The symbol table aids in managing the activation records (stack frames) for each function call during runtime.

Memory Allocation:

- The symbol table helps in managing memory by keeping track of allocated memory locations for variables and other program entities.
- It ensures that memory is allocated and deallocated appropriately during the program's execution.

Address Resolution:

- The symbol table is used to resolve the memory addresses of variables and functions during runtime.
- When a variable is accessed or a function is called, the symbol table provides the necessary information to locate the corresponding memory address.

Scope Management:

- Symbol tables are organized based on the scope of variables and functions in the program.
- During runtime, the symbol table helps in resolving variable and function names based on their scope, ensuring correct references and preventing naming conflicts.

Dynamic Memory Management:

- In languages with dynamic memory allocation (e.g., heap memory), the symbol table plays a role in tracking dynamically allocated memory and managing its lifecycle.

Error Handling:

- The symbol table is instrumental in detecting and reporting errors related to undefined variables, type mismatches, or other issues during runtime.

In summary, the symbol table is a critical component of runtime storage management, helping to organize and manage program entities' information, allocate and deallocate memory, resolve addresses, and ensure proper execution of the program. It facilitates efficient runtime behavior by providing the necessary data for memory management and addressing during the execution of the program.

Suggest atleast two data structures for maintaining symbol table

There are several data structures suitable for maintaining a symbol table, each with its own advantages and use cases. Here are two commonly used data structures for symbol table management:

Hash Table:

- A hash table is a widely used data structure for implementing symbol tables due to its efficiency in providing constant time average-case complexity for basic operations (insertion, deletion, and lookup).
- In a symbol table context, variable and function names can be hashed to generate an index into the hash table, making it quick to locate and access information about a particular symbol.
- Collisions can be handled using techniques like chaining or open addressing.

Binary Search Tree (BST):

- A Binary Search Tree is another effective data structure for symbol table management, especially when a sorted order of symbols is desirable.
- In a BST, each node has a left subtree with values less than the node and a right subtree with values greater than the node. This arrangement allows for efficient searching and insertion operations.
- In the context of symbol tables, a BST can be used to maintain a sorted order of symbols based on their names, making it easy to perform alphabetical searches.

With reference to two pass assembler give the use of following pseudo ops using short example:

1. USING and DROP

2. LTORG

Sure, let's discuss these pseudo-ops:

1. [**USING and DROP: These are used in IBM mainframe assembly language¹²**](#)

The **USING** pseudo-op tells the assembler that a particular register contains a base address that will be used for address resolution. [The DROP pseudo-op tells the assembler to stop using the register for address resolution¹²](#). Here's a short example:

```
USING *,12 // Register 12 is now a base register
L 1,0(0,12) // Load from the address in register 12
DROP 12 // Register 12 is no longer a base register
```

2. **LTORG**: This pseudo-op is used in assembly languages like IBM mainframe assembly³⁴. The LTORG pseudo-op tells the assembler to create a literal pool, which is a table of constants used in the program³⁴. The assembler inserts the literal pool into the program at the location of the LTORG³⁴. Here's a short example:

```
LA 1,=A('HELLO') // Load address of the string 'HELLO' into register 1
LTORG // Assemble literals into a literal pool
```

In the first example, the string 'HELLO' is a literal. The =A(' HELLO ') syntax tells the assembler to add this literal to the literal pool. The LTORG then causes the assembler to insert the literal pool into the program at that location³⁴.

MSE MARCH 2019

Q.1	Enlist the four functions used while constructing a DFA directly from a regular expression without constructing an intermediate NFA? Write rules to compute any two of the distinct(not similar) four functions without giving examples. (State only rules. No detailed explanation required.)
Q.2	Consider the following DFA.

Listing all four functions- 1 mark

NODE n	$\text{nullable}(n)$	$\text{firstpos}(n)$
A leaf labeled ϵ	true	\emptyset
A leaf with position i	false	$\{i\}$
An or-node $n = c_1 c_2$	$\text{nullable}(c_1) \text{ or } \text{nullable}(c_2)$	$\text{firstpos}(c_1) \cup \text{firstpos}(c_2)$
A cat-node $n = c_1 c_2$	$\text{nullable}(c_1) \text{ and } \text{nullable}(c_2)$	$\begin{aligned} &\text{if } (\text{nullable}(c_1)) \\ &\text{firstpos}(c_1) \cup \text{firstpos}(c_2) \\ &\text{else firstpos}(c_1) \end{aligned}$
A star-node $n = c_1^*$	true	$\text{firstpos}(c_1)$

Rules for computing nullable and firstpos

"Rules for computing lastpos are same as that of firstpos."

Rules for computing followpos:-

1. If n is a cat-node with left child C_1 and right child C_2 , then for every position i in $\text{lastpos}(C_1)$, all positions in $\text{firstpos}(C_2)$ are in $\text{followpos}(i)$.
2. If n is a star-node, and i is a position in $\text{lastpos}(n)$, then all positions in $\text{firstpos}(n)$ are in $\text{followpos}(i)$.

2 marks for writing the rules for any of the functions mentioned.

ESE MAY 2022

Solve following MCQs

i) When is the symbol table generated in a two-pass assembler?

- a. Generated in second pass
- b. Generated and used only in second pass
- c. Generated in first pass
- d. Not generated at all

ii) Which derivation is generated by the bottom-up parser?

- a. Right-most derivation in reverse
- b. Leftmost derivation in reverse
- c. Right-most derivation
- d. Left-most derivation

iv) Which phenomenon happens when the non-terminal on the left side is repeated as the first symbol on the right side?

- e. Left-most derivation
- f. **Left recursion**
- 9. Left factoring
- h. Left Parsing

V) lexical analysis is used to generate sequences of ,

- a. tokens
- b. Variables
- c. constants
- d. none Of these

Explain Stack Storage allocation

Stack Storage Allocation is a runtime storage management technique used in programming¹. It's also known as **Dynamic Allocation**¹. Here's how it works:

1. **Activation Records:** When a function or procedure is called in a program, an activation record is created¹. This record contains information about the function including its local variables, parameters, and return address¹.
2. **Pushing to the Stack:** The activation record is then pushed onto the stack¹. The stack is a data structure that follows the Last-In-First-Out (LIFO) principle¹. This means that the most recently added item (in this case, the activation record) is the first one to be removed¹.
3. **Memory Allocation:** Memory for the local variables in the activation record is allocated at runtime, i.e., during the execution of the program¹. This is why it's called dynamic allocation¹.
4. **Popping from the Stack:** When the function call ends, the activation record is popped from the stack¹. The memory allocated for the local variables is deallocated, and their values are erased¹.
5. **LIFO Principle:** Because of the LIFO principle, the stack can handle nested function calls. When a function calls another function, a new activation record is pushed onto the stack. When the called function ends, its activation record is popped from the stack, and control returns to the calling function¹.

Here are some key points about stack allocation:

- It's a **temporary memory allocation scheme** where the data members are accessible only if the method that contained them is currently running².
- It **allocates or de-allocates the memory automatically** as soon as the corresponding method completes its execution².
- Stack memory allocation is considered **safer** as compared to heap memory allocation because the data stored can only be accessed by the owner thread².
- Memory allocation and de-allocation are **faster** as compared to Heap-memory allocation².
- Stack memory has **less storage space** as compared to Heap-memory².

Explain structure of general activation records and it's usage with reference to runtime environment

[Activation Records - GeeksforGeeks](#)

An **Activation Record**, also known as a **Stack Frame**, is a data structure that composes a call stack¹. It's used to manage the information needed by a single execution of a procedure²³. When a procedure is called, an activation record is created and pushed onto the stack²³. When the procedure finishes execution, the activation record is popped from the stack.

Usage in the Runtime Environment

The runtime environment uses activation records to manage the execution of programs, particularly the function calls and returns. Here's how they are used:

1. Function Call:
 - When a function is called, a new activation record is created and pushed onto the stack.
 - The return address is stored in the activation record to ensure the control can return to the correct place.
 - The function parameters are stored in the activation record.
 - The frame pointer is updated to point to the base of the new activation record.
 - The stack pointer is adjusted to allocate space for the new activation record.
2. Function Execution:
 - During the function execution, local variables and temporary values are stored in the activation record.
 - The function can access its parameters and local variables through offsets from the frame pointer or stack pointer.
3. Function Return:
 - When the function completes, its activation record is popped off the stack.
 - The stack pointer is adjusted to remove the space allocated for the activation record.
 - The return address is used to transfer control back to the caller function.
 - The previous frame pointer is restored.

Example of Activation Record Usage

Consider the following example in a high-level language like C:

```
void bar(int x) {  
    int y = x + 1;  
    // More code...  
}
```

```
void foo() {  
    int a = 10;  
    bar(a);  
    // More code...  
}
```

```
int main() {  
    foo();  
    return 0;  
}
```

Here's a step-by-step breakdown of how activation records are used:

1. Calling main:
 - An activation record for main is created and pushed onto the stack.
 - The activation record contains the return address for the end of the program.
2. Calling foo:
 - main calls foo, creating a new activation record for foo.
 - The return address to main is saved.
 - Space is allocated for foo's local variable a.
3. Calling bar:
 - foo calls bar, creating a new activation record for bar.
 - The return address to foo is saved.
 - The parameter x is stored in the activation record.
 - Space is allocated for bar's local variable y.
4. Returning from bar:
 - bar completes execution and its activation record is popped off the stack.
 - Control returns to foo, and the stack pointer and frame pointer are restored.
5. Returning from foo:
 - foo completes execution and its activation record is popped off the stack.
 - Control returns to main, and the stack pointer and frame pointer are restored.

Here's the general structure of an activation record⁴²:

1. **Return Value:** This field is used by the called procedure to return a value to the calling procedure⁴.
2. **Actual Parameters:** These are used by the calling procedures to supply parameters to the called procedures⁴.
3. **Control Link:** This points to the activation record of the caller⁴.

4. **Access Link:** This is used to refer to non-local data held in other activation records⁴.
5. **Saved Machine Status:** This holds the information about the status of the machine just before the procedure is called⁴. This information includes the value of the program counter and machine registers that have to be restored when control returns to the procedure⁴.
6. **Local Data:** This holds the data that is local to the execution of the procedure⁴.
7. **Temporaries:** This stores the values that arise in the evaluation of an expression⁴.

In the context of a runtime environment, activation records play a crucial role³. They allow the program to keep track of procedure calls and handle the execution flow³. For example, when a function calls another function, a new activation record is created and pushed onto the stack³. When the called function finishes execution, its activation record is popped from the stack, and control returns to the calling function³.

The access link in the activation record allows a function to access data that is not present in its local scope². For instance, if a function needs to access a global variable, it would use the access link to access that variable from the global scope².

Explain synthesized and inherited attributes used in syntax directed definitions with examples.

[Compiler Design | Syntax Directed Definition - GeeksforGeeks](#)

Sure, I'd be happy to explain synthesized and inherited attributes used in syntax directed definitions with examples.

Syntax Directed Definitions (SDD) is a kind of abstract specification. It is a generalization of context-free grammar in which each grammar production $X \rightarrow a$ is associated with a set of production rules of the form $s = f(b_1, b_2, \dots, b_k)$ where s is the attribute obtained from function f . [The attribute can be a string, number, type, or a memory location¹](#).

There are two types of attributes used in SDD:

1. **Synthesized Attributes:** These are those attributes which derive their values from their children nodes i.e., the value of a synthesized attribute at a node is computed from the values of attributes at children nodes in the parse tree¹. For example, consider the production $E \rightarrow E_1 + T \{ E.val = E_1.val + T.val \}$. In this, $E.val$ derives its values from $E_1.val$ and $T.val$ ¹. Synthesized attributes can be computed during a single bottom-up traversal of the parse tree².
2. **Inherited Attributes:** These are the attributes which derive their values from their parent or sibling nodes i.e., the value of inherited attributes are computed by the value of parent or sibling nodes¹. For example, consider the production $A \rightarrow BCD \{ C.in = A.in, C.type = B.type \}$. In this, $C.in$ and $C.type$ are inherited attributes¹. Inherited attributes can be computed during a single top-down and sideways traversal of the parse tree².

In the context of a runtime environment, these attributes play a crucial role. They allow the program to keep track of procedure calls and handle the execution flow². For example, when a function calls another function, a new activation record is created and pushed onto the stack. When the called function finishes execution, its activation record is popped from the stack, and control returns to the calling function².

The access link in the activation record allows a function to access data that is not present in its local scope². For instance, if a function needs to access a global variable, it would use the access link to access that variable from the global scope².

Discuss the different issues in design of Code Generator

1. **Input to the Code Generator:** The input to the code generator is the intermediate representation of the source code and the information in the symbol table¹². The code generation phase assumes that the input is free from all syntactic and semantic errors, and necessary type checking has taken place¹.
2. **Target Program:** The target program is the output of the code generator. The output can be absolute machine language, relocatable machine language, or assembly language¹². Each type of output has its own advantages and disadvantages¹².
3. **Memory Management:** Mapping the names in the source program to the addresses of data objects is done by the front end and the code generator¹². This involves managing local variables in the stack and global variables in the static area².
4. **Instruction Selection:** Selecting the best instructions will improve the efficiency of the program¹². It includes the instructions that should be complete and uniform. Instruction speeds and machine idioms also play a major role when efficiency is considered¹.
5. **Register Allocation:** Registers can be accessed faster than memory. The instructions involving operands in registers are shorter and faster than those involving memory operands². Register allocation involves selecting the set of variables that will reside in registers².
6. **Evaluation Order:** The efficiency of the target code can be affected by the order in which the computations are performed². Some computation orders need fewer registers to hold results of intermediate computations than others².

The design of a code generator should be done in such a way that it can be easily implemented, tested, and maintained¹. The code generator must always generate the correct code¹. It is essential because of the number of special cases that a code generator might face¹.

State various assembler directives and explain with example

Sure, assembler directives are special instructions that provide information to the assembler but do not generate any code¹. Here are some examples:

DB (Define Byte): This directive is used for the purpose of allocating and initializing single or multiple data bytes². For example:

PRICE DB 49h, 98h, 29h ; Declare an array of 3 bytes, named as PRICE and initialize.

NAME DB 'ABCDEF' ; Declare an array of 6 bytes and initialize with ASCII code for letters

1.

DW (Define Word): It is used for initializing single or multiple data words (16-bit)². For example:

MULTIPLIER DW 437Ah ; Declare a variable of type word and named it as MULTIPLIER. This variable is initialized with the value 437Ah when it is loaded into memory to run.

EXP1 DW 1234h, 3456h, 5678h ; Declare an array of 3 words and initialized with specified values.

2.

3. **END:** This directive is used at the end of a program to tell the assembler that this is the end of the program module².

EQU (Equate): It is used to assign any numerical value or constant to the variable². For example:

FACTOR EQU 03H ; Assign value 03H to the variable FACTOR

4.

5. **ORG (Origin):** This directive is used to assign a starting address for a module or segment².

MACRO and ENDM: These directives show the beginning and end of a macro along with defining name and parameters². For example:

MACRO1 MACRO &ARG1, &ARG2

...

MEND

6.

ASSUME: The ASSUME directive is used to tell the assembler that the name of the logical segment should be used for a specified segment³. For example:

ASSUME CS:CODE ; This tells the assembler that the logical segment named CODE contains the instruction statements for the program and should be treated as a code segment.

ASSUME DS:DATA ; This tells the assembler that for any instruction which refers to a data in the data segment, data will be found in the logical segment DATA.

7.

I hope this helps! Let me know if you have any other questions. 😊

Explain the use of NAMTAB, DEFTAB, and ARGTAB in the context of a SIC Macroprocessor, with the example of a nested macro

In the context of the SIC (Simplified Instructional Computer) Macro Processor, three primary data structures are used to manage macros: NAMTAB, DEFTAB, and ARGTAB. Let's explain their use with an example of a nested macro.

Data Structures in SIC Macro Processor

1. NAMTAB (Name Table):
 - NAMTAB stores the names of the macros defined in the program.
 - It maintains pointers to the beginning and the end of the corresponding macro definitions in DEFTAB.
2. DEFTAB (Definition Table):
 - DEFTAB holds the definitions of all macros, including their instructions.
 - It includes placeholders for parameters that will be replaced with actual arguments during macro expansion.
3. ARGTAB (Argument Table):
 - ARGTAB is used during macro expansion to store the actual arguments passed to the macro.
 - It facilitates the substitution of formal parameters with actual arguments in the macro body.

Example of Nested Macro

Consider the following example with nested macros:

```
MACRO
MACRO1 &ARG1, &ARG2
LDA &ARG1
MACRO2 &ARG2
ADD &ARG2
MEND
MACRO2 &ARG3
STA &ARG3
MEND
MEND
MACRO1 A, B
```

Explanation

1. Defining MACRO1 and MACRO2:
 - When MACRO1 and MACRO2 are defined, their names are entered into NAMTAB.
 - The definitions, including the nested macro MACRO2, are stored in DEFTAB with placeholders for &ARG1, &ARG2, and &ARG3.
2. NAMTAB:
 - NAMTAB will have entries for MACRO1 and MACRO2 with pointers to their respective definitions in DEFTAB.
 - For instance, MACRO1 might have a pointer to the start and end of its definition in DEFTAB, including the nested call to MACRO2.
3. DEFTAB:
 - DEFTAB will contain the actual instructions of the macros. It will store the instructions of MACRO1 and MACRO2 with placeholders.
 - For MACRO1:

```
LDA &ARG1  
ADD &ARG2 (calling MACRO2 within MACRO1)
```

For MACRO2:

```
ADD &ARG2  
STA &ARG3
```

1. ARGTAB during Expansion:
 - When MACRO1 A, B is invoked, ARGTAB is populated with the actual arguments: A for &ARG1 and B for &ARG2.
 - ARGTAB ensures that &ARG1 is replaced with A and &ARG2 with B during expansion.

Expansion Process

- During the expansion of MACRO1 A, B:
 - The macro processor looks up MACRO1 in NAMTAB to find its definition in DEFTAB.
 - It replaces &ARG1 with A and &ARG2 with B using ARGTAB.
 - When MACRO2 is encountered within MACRO1, the processor again uses NAMTAB to find MACRO2's definition and expands it, replacing &ARG3 appropriately

The expanded code will look like:

```
LDA A  
ADD B  
STA B
```

Here's a step-by-step breakdown:

1. Lookup MACRO1:
 - Find the start of MACRO1 in DEFTAB using NAMTAB.
 - Begin expansion, substituting &ARG1 and &ARG2 with A and B.
2. Encounter MACRO2 within MACRO1:

- Find MACRO2 in NAMTAB and expand it using the current value of &ARG2 as the actual argument.
- Substitute &ARG3 with B.

This example illustrates how NAMTAB, DEFTAB, and ARGTAB are used in a SIC Macro Processor to manage nested macros, ensuring proper parameter substitution and expansion.

Easy Answer:

Sure, I can explain the use of NAMTAB, DEFTAB, and ARGTAB in the context of a SIC Macroprocessor.

In a macroprocessor, these three data structures are used to handle macro definitions and expansions¹:

1. **NAMTAB (Name Table):** This table stores the names of the macros along with pointers to the beginning and end of the corresponding macro in DEFTAB². This allows the macroprocessor to quickly locate the definition of a macro when it encounters a macro invocation¹.
2. **DEFTAB (Definition Table):** This table stores the definitions and bodies of the macros². When a macro is defined, its body is stored in DEFTAB. References to macro instruction parameters within the macro body are converted to a positional notation².
3. **ARGTAB (Argument Table):** This table is used to store the arguments that are passed in a macro invocation². During macro expansion, the arguments stored in ARGTAB replace the corresponding parameters in the macro definition¹.

Let's consider an example of a nested macro in a SIC Macroprocessor:

MACRO1 MACRO &ARG1, &ARG2

...

MACRO2 &ARG1

...

MEND

MACRO2 MACRO &ARG3

...

LDA &ARG3

...

MEND

In this example, **MACRO1** is a macro that contains a call to another macro **MACRO2**. This is a nested macro. When **MACRO1** is invoked with certain arguments, these arguments are stored in ARGTAB. During the expansion of **MACRO1**, when the macroprocessor encounters the **MACRO2** invocation, it uses the arguments stored in ARGTAB to replace the parameters in the **MACRO2** definition.

Special Exam 2023

State various memory allocation strategies in compiler design.
Explain stack allocation strategy with examples.

In compiler design, memory allocation strategies are essential for managing the storage of variables and data structures during the execution of a program. The primary strategies include:

1. **Static Allocation:**

- Memory for variables is allocated at compile time.
- The size and location of data structures are fixed and known at compile time.
- No dynamic memory management at runtime.
- Suitable for global variables, constants, and fixed-size data structures.

2. **Stack Allocation:**

- Memory is allocated and deallocated in a last-in-first-out (LIFO) order.
- Used for local variables and function call management.
- Efficient and fast, but the size of variables must be known at compile time.
- Supports recursion by allocating a new stack frame for each function call.

3. **Heap Allocation:**

- Memory is allocated and deallocated dynamically at runtime.
- Suitable for data structures whose size may change or is not known at compile time (e.g., linked lists, dynamic arrays).
- Requires memory management techniques like garbage collection or manual deallocation to prevent memory leaks and fragmentation.

Stack Allocation Strategy

The stack allocation strategy is used primarily for managing the memory of local variables and function calls. It involves using a stack data structure where memory is allocated for variables when a function is called and deallocated when the function returns.

Key Components of Stack Allocation:

1. **Stack Frame (Activation Record):**

- Each function call creates a new stack frame, containing:
 - **Return Address:** The address to return to after the function execution completes.
 - **Function Parameters:** Arguments passed to the function.
 - **Local Variables:** Variables declared within the function.
 - **Saved Registers:** Registers that need to be preserved across function calls.
 - **Control Link (Dynamic Link):** Pointer to the previous stack frame.
 - **Access Link (Static Link):** Used in languages with nested functions to access variables from the enclosing scope.

2. **Stack Pointer (SP):**

- A register that points to the top of the stack. It is adjusted as data is pushed to or popped from the stack.

3. **Frame Pointer (FP):**

- A register that points to the base of the current stack frame, providing a stable reference point for accessing function parameters and local variables.

Stack Allocation Process:

1. **Function Call:**

- A new stack frame is created and pushed onto the stack.
- The return address and parameters are stored in the stack frame.
- The current frame pointer is saved, and a new frame pointer is set to the base of the new stack frame.
- Local variables are allocated space in the stack frame.

2. **Function Execution:**

- The function executes, using the stack frame for accessing its parameters and local variables.
- If the function calls another function, the process repeats, and a new stack frame is created.

3. **Function Return:**

- The stack frame is popped off the stack.
- The stack pointer is adjusted to remove the space allocated for the stack frame.
- The return address is used to transfer control back to the caller function.
- The previous frame pointer is restored.

Example of Stack Allocation

Consider a simple program with nested function calls:

```
```c
#include <stdio.h>

void bar(int x) {
 int y = x + 1;
 printf("y = %d\n", y);
}

void foo() {
 int a = 10;
 bar(a);
}

int main() {
```

```
foo();
return 0;
}
...
```

#### #### Step-by-Step Stack Allocation:

##### 1. \*\*Initial State:\*\*

- The stack is empty before `main` is called.

##### 2. \*\*Calling `main`:\*\*

- A stack frame for `main` is created.
- The return address for exiting the program is stored.
- Stack pointer (SP) is adjusted.

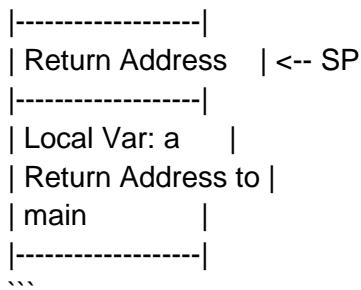
```



3. **Calling `foo` from `main`:**

- A new stack frame for `foo` is created.
- The return address to `main` is stored.
- Space for local variable `a` is allocated.

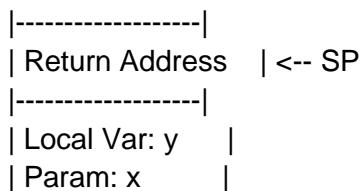
```

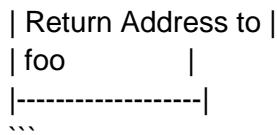


##### 4. \*\*Calling `bar` from `foo`:\*\*

- A new stack frame for `bar` is created.
- The return address to `foo` is stored.
- Parameter `x` is stored.
- Space for local variable `y` is allocated.

```

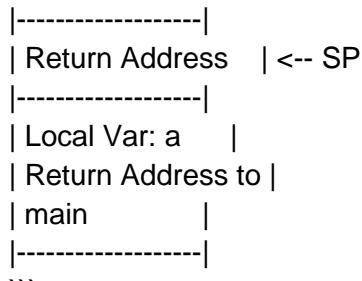




5. **Returning from `bar` to `foo`:**

- The stack frame for `bar` is popped off.
- SP is adjusted to the top of the `foo` stack frame.
- Control returns to `foo`.

``



6. **Returning from `foo` to `main`:**

- The stack frame for `foo` is popped off.
- SP is adjusted to the top of the `main` stack frame.
- Control returns to `main`.

``



7. **Exiting `main`:**

- The stack frame for `main` is popped off.
- The program exits.

Advantages of Stack Allocation

- **Efficiency:** Fast allocation and deallocation of memory using simple pointer operations.
- **Automatic Memory Management:** Local variables are automatically deallocated when a function exits, reducing memory leaks.
- **Support for Recursion:** Each function call, including recursive calls, gets its own stack frame, maintaining separate contexts.

Disadvantages of Stack Allocation

- **Fixed Size:** Stack size is usually fixed, leading to stack overflow in cases of deep recursion or large local variables.

- ****Limited Lifetime:**** Variables allocated on the stack only exist within the function's execution context and are not suitable for data that needs to persist beyond the function call.
- ****No Random Access:**** Access to stack frames is LIFO, meaning only the most recent function's variables can be directly accessed.

In conclusion, stack allocation is a fundamental memory management strategy used in many programming languages and systems, providing efficient, automatic management of function calls and local variables. Its design supports recursion and structured programming but comes with limitations related to fixed stack size and variable lifetime.

Explain conditional macro expansion in the context of the SIC Macro processor along with an example

Sure, I can explain conditional macro expansion in the context of the SIC Macro processor along with an example, and also explain two assembler directives related to it.

Conditional Macro Expansion: Conditional Macro Expansion allows the macro processor to modify the sequence of statements generated for a macro expansion, depending on the arguments supplied in the macro invocation¹. This provides great power and flexibility¹. For example, consider the following macro definition:

```
%MACRO CHECK_ZERO &ARG
    LDA &ARG
    JEQ ZERO
    ...
ZERO ...
%MEND
```

In this example, the macro **CHECK_ZERO** checks if the argument **&ARG** is zero. If **&ARG** is zero, the jump to label **ZERO** is executed.

Assembler Directives: Assembler directives are instructions that control the operation of the assembler². Here are two assembler directives related to macro processing in the SIC Macro processor:

1. **MACRO:** This directive identifies the beginning of a macro definition³⁴. The macro name and parameters define a pattern or prototype for the macro³⁴. In SIC/XE, each parameter begins with the character **&**, which facilitates the substitution of parameters during macro expansion³⁴.
2. **MEND:** This directive identifies the end of a macro definition³⁴. After the **MEND** directive, the macro processor knows that it has reached the end of the current macro definition³⁴.

I hope this helps! Let me know if you have any other questions. 😊

Heap Allocation Vs Stack Allocation

[Stack vs Heap Memory Allocation - GeeksforGeeks](#)

Analysis and Synthesis Phase of a compiler

[Analysis synthesis model of compilation - EasyExamNotes.com](#)

[Phases of a Compiler - GeeksforGeeks](#)

explain the two methods of code optimization: Common-sub expression elimination and Copy propagation.

Common-sub expression elimination: This is a technique used to optimize codes. It works by computing the value of the subexpression and assigning the value to a variable. Now, the initial common subexpression is replaced by that variable. [It helps in reducing the number of repeated computations¹](#). For example:

```
int a = 10;
int b = a + 1 * 2;
int c = a + 1 * 2; // 'c' has common expression as 'b'
int d = c + a;
```

After elimination:

```
int a = 10;
int b = a + 1 * 2;
int d = b + a;
```

1. In this example, the common subexpression `a + 1 * 2` is computed once and its value is stored in `b`. Then, `b` is used in place of the common subexpression in the calculation of `d`.

Copy propagation: Copy propagation is an optimization technique used in compiler design.

[It is used to replace the occurrence of target variables that are the direct assignments with their values²](#). For example:

```
int a = 5;
int b = a;
int c = b + 2;
```

After copy propagation:

```
int a = 5;
int c = a + 2;
```

2. In this example, the variable `b` is a direct assignment of `a`. So, in the calculation of `c`, `b` is replaced with `a`.

Explain the data structures involved in the SIC/XE macroprocessor.

In a SIC/XE macroprocessor, three main data structures are involved¹²³:

1. **DEFTAB (Definition Table):** This table stores the definitions and bodies of the macros¹²³. When a macro is defined, its body is stored in DEFTAB. References to macro instruction parameters within the macro body are converted to a positional notation¹²³.
2. **NAMTAB (Name Table):** This table stores the names of the macros along with pointers to the beginning and end of the corresponding macro in DEFTAB¹²³. This allows the macroprocessor to quickly locate the definition of a macro when it encounters a macro invocation¹²³.
3. **ARGTAB (Argument Table):** This table is used to store the arguments that are passed in a macro invocation¹²³. During macro expansion, the arguments stored in ARGTAB replace the corresponding parameters in the macro definition¹²³.

These data structures work together to enable the macroprocessor to efficiently process macro definitions and expansions¹²³.

Explain how unique labels are generated during macro expansions in the context of the SIC/XE macro processor.

During macro expansion, unique labels are typically generated using a combination of **predefined prefixes and a counter**¹. These labels are used to ensure that each instance of a macro expansion has its own unique set of labels, preventing any conflicts or errors that could arise from duplicate labels¹.

In the SIC/XE macro processor, many processors create special labels within macro instructions to solve the labeling problem². Each symbol begins with a special character (for example, \$), and is modified by a two-character alphanumeric counter². For example, the first expansion might use \$AA, and the succeeding ones might use \$AB, \$AC, etc².

This technique ensures that each macro expansion has its own unique set of labels, even when the same macro is expanded multiple times within the same program².

Describe the structure of small lex program with example.

Sure, I can help with that. [A Lex program, used in compiler design, has a specific structure that consists of three main parts](#)¹²³⁴:

1. **Declarations:** This section includes the declarations of variables¹².
2. **Translation rules:** These rules consist of a pattern and an action¹².
3. **Auxiliary procedures:** This section holds auxiliary functions used in the actions¹².

[Here is a simple example of a Lex program that recognizes numbers](#)²:

```

%{
#include <stdio.h>
%}

%%

[0-9]+ {
    printf("Saw an integer: %s\n", yytext);
}

.\n { ; }

%%

main() {
    printf("Enter some input that consists of an integer number\n");
    yylex();
}

int yywrap() {
    return 1;
}

```

In this program:

- The **Declarations** section includes the `stdio.h` header file.

- The **Translation rules** section has a rule for recognizing one or more digits (`[0-9]+`) and printing them. It also has a rule for recognizing any other character or newline and doing nothing.
- The **Auxiliary procedures** section includes the `main` function, which calls `yylex` to start the lexer, and the `yywrap` function, which returns 1 when the end of the input is reached.

Re-Exam Jan 2019

Phases of a compiler

[Phases of a Compiler - GeeksforGeeks](#)

five distinguishing points between compilers and interpreters:

1. **Execution Time:** Compilers translate the entire program before execution, while interpreters translate one line at a time during execution.
2. **Output:** Compilers generate an intermediary or machine code file, whereas interpreters do not produce any output files.
3. **Error Handling:** Compilers display all errors after translation, making it less interactive. Interpreters display errors line by line, making it more interactive.
4. **Speed:** Programs compiled using compilers generally run faster because they are fully translated before execution. Interpreted programs might be slower as they are translated during execution.
5. **Platform Dependency:** Compiled programs can be platform-dependent if machine code is generated; interpreted programs are generally platform-independent.

Directed Acyclic Graph with example

[Directed Acyclic graph in Compiler Design \(with examples\) - GeeksforGeeks](#)

Operator Precedence Parser

[Role of Operator Precedence Parser - GeeksforGeeks](#)

Explain the use of assembler . Explain the use of POT and MOT and it's uses

An **Assembler** is a program that translates an assembly language program into a machine language program¹²³. It takes basic computer commands and converts them into binary code that a computer's processor can use to perform its basic operations². Assemblers are used to create executables that can be run on a computer⁴. They are commonly used to write operating system kernels, device drivers, and other low-level code that requires direct control over the hardware⁴.

The structure of an Assembler involves two tables: the **POT (Pseudo-Operation Table)** and the **MOT (Machine Operation Table)**¹.

1. **POT (Pseudo-Operation Table)**: This table is used to process pseudo-operations. Pseudo-operations are non-executable and provide instructions to the assembler for certain tasks, such as defining constants, organizing data, and specifying the format of the output¹.
2. **MOT (Machine Operation Table)**: This table is used to store all machine-opcodes with their symbolic code, their length, and their bit configuration¹⁵. It is used to accept the instructions and convert/gives its binary opcode⁵.

Use of Macro Processor and use of AIF and AGO

A Macro-processor is a program that replaces each macro instruction with the corresponding group of source language statements¹²³. This is known as the expansion of macros¹. Macro-processors are used in machine assemblers, high-level programming languages, and operating system command languages⁴.

The AIF (Assembler If) and AGO (Assembler Go) are two important directives used in macro-processors⁵⁶.

1. **AIF (Assembler If)**: This directive is used to make decisions during assembly. It's used to test a condition and if the condition is true, the assembler will execute the statement following the AIF directive⁵.
2. **AGO (Assembler Go)**: This directive is used for unconditional branching. It alters the sequence in which your assembler language statements are processed⁶. This provides you with final exits from conditional assembly loops⁶.

Here is a simple example of a macro that uses AIF and AGO⁶:

MACRO &NAME

MOVE &T,&F

AIF (T'&T EQ 'F').FIRST

Statement 1

AGO .END

Statement 2

.FIRST

AIF (T'&T NE T'&F).END

Statement 3

&NAME ST 2,SAVEAREA

L 2,&F

ST 2,&T

L 2,SAVEAREA

.END

MEND

In this example:

- The **AIF** directive is used to check if the type attribute of the first macro instruction operand is the letter 'F'. If it is, it branches to the label **.FIRST**⁶.
- The **AGO** directive is used to branch unconditionally to the label **.END**⁶.

I hope this helps! Let me know if you have any other questions. 😊

Pending Sums

ESE May 2023

| | |
|------|---|
| Q3 A | for the boolean expression, A < B OR C < D AND P < Q using the translation scheme for backpatching of Boolean expression,
1. draw an annotated parse tree with the true and false lists for each subexpression.
2. Also generate the 3AC , assuming that the address of the first instruction generated is 100. |
|------|---|

ESE May 2022

Q3. B.

Give 3 address code for following code fragment

```
if( ( a < b )
{
    while( c > d )
        x = x + y
}
else
    do
{
    p = p + q;
}
while(e <= f);
```

done

ESE May 2019

Generate an assembly code for the following intermediate code snippet using code generation algorithm:

$d = (a - b) + (a - c) + (a - c)$ Register R0, R1 are the only registers available. Assume ST, MOV, ADD and SUB are the instruction op-codes which have two operands only. For e.g. Mov R0,a means that memory address of a(source) is copied in register R0(destination). Assumptions for the other instruction opcodes can be made similarly. Clearly show the contents of different descriptors after each conversion of three address instruction to assembly instruction.

How syntax tree is created for $4 \cdot b + c$? Explain with the help of Syntax Directed Definition.

Generate three address code and a quadruple, triple of a generated three-address code in static single assignment form for the expression $a[i] = b[i + 1] + (k * 2)$.

ESE May/ June 2018

What do you mean by three address code.

Generate three address code for given expression

while (a < b) do

if (c < d) then

x = y + z

else

x = y + z

done

Write a syntax directed definition that generates three address code for Booleans.

ESE 2018 April/May

Draw the diagram of phases of compiler. Illustrate the output after each phase of the compiler for the following statement:

$$d=b^2-4*a*c$$

ESE May 2023

1. List the 2 benefits of using machine independent intermediate form (ICG)
2. Represent Indexed assignments of the form $x = y[i]$ and $x[i]=y$ using triples
3. consider the following statement

```
while ( a < b )
{
    if ( c < d )
        x = y + z
    else
        x = y - z
}
```

Generate 3 AC for the above statement done

done

What do you mean by backpatching ?

for the boolean expression $a==b \&& (c==d || e==f)$ Using the translation scheme for Boolean expression,

1. draw an annotated parse tree with the true and false lists for each subexpression.
2. Also generate the 3AC , assuming that the address of the first instruction generated is 100.