Ans1) (b)

Feature	Linker	Loader
Purpose	Combines object files into an executable file.	Loads an executable file into memory and prepares it for execution.
When used	After the compiler has finished compiling the source code.	Before the program is executed.
What it does	Resolves any references between object files and creates an executable file.	Maps the executable file into memory and initializes the processor.

(c)

Feature	Symbol Table	Data Structure
Purpose	Stores information about the entities in a program.	Stores data in general.
Data type	Symbols.	Data of any type.
Organization	Hierarchical.	Linear or non-linear.
Access	Sequential or random.	Sequential or random.
Implementation	Typically implemented as a hash table or a linked list.	Can be implemented using any data structure.

(d)

```
t1 = b + c

t2 = -t1

a = a * t2
```

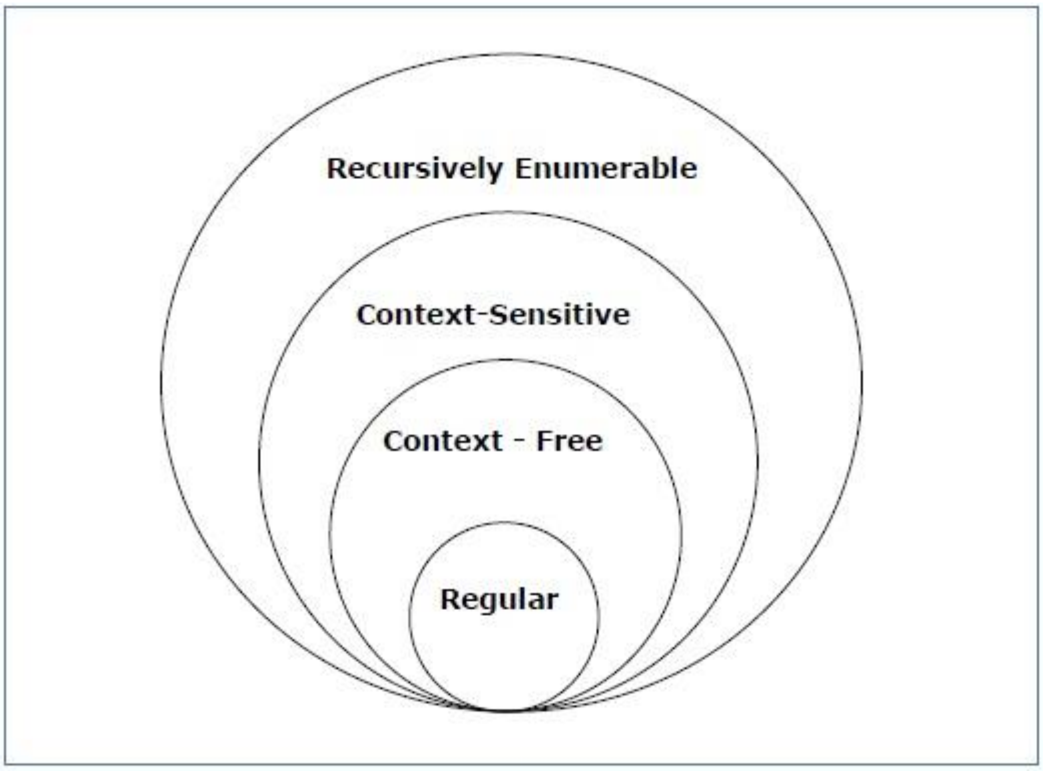
(e)

- **Single pass compiler:** A single pass compiler is a type of compiler that reads the source code only once and generates the target code in a single pass. This type of compiler is typically faster than a multi-pass compiler, but it is less flexible and cannot perform some optimizations that multi-pass compilers can.
- **Multi pass compiler:** A multi-pass compiler is a type of compiler that reads the source code multiple times and generates the target code in multiple passes. This type of compiler is typically slower than a single pass compiler, but it is more flexible and can perform more optimizations.

Single pass compilers are better for simple applications that do not require a lot of flexibility or optimization. **Multi pass compilers are better** for complex applications that require a lot of flexibility and optimization.

(f)

Grammar Type	Grammar Accepted	Language Accepted	Automaton
Type 0	Unrestricted grammar	Recursively enumerable language	Turing Machine
Type 1	Context-sensitive grammar	Context-sensitive language	Linear-bounded automaton
Type 2	Context-free grammar	Context-free language	Pushdown automaton
Type 3	Regular grammar	Regular language	Finite state automaton



(g)

- **Left factoring can introduce ambiguity.** Ambiguity occurs when a grammar can generate two or more different parse trees for the same input string. This can make it difficult for a compiler or interpreter to determine the correct meaning of a program.
- **Left factoring can make grammars more complex.** When a grammar is left factored, the number of productions increases. This can make it more difficult to understand and maintain the grammar.
- **Left factoring can reduce the efficiency of parsers.** Parsers that use left factoring must often backtrack, which can slow down the parsing process.

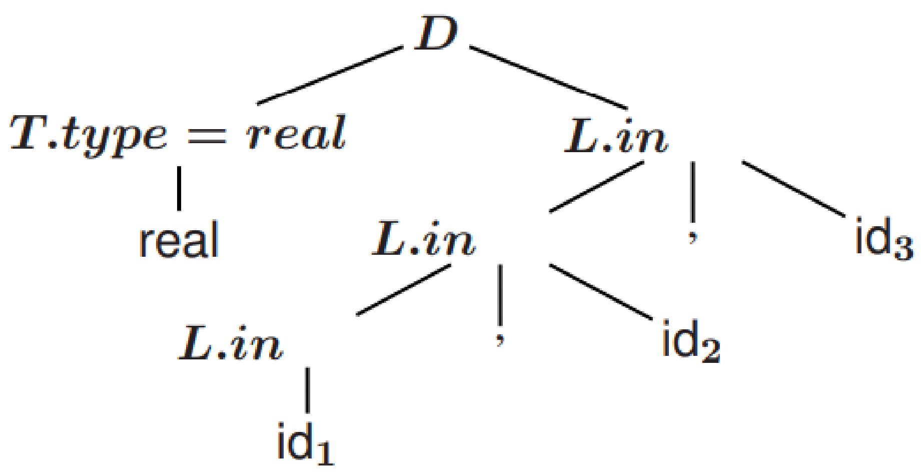
(h)

There are two main approaches to register allocation:

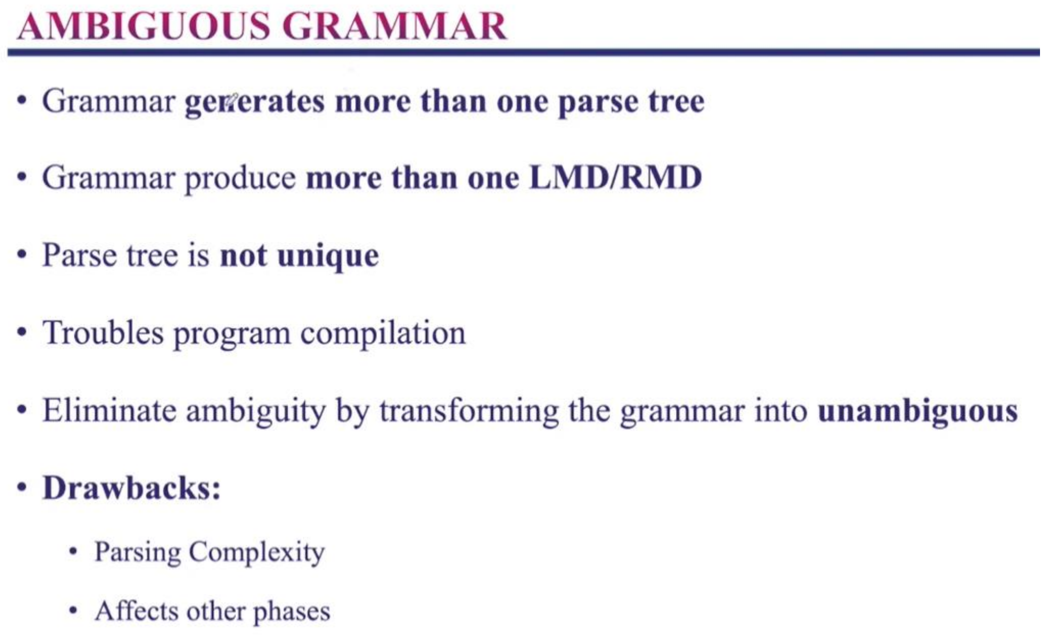
- **Local register allocation:** This approach allocates registers to variables and expressions within a single basic block. A basic block is a sequence of instructions that cannot be interrupted by any jumps or calls.
- **Global register allocation:** This approach allocates registers to variables and expressions across multiple basic blocks. This approach is more complex than local register allocation, but it can often produce better results.

(i)

The annotated parse-tree for the input real id_1, id_2, id_3 is:

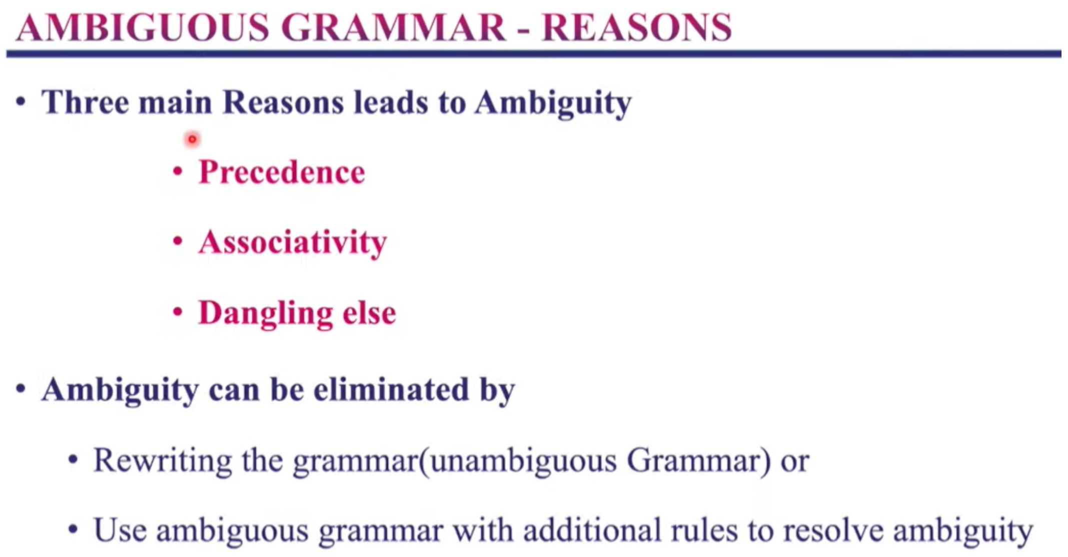


(j)



AMBIGUOUS GRAMMAR

- Grammar **generates more than one parse tree**
- Grammar produce **more than one LMD/RMD**
- Parse tree is **not unique**
- Troubles program compilation
- Eliminate ambiguity by transforming the grammar into **unambiguous**
- **Drawbacks:**
  - Parsing Complexity
  - Affects other phases



AMBIGUOUS GRAMMAR - REASONS

- **Three main Reasons leads to Ambiguity**
  - **Precedence**
  - **Associativity**
  - **Dangling else**
- **Ambiguity can be eliminated by**
  - Rewriting the grammar(unambiguous Grammar) or
  - Use ambiguous grammar with additional rules to resolve ambiguity

Ans2) (a)

1. **Lexical analysis:** This phase converts the source code into a sequence of tokens. A token is a basic unit of the source code, such as a keyword, identifier, operator, or constant.
2. **Syntax analysis:** This phase checks the source code for syntax errors. Syntax errors are errors in the structure of the source code, such as missing keywords, unmatched parentheses, or incorrect punctuation.
3. **Semantic analysis:** This phase checks the source code for semantic errors. Semantic errors are errors in the meaning of the source code, such as using a variable that has not been declared or calling a function that does not exist.
4. **Intermediate code generation:** This phase generates an intermediate representation of the source code. The intermediate representation is a machine-independent format that can be used for code optimization and code generation.

5. **Code optimization:** This phase improves the performance of the generated code. Code optimization can be done by removing unnecessary instructions, rearranging instructions for better performance, or using more efficient algorithms.
6. **Code generation:** This phase generates the machine code for the target architecture. The machine code is the actual instructions that will be executed by the target machine.

Ans3) (a)

- **Lex:** A lexical analyzer, also known as a lexer, is a program that reads a stream of characters and converts it into a sequence of tokens.
- **Lexeme:** A lexeme is a sequence of characters that is recognized by the lexical analyzer as a single unit.
- **Lexical analyzed:** A lexical analyzed program is a program that has been converted into a sequence of tokens by the lexical analyzer.
- **Token:** A token is a basic unit of a programming language, such as a keyword, identifier, operator, or constant.

Roles of lexical analyser are:

- **Tokenization:** The lexical analyzer breaks the source code into a sequence of tokens. A token is a basic unit of the source code, such as a keyword, identifier, operator, or constant.
- **Comment removal:** The lexical analyzer removes comments from the source code. Comments are not part of the program's syntax and are ignored by the compiler.
- **White space removal:** The lexical analyzer removes white space from the source code. White space includes characters such as spaces, tabs, and newlines.
- **Error detection:** The lexical analyzer detects errors in the source code. Errors such as misspelled keywords, invalid identifiers, and unmatched parentheses can be detected by the lexical analyzer.

Ans6) (a)

Type checking is a process of verifying that the types of variables, expressions, and statements are consistent with the programming language's type system.

There are two main types of type checking: static type checking and dynamic type checking.

- **Static type checking** is performed during compilation. The compiler analyzes the program's source code and checks the types of all variables, expressions, and statements. If the compiler finds any type errors, it will report an error to the user.
- **Dynamic type checking** is performed at runtime. The program is executed and the types of variables, expressions, and statements are checked at runtime. If the program tries to perform an operation that is not allowed for the types of the involved variables, an error will be raised.

Type checking is **performed** using a type system. A **type system** is a set of rules that define the types of variables, expressions, and statements. The type system is used by the compiler to verify that the program is well-typed.

(b)

- **Heap:** The heap is a region of memory that is dynamically allocated during program execution. It is used to store data that is not known at compile time, such as the results of function calls and the contents of dynamically allocated arrays.
- **Dynamic storage allocation:** Dynamic storage allocation is the process of allocating memory on the heap. It is performed by the runtime system, which is a part of the operating system.

  There are two main techniques for dynamic storage allocation:

  - **Allocating memory from a free list:** A free list is a list of blocks of memory that are available for allocation. When the runtime system needs to allocate memory, it removes a block from the free list.
  - **Using a garbage collector:** A garbage collector is a program that automatically manages memory allocation and deallocation. It tracks which blocks of memory are in use and which blocks are free. When a block of memory is no longer in use, the garbage collector deallocates it.
- **Synthesized attributes:** Synthesized attributes are attributes that are computed by the compiler from other attributes. For example, the size of an array is a synthesized attribute that is computed from the size of the array elements and the number of elements in the array.

Ans7) (a)

An activation record (also known as a stack frame) is a data structure that is used to store the state of a function when it is called. It contains information such as the function's arguments, local variables, and return address.

The task of dividing the task between the calling and called program is called **calling convention**. Calling convention ensures that the calling and called programs can communicate with each other effectively.

Calling convention specifies the following:

- How arguments are passed to the called function.
- Where local variables are stored in the activation record.
- How the return value is returned from the called function.

Some calling conventions are:

- **Conventional calling convention:** This is the most common calling convention. In this convention, arguments are passed to the called function on the stack. The return value is returned in the EAX register on the x86 architecture.
- **Register calling convention:** In this convention, arguments are passed to the called function in registers. The return value is returned in a register, usually EAX.
- **Mixed calling convention:** This is a hybrid of the conventional and register calling conventions. Some arguments are passed on the stack, while others are passed in registers.

(b)

1. Call by Value (CBV): In this method, the value of the actual parameter is copied into the formal parameter. Any modifications made to the formal parameter within the function do not affect the value of the actual parameter.

Example:

```
procedure swap(a, b: integer);
    var temp: integer;
begin
    temp := a;
    a := b;
    b := temp;
end;

var x, y: integer;
x := 5;
y := 10;
swap(x, y);
// After the swap function is called, the values of x and y remain unchanged.
```

2. Call by Name (CBN): In this method, the name of the actual parameter is substituted directly into the function body. The function evaluates the argument expression each time it is referenced.

Example:

```
function max(a, b: integer): integer;
begin
    if a > b then
        return a;
    else
        return b;
end;
var x, y: integer;
x := 5;
y := 10;
max(x, y + 2);
// In this case, the expression 'y + 2' is evaluated each time it is referenced in the max function.
```

3. Call by Result (CBR): In this method, the formal parameter is evaluated first, and then the value is copied back to the actual parameter at the end of the function. Any modifications made to the formal parameter within the function are reflected in the actual parameter.

Example:

```
procedure increment(var n: integer);
begin
   n := n + 1;
end;

var x: integer;
x := 5;
increment(x);
// After the increment function is called, the value of x is modified and becomes 6.
```

Ans8) (a)

Peephole optimization is a compiler optimization technique that involves searching for small sequences of instructions in the generated code and replacing them with more efficient sequences.

Some examples of peephole optimizations:

- **Redundant instructions:** Peephole optimization can remove redundant instructions. For example, if a code sequence contains two instructions that do the same thing, the compiler can remove one of the instructions.
- **Commutative instructions:** Peephole optimization can also combine commutative instructions. For example, the instructions `a = b + c` and `b = c + a` are commutative, meaning that the order of the operands does not matter. The compiler can combine these two instructions into a single instruction `a = b + c + b`.
- **Redundant loads and stores:** Peephole optimization can also remove redundant loads and stores. For example, if a code sequence contains two instructions that load the same value from memory, the compiler can remove one of the loads. Similarly, if a code sequence contains two instructions that store the same value to memory, the compiler can remove one of the stores.

Ans9) (a)

- **Dead code elimination:** This technique removes code that is never executed. This can be done by analyzing the control flow of the program.
- **Constant folding:** This technique replaces expressions that contain constants with their values. This can improve performance by eliminating the need to evaluate the expressions at runtime.
- **Common subexpression elimination:** This technique removes common subexpressions from the code. This can improve performance by eliminating the need to evaluate the subexpressions multiple times.

- **Copy propagation:** This technique propagates copies of values through the code. This can improve performance by eliminating the need to load and store values multiple times.
- **Loop invariant hoisting:** This technique moves loop invariants out of loops. This can improve performance by reducing the number of times that the invariants are evaluated.
- **Loop unrolling:** This technique breaks loops into smaller loops. This can improve performance by reducing the number of conditional branches in the code.
- **Function inlining:** This technique replaces calls to functions with the body of the functions. This can improve performance by reducing the number of function calls.
- **Interprocedural optimization:** This technique performs optimizations across multiple functions. This can improve performance by finding optimizations that are not possible within a single function.

(b)

- **Register allocation:** Register allocation is the process of assigning registers to variables in the generated code. This is a challenging problem because there are often not enough registers to store all of the variables, and the compiler must choose which variables to store in registers and which to store in memory.
- **Instruction selection:** Instruction selection is the process of choosing the best instructions to implement each operation in the generated code. This is a challenging problem because there are often several different instructions that can implement the same operation, and the compiler must choose the instructions that will perform the operation the fastest.
- **Memory management:** Memory management is the process of allocating and deallocating memory for variables in the generated code. This is a challenging problem because the compiler must ensure that memory is allocated and deallocated correctly, and that memory is not leaked.
- **Code size:** The code size of the generated code is important because it can affect the performance of the program. The compiler must try to generate code that is as small as possible without sacrificing performance.
- **Code readability:** The readability of the generated code is important because it can affect the maintainability of the program. The compiler must try to generate code that is easy to read and understand.