Peephole Optimization

Peephole optimization: It is performed on a very small set of instructions in a segment of code. A part of code is replaced by shorter and faster code without a change in output. The peephole is machine-dependent optimization.

The objectives of peephole optimization are:

1. To improve performance
2. To reduce memory footprint
3. To reduce code size

A. Redundant load and store elimination: In this technique, redundancy is eliminated.

```
Initial code:
y = x + 5;
i = y;
z = i;
w = z * 3;

Optimized code:
y = x + 5;
w = y * 3; //* there is no i now

//* We've removed two redundant variables i & z whose value were just being
copied from one another.
```

B. Constant folding: The code that can be simplified by the user itself, is simplified. Here simplification to be done at runtime are replaced with simplified code to avoid additional computation.

```
Initial code:
x = 2 * 3;

Optimized code:
x = 6;
```

C. Strength Reduction: The operators that consume higher execution time are replaced by the operators consuming less execution time.

```
Initial code:
y = x * 2;

Optimized code:
y = x + x;    or     y = x << 1;

Initial code:
y = x / 2;

Optimized code:
y = x >> 1;
```

D. Null sequences/ Simplify Algebraic Expressions : Useless operations are deleted.

```
a := a + 0;

a := a * 1;
```

```
a := a/1;

a := a - 0;
```

E. Combine operations: Several operations are replaced by a single equivalent operation.
F. Dead code Elimination: A part of the code which can never be executed, eliminating it will improve processing time and reduces set of instruction.

Difficulties in Lexical Analysis

- **Ambiguity:** The same sequence of characters can be interpreted in multiple ways. For example, "while" can be a keyword or a two-word identifier.
- **Whitespace:** Whitespace characters can be ignored, which can make it difficult to identify the boundaries of tokens.
- **Comments:** Comments are typically ignored by the lexical analyzer, but it can be difficult to distinguish them from tokens.
- **Keywords:** Reserved words that have a special meaning in the programming language.
- **Identifiers:** Names that are used to refer to variables, functions, and other entities in the program.
- **Literals:** Constants that have a fixed value.

L and S Attributes

L Attributes	S Attributes
Uses both synthesized attributes and inherited attributes with a restriction that inherited attribute can inherit values from left siblings only	Uses only synthesized attributes
Evaluated by depth-first and left-to-right parsing manner.	Evaluated in bottom-up parsing
Semantic actions are placed anywhere in RHS.	Semantic actions are placed in rightmost place of RHS.
Example: Type of an expression	Example: Value of an identifier

SDD

Grammar + Semantic rules = SDD (Syntax Directed Definition)

SDDs are used in compilers to implement semantic analysis. Semantic analysis is the process of determining the meaning of a program construct. It is used to check for errors, generate code, and perform other tasks.

SDDs are written in **attribute grammars**. Attribute grammars are a formal notation for specifying the semantics of a programming language.

There are two types of SDDs:

L Attributes	S Attributes
Uses both synthesized attributes and inherited attributes with a restriction that inherited attribute can inherit values from left siblings only	Uses only synthesized attributes
Evaluated by depth-first and left-to-right parsing manner.	Evaluated in bottom-up parsing
Semantic actions are placed anywhere in RHS.	Semantic actions are placed in rightmost place of RHS.
Example: Type of an expression	Example: Value of an identifier

Steps involved in attribute evaluation:

1. **Initialization:** The attributes of all nodes in the parse tree are initialized to their default values.
2. **Evaluation:** The attributes of each node in the parse tree are evaluated using the SDD.
3. **Error detection:** If any of the attributes in the parse tree cannot be evaluated, an error is reported.
4. **Termination:** The attribute evaluation process terminates when all of the attributes in the parse tree have been evaluated.

Token Generator and Token Recognizer

1. Token Generator (Scanner):

The token generator, also known as a lexer or scanner, is responsible for scanning the input source code and breaking it into tokens. It performs lexical analysis by identifying and classifying the individual tokens based on predefined rules or patterns.

2. Token Recognizer(Parser):

The token recognizer, also known as a parser or syntax analyzer, receives the tokens generated by the lexer and checks whether they conform to the grammar rules of the programming language. It constructs a syntax tree or performs further analysis based on the sequence and structure of the tokens.

Example:

Consider a simple arithmetic expression: 2 * (3 + 4). The scanner would tokenize this expression into the following sequence of tokens:

TOKEN_INTEGER_LITERAL(2)

TOKEN_MULTIPLICATION_OPERATOR(*)

TOKEN_OPEN_PARENTHESIS("(")

TOKEN_INTEGER_LITERAL(3)

TOKEN_ADDITION_OPERATOR(+)

TOKEN_INTEGER_LITERAL(4)

TOKEN_CLOSE_PARENTHESIS(")")

Token Recognizer would use the tokens provided by the scanner to build a parse tree

```
    *
  /   \
 2      +
      /   \
     3      4
```

Lex Program for $a^n b^n$

Lexical Analyzer Source Code :

```
%{
        /* Definition section */
        #include "y.tab.h"
%}
/* Rule Section */
%%
        [aA] {return A;}
        [bB] {return B;}
        \n {return NL;}
        . {return yytext[0];}
%%
int yywrap()
{
        return 1;
}
```

Parser Source Code :

```
%{
        /* Definition section */
        #include<stdio.h>
        #include<stdlib.h>
%}
%token A B NL
/* Rule Section */
%%
stmt: S NL {
printf("valid string\n");
exit(0);
}
;
S: A S B |
;

%%
int yyerror(char *msg)
{
        printf("invalid string\n");
        exit(0);
}
//driver code
main()
{
        printf("enter the string\n");
        yyparse();
}
```

Key	Top Down Parsing	Bottom Up Parsing
Strategy	Top-down approach starts evaluating the parse tree from the top and move downwards for parsing other nodes.	Bottom-up approach starts evaluating the parse tree from the lowest level of the tree and move upwards for parsing the node.
Attempt	Top-down parsing attempts to find the left most derivation for a given string.	Bottom-up parsing attempts to reduce the input string to first symbol of the grammar.
Derivation Type	Top-down parsing uses leftmost derivation.	Bottom-up parsing uses the rightmost derivation.
Objective	Top-down parsing searches for a production rule to be used to construct a string.	Bottom-up parsing searches for a production rule to be used to reduce a string to get a starting symbol of grammar.

Feature	Pass1	Pass2
Purpose	Scans the input code and creates a symbol table	Analyzes the input code and generates the machine code
What it does	Reads the source code line by line and creates a symbol table for each variable and function.	Reads the symbol table and generates the machine code for each statement in the source code.
When it happens	First pass	Second pass
Complexity	Less complex	More complex
Efficiency	More efficient	Less efficient
Applications	Used for compilers that target simple languages	Used for compilers that target complex languages

Feature	Inherited attributes	Synthesized attributes
Determined by	Attributes of the parent node	Attributes of the child nodes
Example	Type of an expression	Value of an expression
When it is evaluated	During the parse tree traversal	During the code generation
Used for	Semantic analysis	Code generation

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.

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.

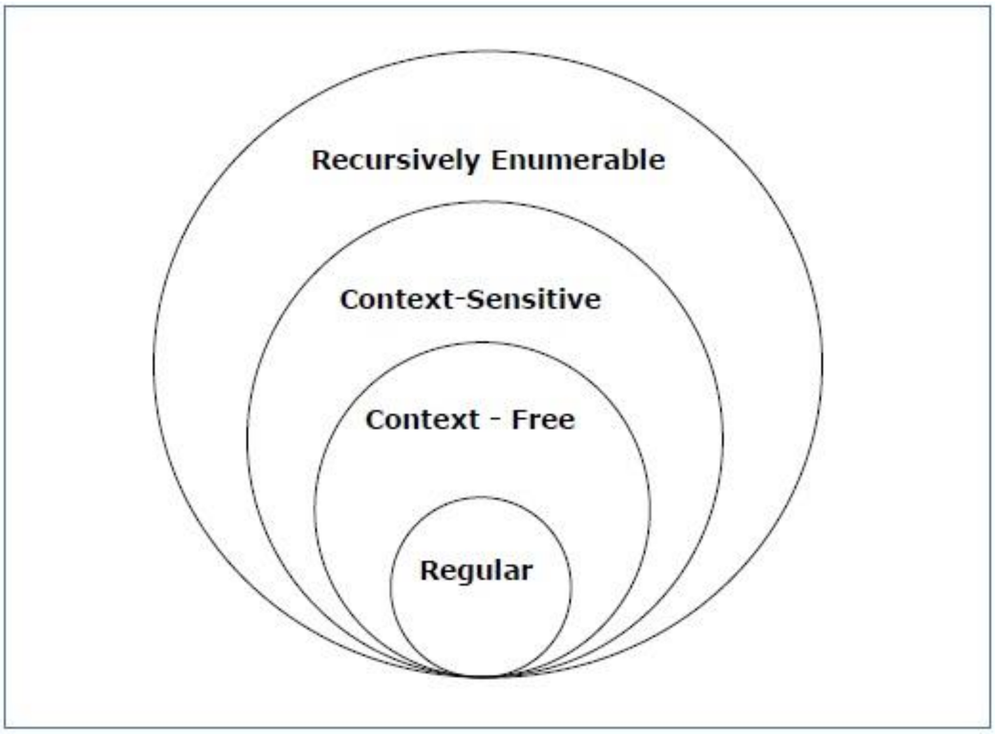
Single Pass vs Multi Pass Compiler

- **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.

Chomsky Classification of Grammar

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



Problems of Left Factoring

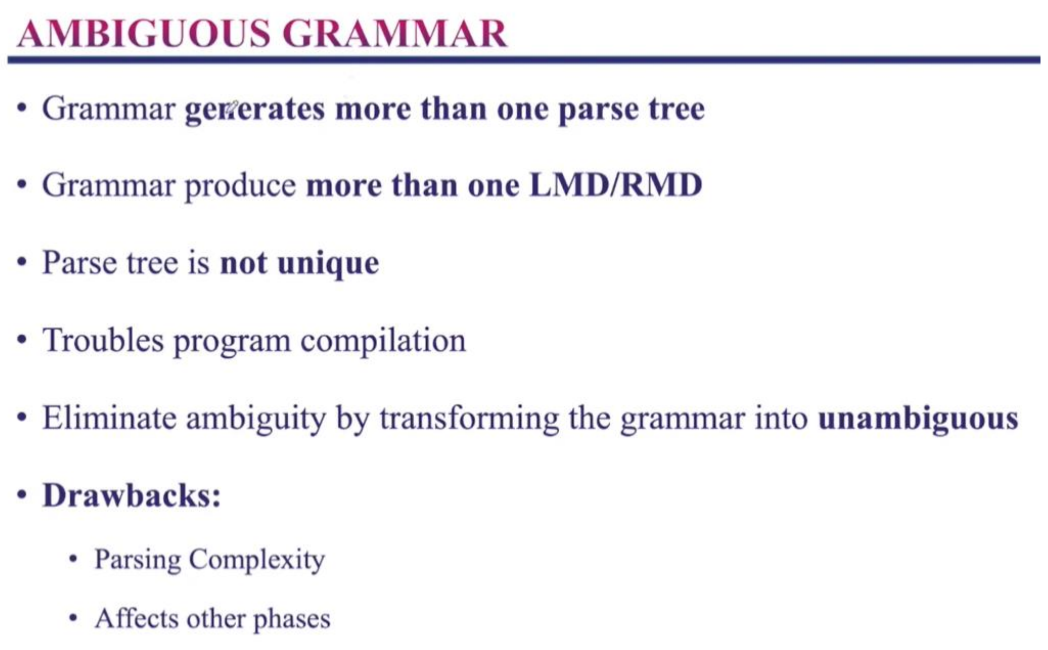
- **Left factoring can introduce ambiguity.** Ambiguity occurs when a grammar can generate two or more different parse trees for the same input string. This can make it difficult for a compiler or interpreter to determine the correct meaning of a program.
- **Left factoring can make grammars more complex.** 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.

Register Allocation

There are two main approaches to register allocation:

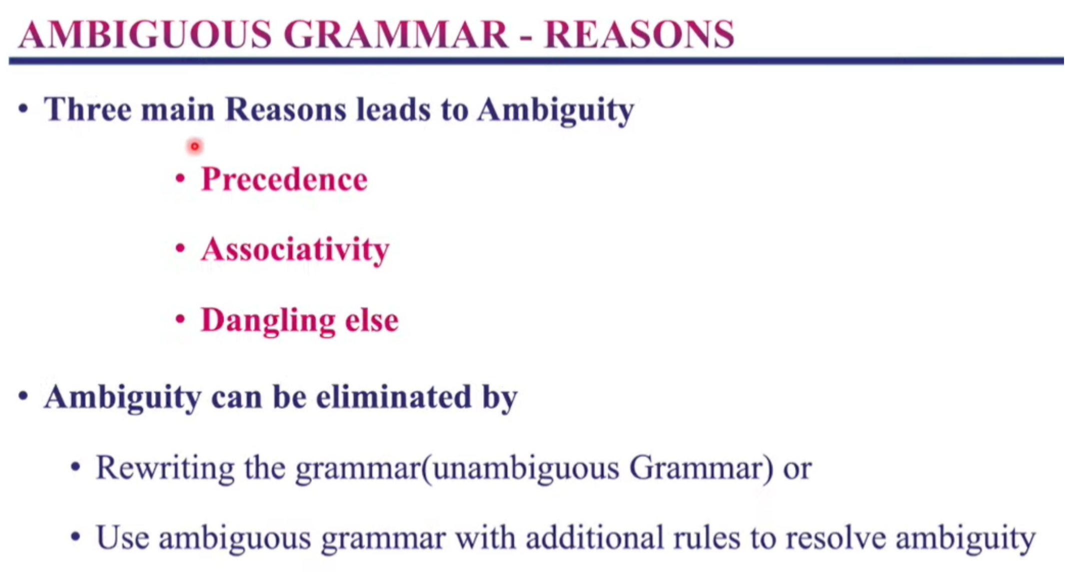
- **Local register allocation:** This approach allocates registers to variables and expressions within a single basic block. A basic block is a sequence of instructions that cannot be interrupted by any jumps or calls.
- **Global register allocation:** This approach allocates registers to variables and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

Ambiguity



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

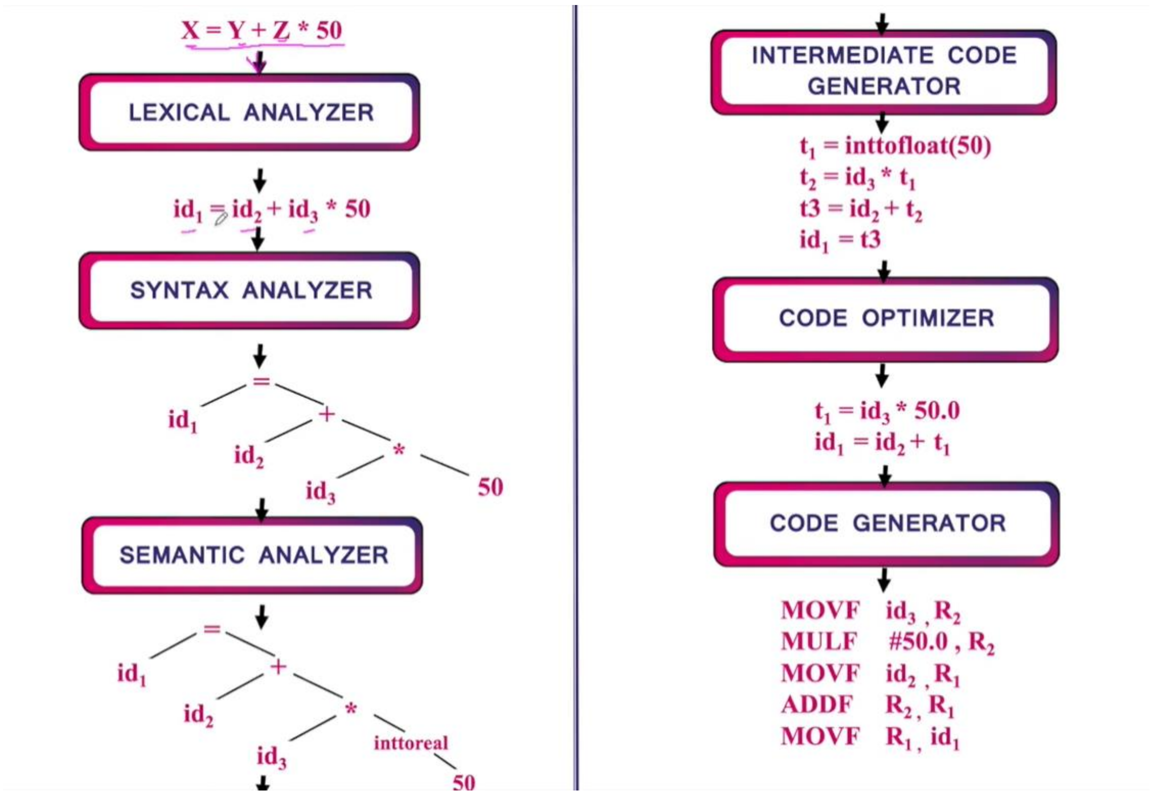
Phases of Compiler

Analysis Phase of Compiler

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.

Synthesis Phase of Compiler

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.



Lex, Lexeme, Lexical Analysed, Token, Role of Lexical Analyser

- **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.
- **Pattern:** In compiler design, a pattern is a set of rules that the lexical analyzer uses to recognize tokens. A pattern can be a regular expression, a finite state machine, or a table lookup.
- **Input buffering:** Input buffering stores the input program in a buffer, and the lexical analyzer reads the input program from the buffer one token at a time. This reduces the number of times the lexical analyzer has to access the input program from disk, which can improve the performance of the compiler.

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.

Lexical Error:

1. Exceeding length of identifier or numeric constants.
2. Appearance of illegal characters
3. Unmatched string
4. Spelling Error
5. Replacing a character with an incorrect character.

Type Checking and Error Reporting

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.

Error reporting is the process of identifying and reporting errors in a program.

There are two main types of errors that can be reported by a compiler:

- **Compile-time errors:** These errors occur when the compiler encounters a syntax error or a semantic error in the program. Compile-time errors prevent the compiler from generating an executable file.

- **Run-time errors:** These errors occur when the program is executed. Run-time errors can be caused by a variety of factors, such as invalid input, division by zero, or accessing an array out of bounds.

Compilers typically report errors to the user in a variety of ways, including:

- **Error messages:** Error messages are displayed to the user on the console or in a text file. Error messages typically include the line number and column number of the error, as well as a description of the error.
- **Colour coding:** Compilers may use color coding to highlight errors in the source code. This can help programmers to quickly identify errors in their code.
- **Warnings:** Warnings are similar to errors, but they do not prevent the compiler from generating an executable file. Warnings are typically used to indicate potential problems with the program, such as unused variables or possible errors in the code.

Heap and Dynamic Storage Allocation

- **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.

Activation Record and Calling Convention

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.

Parameter Passing Methods

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.
```

Code Optimization Techniques

Compiler optimization is the process of improving the performance of a program by analyzing its source code and generating more efficient code.

- **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.

Issues in Design of Code Generator

- **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.
- **Target machine:** The code generator must be able to generate code for the target machine. This includes considering the instruction set architecture, the memory model, and the calling convention.
- **Code quality:** The code generator must generate high-quality code. This means that the code must be correct, efficient, and portable.

Lexical Analyzer: Interaction with Parser and It’s Roles

A **lexical analyzer**, also known as a lexer or scanner, is a program that performs lexical analysis, which is the process of converting a sequence of characters into a sequence of tokens. A token is a sequence of characters that has a specific meaning in the programming language. For example, the token "int" might represent the integer data type, and the token "+" might represent the addition operator.

The lexical analyzer **interacts** with the parser, which is the program that analyzes the syntax of a programming language. The parser takes the tokens produced by the lexical analyzer and uses them to construct a parse tree, which is a representation of the syntactic structure of the program.

The **roles** of the lexical analyzer include:

- Identifying tokens in the input program
- Assigning a meaning to each token
- Passing tokens to the parser
- Reporting errors if an invalid token is found

Tools to Generate Lexical Analyzer

- **LEX** is a tool that was originally developed by Mike Lesk at Bell Labs. LEX is a regular expression-based lexical analyzer generator.
- **Flex** is a free and open-source lexical analyzer generator that is based on LEX. Flex is a more powerful tool than LEX, and it supports a wider range of features.

The **working** of a lexical analyzer generator is as follows:

1. The user provides a lexical specification file to the lexical analyzer generator. The lexical specification file contains a set of rules that define the tokens that are allowed in the language.
2. The lexical analyzer generator uses the lexical specification file to generate a C program that implements the lexical analyzer.
3. The C program is compiled and linked into the compiler.
4. When the compiler encounters an input program, it passes the input program to the lexical analyzer.
5. The lexical analyzer scans the input program and produces a sequence of tokens.
6. The parser then takes the sequence of tokens from the lexical analyzer and constructs a parse tree.

Difficulties in Lexical Analysis

- **Ambiguity:** Some tokens can be ambiguous, meaning that they can be interpreted in multiple ways. For example, the token "if" can be interpreted as either the keyword "if" or the identifier "if".
- **Incomplete tokens:** Sometimes, a token may be incomplete, meaning that it does not have enough characters to be identified. For example, the token "in" is incomplete if it is not followed by a valid identifier.
- **Invalid tokens:** Sometimes, a token may be invalid, meaning that it does not conform to the rules of the programming language. For example, the token "123abc" is invalid because it contains a digit and a letter.

Symbol Table

A symbol table is a data structure used by compilers to store information about the identifiers used in a program. The **main functions** of a symbol table are:

- To store information about identifiers used in a program.
- To provide a way to look up identifiers by name.
- To provide a way to determine the type, scope, and value of an identifier.

The symbol table is **generated** during the lexical analysis phase of the compiler. The lexical analyzer breaks down the source code into tokens. Each token is associated with a key in the symbol table. The value of the key is the type of the token.

The symbol table is **managed** by the compiler during the semantic analysis, code optimization, and code generation phases of the compiler. The semantic analyzer uses the symbol table to check the meaning of the source code. The code optimizer uses the symbol table to optimize the generated machine code. The code generator uses the symbol table to generate the machine code.

The symbol table is used by the compiler to perform a variety of tasks, such as:

- **Name resolution:** The compiler uses the symbol table to resolve identifiers to their corresponding objects. This is necessary to ensure that the compiler can correctly interpret the program.
- **Type checking:** The compiler uses the symbol table to check the types of expressions to ensure that they are compatible. This is necessary to prevent errors at runtime.
- **Scope checking:** The compiler uses the symbol table to verify that identifiers are used in the correct scope. This is necessary to prevent errors at runtime.
- **Optimization:** The compiler can use the symbol table to perform optimizations, such as common subexpression elimination and dead code elimination.

Symbol tables are typically stored in the compiler's **heap memory**. This is because the size of the symbol table can vary depending on the size of the program.

The data structure used for symbol table organization is typically a **hash table**. A hash table is a data structure that maps keys to values. The key is the identifier name, and the value is the identifier's information.

Scope information is represented by symbol table organization using hash tables by dividing the symbol table into a hierarchy of scopes. Each scope has a unique name, and the identifiers declared in a scope are only visible within that scope.

For example, the following code shows a simple symbol table with two scopes:

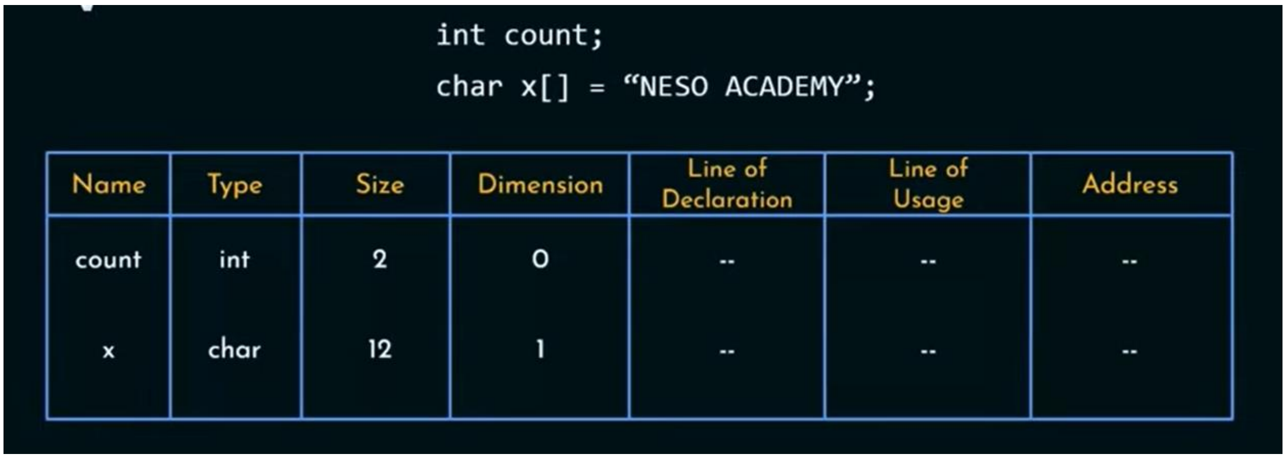
Code snippet
```
int main() {
  int a = 10; // Identifier `a` is declared in the global scope.
  {
    int b = 20; // Identifier `b` is declared in the local scope.
  }
}
```

The symbol table for this code would look like this:

Code snippet

```
Global scope:
  a: int
Local scope:
  b: int
```

STRUCTURE



```
int count;
char x[] = "NESO ACADEMY";
```

Name	Type	Size	Dimension	Line of Declaration	Line of Usage	Address
count	int	2	0	--	--	--
x	char	12	1	--	--	--

The symbol table is manipulated in different phases of compilation as follows:

- **Lexical analysis:** During lexical analysis, the compiler breaks the source code into tokens. Each token is then added to the symbol table.
- **Syntax analysis:** During syntax analysis, the compiler builds a parse tree for the program. The symbol table is used to resolve identifier names to their corresponding nodes in the parse tree.
- **Semantic analysis:** During semantic analysis, the compiler checks the program for errors. The symbol table is used to perform name resolution, type checking, scope analysis, and error detection.
- **Intermediate code generation:** During intermediate code generation, the compiler generates code that represents the abstract syntax tree of the program. The symbol table is used to determine the names of the variables and functions that are used in the intermediate code.
- **Machine code generation:** During machine code generation, the compiler generates machine code for the target machine. The symbol table is used to determine the addresses of the variables and functions that are used in the machine code.

Shift Reduce Parser

A shift-reduce parser is a bottom-up parser that uses a stack to store the tokens of the input program and a table to store the production rules of the grammar. The parser works by repeatedly shifting tokens from the input program onto the stack and then reducing the top two or more symbols on the stack using a production rule from the table. The process terminates when the stack contains only the start symbol of the grammar.

Run-Time Storage Organization

Run-time storage organization is the process of allocating and managing memory during the execution of a program.

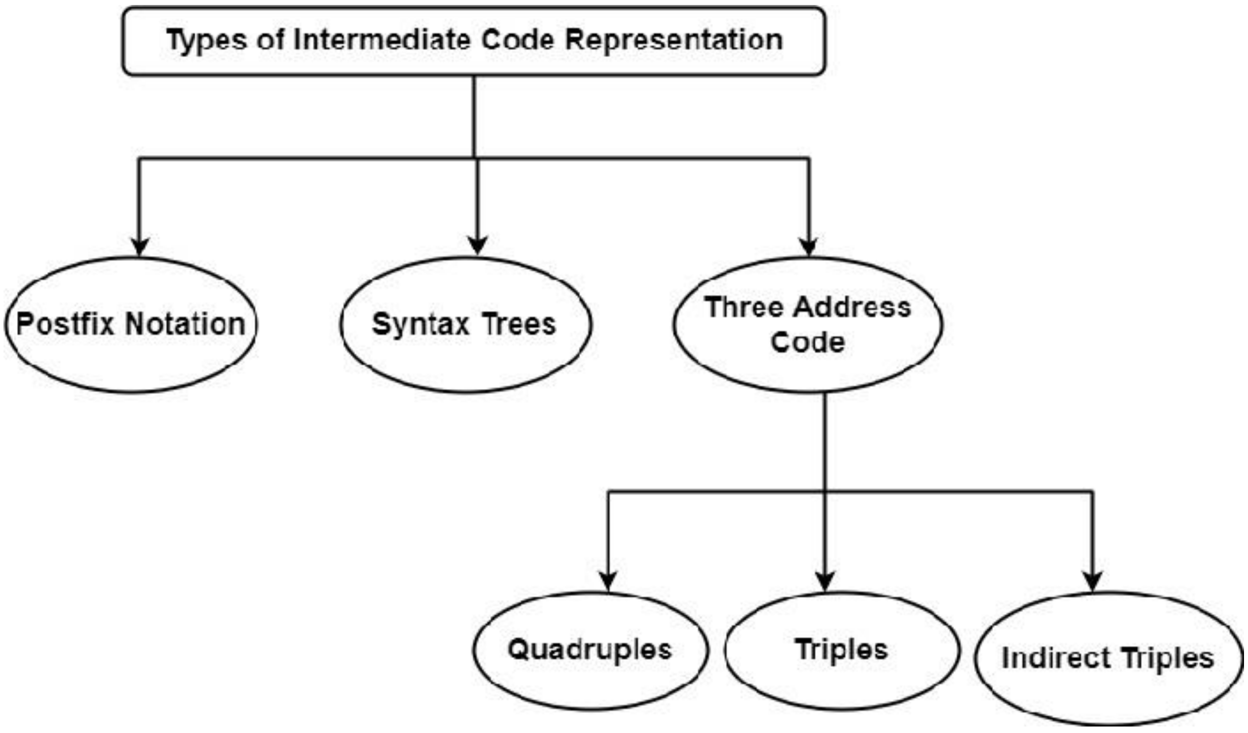
Some common data structures include:

Stack: Stack-based run-time storage organization is a method of allocating and managing memory during the execution of a program. In stack-based storage, memory is allocated on a stack, which is a last-in, first-out (LIFO) data structure. When a function is called, a new activation record is pushed onto the stack. The activation record stores the local variables, function parameters, and return address for the function. When the function returns, the activation record is popped off the stack and the memory is deallocated.

Heap: Heap-based run-time storage organization is a method of allocating and managing memory during the execution of a program. In heap-based storage, memory is allocated dynamically. Heap-based storage is used to store dynamically allocated objects, such as strings and arrays. These objects are created using the `new` keyword and they are deallocated using the `delete` keyword. When an object is created using `new`, the run-time system allocates memory for the object on the heap and returns a pointer to the object. When an object is deleted using `delete`, the run-time system deallocates the memory for the object and frees it up for other use.

Intermediate Code Representation

Intermediate code is used to translate the source code into the machine code.



A **low-level representation** is a representation of data or instructions that is close to the physical hardware of the computer.

Some low-level representations, are:

- **Machine code:** Machine code is the lowest level of representation. It is a set of binary instructions that are directly executed by the computer's processor.
- **Assembly language:** Assembly language is a human-readable form of machine code. It is a low-level language that uses mnemonics to represent machine instructions.
- **Object code:** Object code is a compiled form of a high-level language. It is a low-level representation that can be directly loaded into the computer's memory and executed.

All the intermediate code representations are:

- **Abstract syntax tree (AST):** An AST is a tree-like representation of the source code. Each node in the tree represents a different construct in the source code, such as a variable, a function, or an expression. ASTs are used by compilers to analyse and transform code.
- **Directed acyclic graph (DAG):** A DAG is a graph that has no cycles. DAGs are used by compilers to represent the control flow of a program.
- **Postfix notation:** Postfix notation is a notation for representing expressions in which the operators follow the operands. Postfix notation is often used by compilers to represent expressions.
- **One address code:** One address code is a representation of instructions that uses one address for each instruction. The address specifies the operand of the instruction. One address code is a simple representation of instructions, but it can be inefficient for some types of code.
- **Two address code:** Two address code is a representation of instructions that uses two addresses for each instruction. The first address specifies the operator, and the second address specifies the first operand. Two address code is more efficient than one address code for some types of code, but it is more difficult to understand and generate.
- **Three address code:** Three address code is a representation of instructions that uses three addresses for each instruction. The first address specifies the operator, the second address specifies the first operand, and the third address specifies the second operand. Three address code is the most efficient representation of instructions, but it is the most difficult to understand and generate.

Advantages of intermediate codes are:

- **Machine independence:** Intermediate code is not tied to any particular machine architecture. This means that a compiler can generate intermediate code for a target machine without having to be recompiled. This makes it possible to create compilers that can target multiple machines.
- **Efficient code generation:** Intermediate code can be optimized by a compiler to improve its performance. This optimization can be done at a higher level than it would be possible to do if the compiler were generating machine code directly.
- **Easy debugging:** Intermediate code can be used to debug programs. This is because it is a more readable representation of the program than machine code. This makes it easier to see what the program is doing and to find errors in the program.

Code Generator and Code Generation Algorithm

A **code generator** is a program that converts a high-level language program into a low-level language program, such as machine code or assembly language. The code generator takes the high-level language program as input and produces the low-level language program as output.

The **code generation algorithm** is the set of steps that the code generator follows to convert the high-level language program into the low-level language program. The code generation algorithm typically includes the following steps:

1. **Parsing:** The code generator parses the high-level language program to create an abstract syntax tree (AST). The AST is a tree-like representation of the high-level language program.
2. **Analysis:** The code generator analyzes the AST to determine the meaning of the high-level language program. This includes determining the types of variables, the control flow of the program, and the operations that need to be performed.
3. **Generation:** The code generator generates the low-level language program from the AST. This includes generating instructions for the target machine, assigning registers to variables, and inserting comments.

Register Allocation

Register allocation is a compiler optimization technique that assigns variables to registers in the target machine. Registers are fast memory locations that can be used to store data and instructions. By assigning variables to registers, the compiler can improve the performance of the generated code.

There are two main approaches to register allocation: global register allocation and local register allocation.

- **Global register allocation** attempts to find a global solution that assigns all variables to registers. This is the most challenging approach, but it can lead to the best performance.
- **Local register allocation** attempts to find a local solution that assigns variables to registers within a basic block. This is a less challenging approach, but it can still lead to significant performance improvements.

YACC: Specification, Parser Generated Precedence and Associability

YACC (Yet Another Compiler Compiler) is a parser generator that takes a grammar as input and produces a parser in C or C++. The parser can be used to analyze and interpret programs written in the language described by the grammar.

A YACC **specification** consists of three sections:

1. Rules section: This section defines the grammar for the language.
2. Definitions section: This section defines any constants, variables, or functions that are used in the rules section.
3. User subroutines section: This section defines any subroutines that are called from the rules section.

Yacc (Yet Another Compiler-Compiler) **generates** parsers known as LALR (Look-Ahead Left-to-Right) parsers. LALR parsers are a type of bottom-up parsing technique that can efficiently handle a wide range of context-free grammars.

Precedence and Associativity are specified using the **%left, %right,** or **%nonassoc** directives.

%left indicates left-associativity, %right indicates right-associativity, and %nonassoc indicates non-associativity.

Precedence declarations specify the relative precedence of different operators or grammar rules. Higher precedence is assigned to operators or rules that should be evaluated first. Associativity declarations define the associativity of operators with the same precedence level.

Example:

```
%left '='

%right '-'

%nonassoc '<' '>'
```

The following are some of the **error recovery** actions that YACC can perform:

- **Error recovery by backtracking:** When an error is detected, YACC can try to recover by backtracking to a previous state in the parse. This can be done by discarding the tokens that have been read since the error occurred and then trying to parse the input again.
- **Error recovery by error recovery rules:** YACC can also define error recovery rules that are used to handle specific types of errors. These rules can be used to skip over invalid input or to generate error messages.
- **Error recovery by user-defined actions:** YACC allows the user to define user-defined actions that are executed when an error occurs. These actions can be used to do anything that is necessary to recover from the error, such as printing an error message or skipping over invalid input.

Code Generator

Code generator is a part of a compiler that translates the high-level language code into low-level code, such as assembly language or machine code. The code generator is responsible for generating code that is efficient and easy to understand by the target machine.

The code generator typically works by first translating the high-level language code into an intermediate representation, such as a tree or a graph. The intermediate representation is then used to generate the low-level code.

Compiler Construction Tools

Compiler construction tools are tools that help developers to build compilers. These tools can be used to automate many of the tasks involved in compiler construction, such as lexical analysis, syntax analysis, semantic analysis, and code generation.Some of the most popular compiler construction tools are:

- **Bison:** Bison is another parser generator that is similar to ANTLR.
- **Flex:** Flex is a lexical analyzer generator that can be used to generate lexical analyzers for a variety of programming languages.
- **YACC:** YACC is a parser generator that is similar to Bison.

Scanner, Tokens and Lexeme

A **scanner,** also known as a lexical analyzer, is the first phase of a compiler. It converts the input program into a sequence of tokens. It uses a set of rules to determine whether each character is part of a token or not. If the scanner sees a sequence of letters that matches a keyword, it will create a token for that keyword. Once the scanner has created all of the tokens, it passes them to the parser, which is the next phase of the compiler.

A **token** is a sequence of characters that has a meaning in the programming language. For example, the token "int" is a keyword that represents an integer.

A **lexeme** is a sequence of characters that is recognized by the scanner as a single token. For example, the lexeme "int" is a keyword in C programming language.

Benefits of Using Machine Independent Intermediate Forms

- **Portability:** A compiler that uses a machine independent intermediate form can be easily ported to new target machines. This is because the intermediate form is not specific to any particular machine, so the compiler does not need to be re-written for each new target machine.
- **Efficiency:** A machine independent intermediate form can be used to perform optimizations that would not be possible if the compiler generated code directly for the target machine. This is because the intermediate form is a more abstract representation of the program, so the compiler has more freedom to perform optimizations.
- **Flexibility:** A machine independent intermediate form can be used to generate code for different target machines. This allows the compiler to be used to generate code for a variety of different platforms, without having to be re-written for each platform.

Primary Structure Preserving Transformations On Basic Blocks

- **Common subexpression elimination:** This transformation removes common subexpressions from a basic block. For example, if a basic block contains the expressions `a + b` and `c + b`, then common subexpression elimination can be used to remove the expression `b` from both expressions. This can improve the performance of the generated code.
- **Dead code elimination:** This transformation removes dead code from a basic block. Dead code is code that is never executed. For example, if a basic block contains the statement `if (x == 0) { y = 1; }` and the value of `x` is always known to be non-zero, then dead code elimination can be used to remove the statement `y = 1;` from the basic block. This can improve the performance of the generated code.
- **Renaming of temporary variables:** This transformation renames temporary variables in a basic block. This can improve the readability of the generated code and can also help to prevent conflicts between names of temporary variables.
- **Interchange of two independent adjacent statements:** This transformation interchanges two independent adjacent statements in a basic block. This can improve the performance of the generated code if the two statements are independent of each other.

First and Follow

- **First set:** The first set of a non-terminal symbol is the set of all terminals that can appear at the beginning of a string produced by that non-terminal symbol.
- **Follow set:** The follow set of a non-terminal symbol is the set of all terminals that can appear immediately after that non-terminal symbol in a string produced by the grammar.

Example:

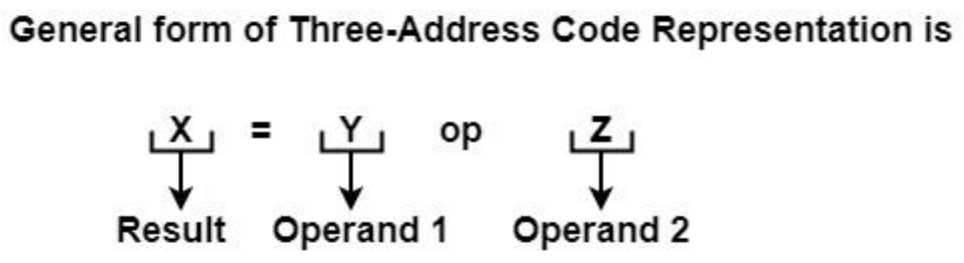
E -> E + T | T

T -> T * F | F

F -> (E) | id

Non Terminal	First	Follow
E	{ (, id }	{ $, +,) }
T	{ (, id }	{ $, +,), * }
F	{ (, id }	{ $, +,), * }

Three Address Code



https://www.youtube.com/watch?v=uzjk7UopdSk&list=PLEqLbPpJwryvcGCpfLTLghmDRDOyrlz72&index=43

Bootstrapping in Self Hosting Compiler

Bootstrapping is the process of writing a compiler for a programming language using the language itself. Bootstrapping is used to produce a self-hosting compiler. **Self-hosting compiler** is a type of compiler that can compile its own source code.

Illustration of bootstrapping in self hosting compiler:

1. Initial Compiler: Create a basic version of the compiler written in a different language or using a different compiler.
2. Language Implementation: Implement a more comprehensive version of the compiler in the target language, supporting a wider range of language features.

3. Compilation Process: Compile the new version of the compiler using the initial compiler, translating the source code into executable machine code or bytecode.
4. Test and Verification: Use the compiled new compiler to compile its own source code and compare the output against the initial compiler to ensure consistency.
5. Iterative Refinement: Fix any discrepancies or bugs in the new compiler's source code, recompile, and repeat the test and verification process.
6. Deployment: Once the new compiler consistently produces the expected results, it becomes the official version for compiling programs in the target language.

There are two common approaches to bootstrapping a compiler:

1. **Top-down bootstrapping:** This approach involves writing a compiler in a different language, such as C or C++, and then using that compiler to compile the target language. This is the most common approach, as it is relatively easy to write a simple compiler in a language like C or C++.
2. **Bottom-up bootstrapping:** This approach involves writing a small subset of the target language first, and then using that subset to compile the rest of the language. This is a more difficult approach, but it can lead to a more efficient compiler.

Runtime Environment

The runtime environment is a set of data structures and routines that are used to manage the execution of a program. It provides the program with access to memory, files, and other resources, and it handles tasks such as garbage collection, exception handling, and thread synchronization.

There are three main types of runtime environments:

- **Static runtime environment:** A static runtime environment is created when the program is compiled. It is stored in the program's executable file and is used to load the program into memory, initialize its data structures, and start its execution.
- **Stack-based runtime environment:** A stack-based runtime environment is created when a function is called. It is stored on the stack and is used to store the function's local variables, return address, and other data.
- **Dynamic runtime environment:** A dynamic runtime environment is created when a program is executed. It is stored in memory and is used to store the program's global variables, heap data, and other data.

Activation Record

The activation record is created when the function is called and is destroyed when the function returns.

The different fields of an activation record are as follows:

- **Return Value:** It is used by calling procedure to return a value to calling procedure.

- **Actual Parameter:** It is used by calling procedures to supply parameters to the called procedures.
- **Control Link:** It points to activation record of the caller.
- **Access Link:** It is used to refer to non-local data held in other activation records.
- **Saved Machine Status:** It holds the information about status of machine before the procedure is called.
- **Local Data:** It holds the data that is local to the execution of the procedure.
- **Temporaries:** It stores the value that arises in the evaluation of an expression.

Example:

```
int factorial(int n) {

    int result = 1;

    if (n == 0 || n == 1) {

        return result; // Return Value

    }

    for (int i = 2; i <= n; i++) {

        result *= i;

    }

    return result; // Return Value

}

int main() {

    int num = 5;

    int factorialResult = factorial(num); // Actual Parameter

    return 0;

}
```

Return Value: The return value field will store the computed factorial result.

Actual Parameter: The actual parameter field will contain the value of the num variable passed from the main() function.

Local Data: The local data field will hold the value of the result variable, which is initially set to 1 and updated during the factorial calculation.

Control Link: Not applicable in this example.

Access Link: Not applicable in this example.

Saved Machine Status: Not applicable in this example.

Temporaries: Not applicable in this example.

Bison and Flex

Bison is a parser generator that uses a formal grammar to generate a parser. A formal grammar is a set of rules that define the syntax of a programming language. Bison can generate parsers for a wide variety of programming languages, including C, C++, Java, and Python.

Flex is a lexical scanner generator that uses a regular expression to generate a lexical scanner. A lexical scanner is a program that analyzes the input stream of a programming language and converts it into a sequence of tokens. Flex can generate lexical scanners for a wide variety of programming languages, including C, C++, Java, and Python.

IN and OUT sets

IN and OUT sets are used in global dataflow analysis to track the flow of data through a program. Global dataflow analysis is a type of dataflow analysis that is used to analyze the flow of data through an entire program.

The **IN** set of a statement contains all of the variables that are defined before the statement, and the **OUT** set of a statement contains all of the variables that are defined by the statement.

Advantages: improved performance, reduced code size, simplified code

Drawbacks: can be time-consuming, can be complex, can be inaccurate

Parameter	**Context Sensitive Grammar**	**Context Free Grammar**
Language	Context Sensitive Language	Context Free Language
Automata	Linear Bounded Automata	Push Down Automata
Also called	Type 1 Grammar	Type 2 Grammar

LR(0)	LR(1)
LR(0) stands for "Look-Ahead, Rightmost derivation with 0 tokens of lookahead."	LR(1) stands for "Look-Ahead, Rightmost derivation with 1 token of lookahead."
In LR(0) parsing, the parser decides the next action based solely on the current state and the next input symbol.	In LR(1) parsing, the parser considers both the current state and the next input symbol, along with the lookahead token, to make parsing decisions.
The LR(0) algorithm constructs a canonical collection of LR(0) items and builds an LR(0) parsing table using this collection.	The LR(1) algorithm constructs a canonical collection of LR(1) items and builds an LR(1) parsing table using this collection.
It is the simplest form of the LR parsing algorithm.	LR(1) parsing is an extension of the LR(0) algorithm that incorporates one token of lookahead information.
The parser uses a set of LR(0) items to represent the parsing states.	The parser uses a set of LR(1) items to represent the parsing states.
Limitation: It cannot handle conflicts that arise due to the lack of lookahead information.	Limitation: It requires a more extensive parsing table and may lead to an increase in the size of the parser.

Power	More Powerful	Less Powerful

Feature	**Parse Tree**	**Syntax Tree**
Concreteness	More concrete	Less concrete
Information	Contains all information about the input	Contains only the information necessary to generate machine code or intermediate code
Efficiency	Less efficient	More efficient
Use	Used during the early stages of compilation	Used during the later stages of compilation

Feature	YACC	Bison
Development date	1970s	1980s
Ease of use	Difficult	Easier
Error recovery mechanisms	Not always robust	More robust
Portability	Not as portable	More portable
Best use	For applications where ease of use is not a priority	For most applications

Feature	Syntax Error	Semantic Error
Cause	Incorrect structure of the program	Incorrect meaning of the program
Detection	Lexical analysis and syntax analysis	Semantic analysis
Example	Missing semicolon, unmatched parenthesis, invalid identifier	Dividing by zero, accessing an array element that is out of bounds, using a variable that has not been declared

Issues in design of Lexical Analyser

- **Tokenization:** It means breaking the input stream into a sequence of tokens. Ambiguities in tokenization can lead to parsing errors and incorrect interpretation of the source code.

- **Identifiers:** Identifiers are names of variables, functions, and other entities in a program. Identifiers must be distinguished from other tokens, such as keywords and operators.
- **Comments:** Comments are used to provide information about the program that is not part of the program's syntax. Comments must be ignored by the lexical analyzer.
- **Whitespace:** Whitespace characters, such as spaces, tabs, and newlines, are not part of the program's syntax. They must be ignored by the lexical analyzer.
- **Error detection:** The lexical analyzer must be able to detect errors in the input stream. For example, if the input stream contains an invalid identifier or an unclosed comment, the lexical analyzer must generate an error message.

Loader and Linker

A **loader** is a software component that loads programs and libraries into memory. The loader performs the following tasks:

- Loads programs and libraries into memory
- Resolves external references
- Initializes the program or library environment
- Transfers control to the program or library entry point

The **loading process** typically involves the following steps:

1. The loader locates the program or library file on disk.
2. The loader reads the program or library file into memory.
3. The loader resolves any external references in the program or library.
4. The loader initializes the program or library environment.
5. The loader transfers control to the program or library entry point.

The **link editor** is a software component that links together object files to create a single executable file. The link editor performs the following tasks:

1. Resolves external references between object files.
2. Combines the code and data from the object files into a single executable file.
3. Creates a symbol table for the executable file.

Compiler Construction Tools

Compiler construction tools are software tools that help in the implementation of various phases of a compiler. Some of the most common compiler construction tools include:

- **Scanner generators:** Scanner generators are typically used to create lexical analyzers. A lexical analyzer is a program that breaks down the input code into tokens. A token is a sequence of characters that has a specific meaning in the programming language. For example, in the C programming language, the token "int" is used to declare an integer variable.
- **Parser generators:** Parser generators are used to create parsers. A parser is a program that analyzes the syntax of the input code. The parser determines whether the input code is

syntactically correct and, if it is, builds a parse tree. A parse tree is a data structure that represents the syntactic structure of the input code.

- **Syntax-directed translation engines:** Syntax-directed translation engines are used to implement the semantic analysis and code generation phases of the compiler. Semantic analysis is the process of determining the meaning of the input code. Code generation is the process of producing output code from the input code.
- **Data-flow analysis engines:** Data-flow analysis engines are used to perform various data-flow analyses. Data-flow analysis is a technique for determining the flow of data through a program. Data-flow analyses are used to perform optimizations such as dead code elimination and constant folding.
- **Compiler construction toolkits:** Compiler construction toolkits are integrated sets of tools that provide support for all phases of compiler construction. Compiler construction toolkits typically include scanner generators, parser generators, syntax-directed translation engines, data-flow analysis engines, and other tools.

Code Generation Process for an Arithmetic Operation

The code generation process for an arithmetic operation in compiler design can be broken down into the following steps:

1. **Parse the source code.** The compiler first parses the source code to create an abstract syntax tree (AST). The AST represents the structure of the source code in a tree-like format.
2. **Generate intermediate code.** The compiler then generates intermediate code from the AST. Intermediate code is a language-independent representation of the source code that is easier to optimize and generate machine code from.
3. **Optimize the intermediate code.** The compiler then optimizes the intermediate code to improve its performance and efficiency.
4. **Generate machine code.** The compiler then generates machine code from the optimized intermediate code. Machine code is the language that the target machine understands.

The following instructions would be generated for the statement `t=a-b, u=a-c, v=t+u`:

```
MOV R1, a
MOV R2, b
SUB R1, R2
MOV t, R1
MOV R1, a
MOV R2, c
SUB R1, R2
MOV u, R1
MOV R1, t
ADD R1, u
MOV v, R1
```

Code Generation from DAG

To generate code from DAG, we follow these steps:

1. Create a list of all the nodes in the DAG.

2. For each node in the list, create a piece of code that represents the operation performed by that node.
3. Arrange the pieces of code in the order that the nodes are connected in the DAG.
4. Join the pieces of code together to create the final output code.

Assembler and Interpreter

Assembler: An assembler is a program that translates assembly language into machine code. Assembly language is a low-level programming language that is used to directly control the hardware of a computer. Assemblers are used to write programs that need to be executed very efficiently, such as operating systems and device drivers.

Interpreter: An interpreter is a program that reads and executes instructions written in a high-level programming language. High-level programming languages are designed to be easy for humans to read and write, but they are not directly understandable by computers. Interpreters translate high-level programming language instructions into machine code one at a time, and then execute the machine code.

Basic Blocks and Flow Graphs

- **Basic blocks:** A basic block is a sequence of instructions that has a single entry point and a single exit point. This means that the instructions in a basic block can only be executed in one order, and they can only be executed from the beginning to the end.
- **Flow graphs:** A flow graph is a graphical representation of a program's control flow. It consists of nodes, which represent basic blocks, and edges, which represent the possible flow of control between basic blocks.

Compiler and Interpreter

- **Compiler:** A compiler is a program that converts all of the source code in a program into machine code at once. This means that the compiler must read the entire program before it can start executing it. Once the compiler has finished compiling the program, it generates an executable file that can be run directly by the computer.
- **Interpreter:** An interpreter is a program that reads and executes one line of source code at a time. This means that the interpreter does not need to read the entire program before it can start executing it. The interpreter can start executing the program as soon as it has read the first line of source code.

Need of Compiler

- **Efficiency:** Compilers can generate machine code that is more efficient than code that is written by hand. This is because compilers can optimize the code for the specific architecture of the computer that it will be running on.
- **Portability:** Compilers can generate machine code that can be run on different types of computers. This makes it possible to write programs that can be run on a variety of platforms, without having to rewrite the code for each platform.

- **Reusability:** Compilers can generate machine code from source code that is written in a high-level language. This makes it possible to reuse code that has been written for other projects.
- **Error checking:** Compilers can check the source code for errors before it is compiled. This can help to prevent errors from occurring when the program is executed.

Storage Allocation Strategies

- **Static storage allocation:** In static storage allocation, the compiler determines the amount of storage required for each variable at compile time. The compiler then allocates that storage in the program's data segment. Static storage allocation is typically used for local variables, parameters, and global variables.
- **Stack-based storage allocation:** In stack-based storage allocation, the compiler allocates storage for local variables and parameters on a stack. The stack is a data structure that grows and shrinks dynamically as the program executes. Stack-based storage allocation is typically used for local variables and parameters.
- **Heap-based storage allocation:** In heap-based storage allocation, the compiler allocates storage for variables dynamically at runtime. The compiler uses a data structure called a heap to manage the memory allocated on the heap. Heap-based storage allocation is typically used for dynamically allocated objects, such as strings and arrays.

Type Expression and Type Conversion

Type expression is a construct that represents the type of a variable or expression.

Different types of type expressions are:

- **Simple type expressions:** Simple type expressions represent the basic types of variables, such as integers, floating-point numbers, and strings.
- **Structured type expressions:** Structured type expressions represent more complex types, such as arrays, classes, and functions.
- **Type constructors:** Type constructors are used to create new types from existing types. For example, the type constructor `array` can be used to create an array of a given type.

Type conversion is the process of converting the type of a variable or expression to another type.

Different types of type conversions are:

- **Implicit type conversions:** Implicit type conversions are performed automatically by the compiler. For example, if an integer variable is used in an operation that expects a floating-point number, the compiler will implicitly convert the integer to a floating-point number.
- **Explicit type conversions:** Explicit type conversions are performed by the programmer using a cast operator. For example, the expression `(float)10` casts the integer `10` to a floating-point number.

Dataflow Analysis of structured flow graph

1. **CFG Construction:** Build a graph representing program control flow without branches.
2. **Initialize Dataflow Sets:** Create sets for each basic block, including "gen" (defined variables) and "kill" (overwritten variables).
3. **Dataflow Equations:** Define equations for data flow between sets, such as "in" (predecessor block's "out" sets minus killed variables) and "out" (union of "gen" and difference between "in" and killed variables).
4. **Dataflow Analysis Algorithm:** Use an iterative algorithm to compute "in" and "out" sets until convergence.
5. **Extract Dataflow Information:** Gather relevant information from "in" and "out" sets, like live variables or conflicts.
6. **Optimization and Transformation:** Apply optimizations based on dataflow analysis, such as dead code elimination or improved register allocation.