Partial C Compiler.

Feb 2023

Kareen Ziadat

Contents

Understanding compiler 4
Definition 4
Role 4
Practical example 4
Features 4
Language processor types 5
Compiler 5
Function 5
Advantages 5
Disadvantages 6
Use Cases 6
Example 6
Interpreter 7
Function 7
Advantages 7
Disadvantages 7
Use Cases 7
Hybrid 9
How it works 9
Advantages 9
Disadvantages 9
Use Cases 9
Compilation within the program execution process 11
The internal phases of the compilation process 14
The lexical analyzer 18
Lexical analyzer role in terms of its inputs and outputs 18
Data structures, and techniques to build. 19
Its advantages and disadvantages 20
Why its separated from the other parts of the phases 21
Example to demonstrate how it works. 22
Relationship between lexical analyzer, source language, finite state automata and regular expression. 23
Conversion 25

Between regular expression to nondeterministic finite automata to deterministic finite automata. 25
RE to DFA RE: (same as RE to DFA) 29
Generating and testing input strings using both DFAs produced. 31
The Parser 32
Role of the parser in terms of its inputs and outputs 32
Data structures, and techniques to build. 33
Its advantages and disadvantages 34
Why its separated from the other parts of the phases 35
Example to demonstrate how it works. 36
The relationships between the parser and the grammar for the source language. 37
Overall 37
Below is an example showing the relationships between the parser and the grammar for the source language. 38
Grammar 38
Grammar and Parsing 44
Grammar in Parser Design 44
Parsing Techniques 45
Top-Down Parsing: 45
Bottom-Up Parsing: 45
Top down: 47
Bottom up: 49
Code generation and optimization 59
Code generation process 59
Code optimization process 62
Referances 65

Understanding compiler

Definition

A compiler is a software tool that converts high-level programming code written in a source language into machine code or intermediate code that a computer can execute. The method consists of numerous steps, including lexical analysis, syntax analysis, semantic analysis, code optimization, and code generation, which will be discussed more.

Role

The compiler's job is to turn a program's whole source code into an executable file that can then be run on a specific computer architecture.

Practical example

Say you've written a story (your program) in English (a high-level programming language), that you would like to explain to someone who only understands Arabic (the machine code). For the other person (the computer) to be able to read your story, you will need to translate it. In this case, the compiler functions as a translator, taking your story (program) and translating it into Arabic (machine code).

Features

A compiler's main features are lexical analysis, syntax analysis, semantic analysis, optimization, and code generation. The compiler breaks down the source code into tokens (smallest units of meaning, during lexical analysis). Syntax analysis guarantees that the code follows the programming language's grammar. Semantic analysis validates the code's meaning by discovering inconsistencies. The process of optimizing increases the code's efficiency and performance. Finally, the compiler generates the target code, which may be run on the target hardware.

Characteristics an effective compiler should have:

Compilation speed: The time it takes a compiler to translate source code into machine code. A faster compilation speed reduces waiting and development time while also allowing for faster iterations during the development process. Optimization techniques are frequently included in compilers to increase the efficiency of the output code. These optimizations, however, can lengthen compilation time and compilers may include options to modify the extent of optimization, allowing developers to achieve a balance between compilation speed and generated code performance.

Correctness of machine code: The compiler must accurately translate high-level source code into machine code while avoiding errors and unexpected behavior. This entails carefully addressing language syntax, grammar, semantics. Compiler should ensure that the translated code accurately matches the original source code and that the instructions are carried out without mistakes or unexpected behavior.

Preservation of code meaning: The compiler should accurately translate the source code into machine code while preserving its functionality and logic. During the translation process, it should respect the semantics of the original code, reproduce the programmer's purpose, and correctly execute the program.

Speed of machine code: A good compiler should generate machine code that is both correct and efficient in terms of execution speed and resource efficiency. Efficient code generation techniques, optimizations, and target-specific optimizations can all have a major impact on the resulting program's runtime performance.

Good error handling: Compiler should check for syntax, type, semantic errors, and other issues that may break language rules or create runtime errors. Every compiler should find and report issues in the source code. It should give clear error messages to help find and resolve errors saving debugging time. It should also detect errors early in the compilation process, preferably at the parsing or semantic analysis stages, so developers can save time by avoiding wasteful compilation attempts on code with known faults.

Checking the code correctly according to grammar: The compiler should check the source code for grammatical problems (syntax) and guarantee that it follows the semantics of the programming language, which includes type checking, scoping rules, and other language-specific constraints.

Language processor types

Compiler

Function

Converts machine code from high-level language code into a semantically equivalent program in target language (machine code) before execution. An executable file is produced by compiling the full code all at once.

Advantages

- Even though they are slower than interpreters, the executable executes more quickly after compilation because it is already written in machine code, as it does not require real-time translation.
- Since the code is already compiled, the executable consumes less resources when it is running.
- Better error detection at compile time since they scan the complete code before it begins executing, allowing them to catch various types of problems (syntax errors, type mismatches, and semantic errors), whereas interpreters simply detect errors line by line as the code executes, potentially overlooking flaws in unexecuted areas.

Disadvantages

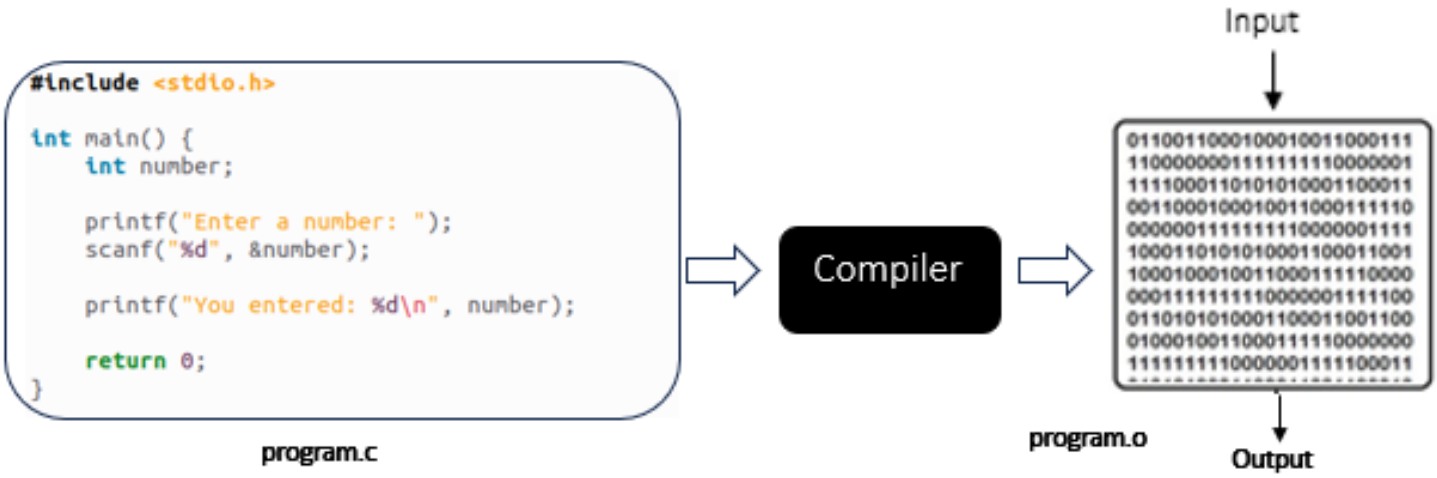
- It is slower since it can take a while to compile and run the application to identify errors.
- Any modification to the code requires recompiling the entire software, which can be time-consuming.
- Compilers often discover problems in the entire program at compile time which may cause error detection to be delayed, especially in large programs.

Use Cases

suitable for large-scale applications requiring high performance, such as system software and game development.

Mainly used in low-level languages (C, C++). C language is processed using a compiler. Before running the program, the compiler converts the full C source code to machine code which is saved in the form of an executable file. There are several C compilers available, including GCC (GNU Compiler Collection) and Clang.

Example



This source code is passed into a compiler, which checks it for syntax mistakes and then creates an object file or executable if none are discovered. The compilation process produces a file containing machine code, which is low-level code that may be run directly by the computer's CPU. When the software is launched in this case, it will prompt the user to enter a number, which it will then read and show to the user.

Interpreter

Function

The interpreter reads and executes high-level programming language code line by line. It converts each line of source code into a set of operations that the computer can understand and execute as it reads it. It directly runs the processes defined in the source program and uses user inputs or data supplied at run time if needed. This interpreter moves progressively through the program's code, translating and executing each line as it goes. This approach to code handling differs from that of a compiler, which translates the complete program before any part of it is run.

Advantages

- Interpreters run code line by line, making it easier to detect and correct errors.
- Program starts executing faster than compiler.
- They enable rapid code modification (changes may be made and tested immediately) since there is no need to recompile.
- Interpreted code can execute on any machine that has the necessary interpreter, enabling compatibility across platforms.

Disadvantages

- Interpreters read, interpret, and execute code line by line so execution is much slower than running an already compiled code.
- They can consume more resources during execution since they must interpret each line of code every time the application runs.
- Interpreters discover problems at runtime, which might be inefficient for identifying certain errors.
- To run the code, the target machine must have the necessary interpreter installed.

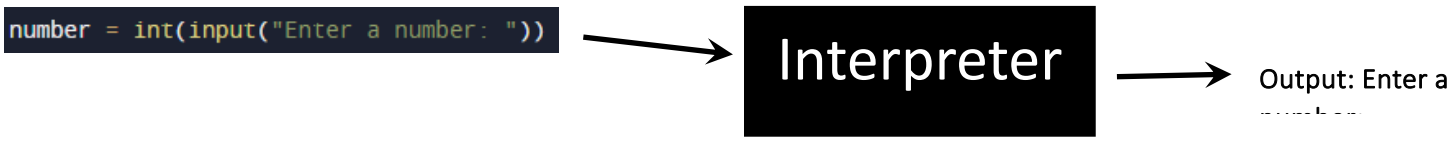
Use Cases

Ideal for development and testing environments, scripting requiring rapid testing and feedback. It dominates high-level languages (Python, Ruby) which immediately executes the instructions in the source code, transforming them into machine code line by line. The most widely used Python interpreter is CPython, which is written in C.

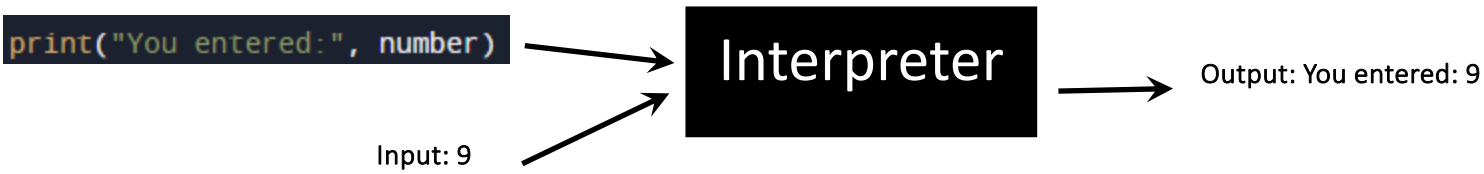
Let’s look at the example. Here is the same source program written in python:

```
number = int(input("Enter a number: "))
print("You entered:", number)
```

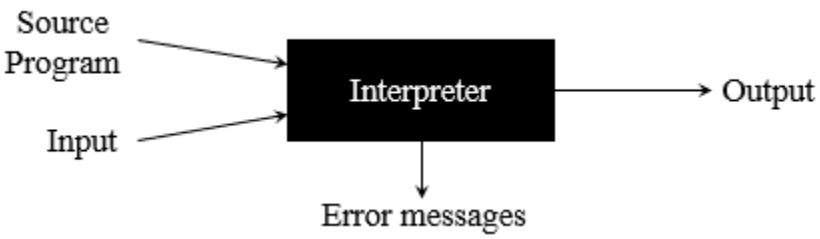
It processes line 1 number = int(input("Enter a number: ")), which instructs the interpreter to show the string "Enter a number:" and wait for the user to input a number. When a number is entered by the user, the interpreter converts it to an integer and stores it in the variable number.



The interpreter then performs the second line of code, which is print("You entered:", number). This line displays the words "You entered:" followed by the number entered by the user.



Process in general



Hybrid

How it works

It employs a compiler and an interpreter, utilizing an approach known as Just-In-Time (JIT) compilation.

The Java compiler (javac) initially compiles Java code which does not generate machine code for any platform, instead, it compiles Java code into bytecode, a platform-independent intermediate form and can now be run on any machine that has a Java Virtual Machine (JVM), enabling Java's "write once, run anywhere" (WORA) philosophy.

The Java Virtual Machine is where this Java bytecode is executed. JVM is platform-dependent, which means that multiple versions of JVM exist for different operating systems (Windows, macOS, Linux, etc.).

Within the JVM, an interpreter will execute the bytecode line by line, converting it to machine code that the host computer can execute.

JIT Compiler is a component of the JVM that optimizes this process for improved performance. It compiles parts of the bytecode that are frequently called into machine code, allowing them to run faster. This compilation occurs at runtime. JIT compiler It improves performance because frequently executed bytecode does not need to be interpreted every time, reducing execution time.

Advantages

- By compiling Java code into bytecode (platform-independent intermediate form), Java programs can run on any machine with a Java Virtual Machine (JVM).
- The JIT compiler (component of the JVM) improves efficiency by converting frequently used bytecode sections into machine code during runtime.
- The system can begin running the code without waiting for the entire compilation process to complete by interpreting bytecode line by line.

Overall, it combines the advantages of interpretation and compilation.

Disadvantages

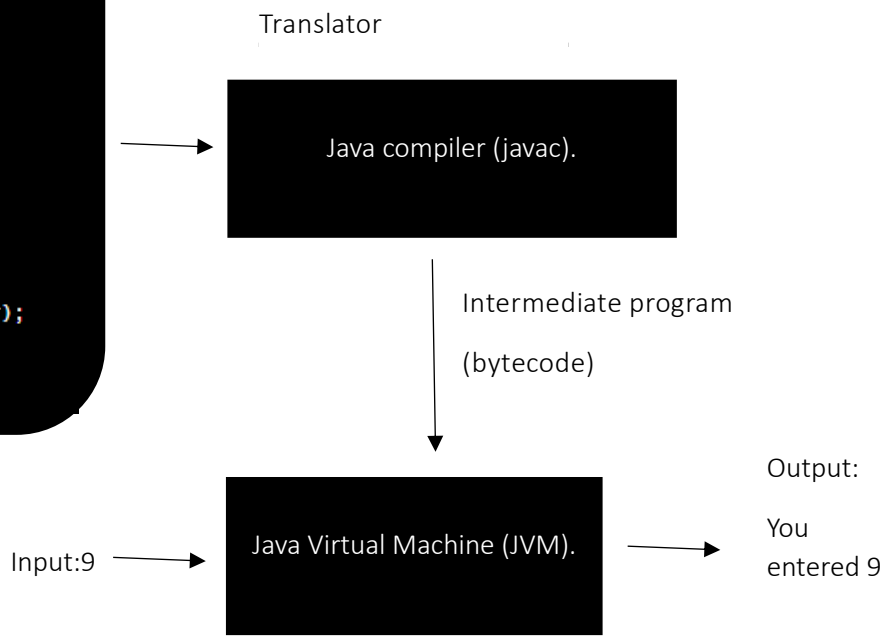
- When the code is first executed, the JIT compilation process may cause a delay since the compiler needs time to convert the bytecode into machine code. This can affect performance, especially in short-running applications.
- JIT compilers use more memory because they keep both the source bytecode and the produced machine code.
- A lot of overhead because JIT compiler requires complicated algorithms to determine which parts of the code to compile and how to optimize them.

Use Cases

Used in Java, Javascript, WebAssembly. Java is often used for server-side programs such as web servers and application servers, where JIT compilation helps handle heavy loads. It is also ideal for applications that must run on several operating platforms (such as Windows, macOS, and Linux) without requiring source code changes.

Source code

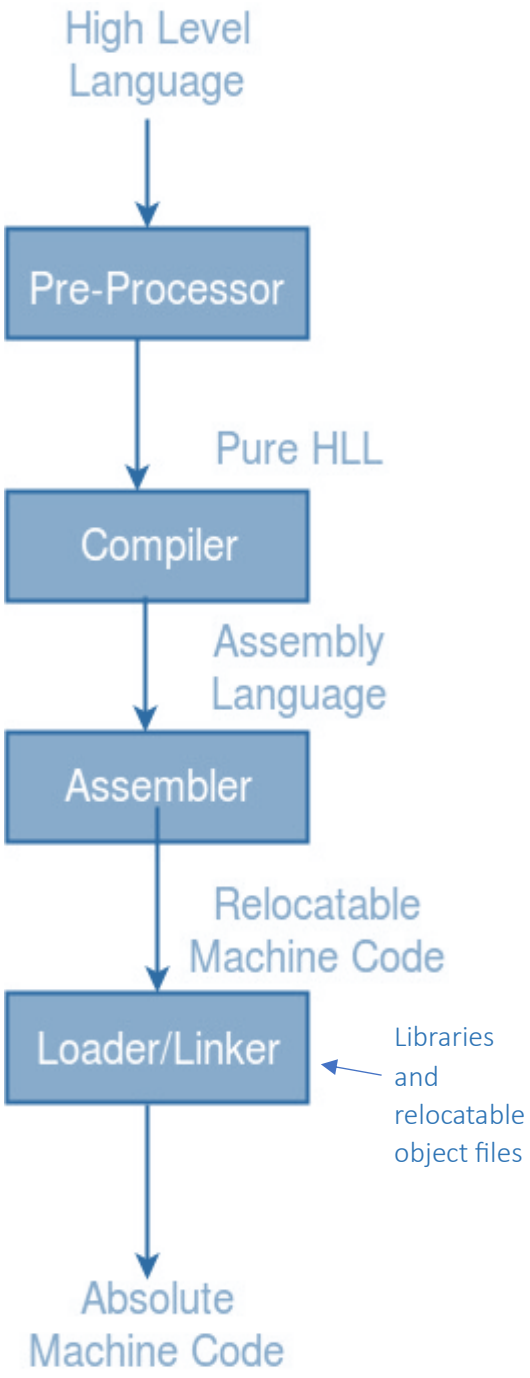
```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter a number: ");
        int number = scanner.nextInt();

        System.out.println("You entered: " + number);
    }
}
```



1. **Compile**: The Java compiler ('javac') converts the code to bytecode.
2. **Run**: The Java Virtual Machine (JVM) executes the bytecode.
3. **Input**: The user is asked to enter a number.
4. **Process**: The entered number is read and saved.
5. **Output**: The application displays "You entered: [number]".

Compilation within the program execution process

To the right you can see the full sequence of the program execution where compilation is the second step.



In general, the procedure is as follows:

Preprocessor directives such as #include and #define are commonly used in high-level language programs such as those written in C. They are closer to human language, but need to be translated so they can understand them These (#) tags are referred to as preprocessor directives and tell the pre-processor what to do.

1. **Pre-processor:** It works on HLL code and processes the directives by replacing #include with the contents of included files. This step produces "pure" high-level source code that is free of these directives and suitable for the next stage of translation.

2. **Compiler:** It converts this "pure" high-level language source code into assembly language. It does lexical analysis to break down the code into tokens, syntax analysis to organize these tokens into a syntax tree, semantic analysis to guarantee logical consistency, optimization to refine the code, and code generation to generate the matching assembly language code. (Talked about in more details in the next task).

3. **Assembler:** It processes the assembly code (intermediate state between the high-level language and machine code) into relocatable machine code (binary representation of the program) that may be executed.

4. **Linker/loader:** takes the relocatable machine code, along with any libraries or other object files that the program depends on and resolves all address references to produce an absolute machine code file. This file is the executable that can be run on the computer.

The result is complete machine code that is ready to be executed by the machine. It contains all of the memory addresses and is written in binary format, which the computer's CPU understands and can execute immediately.

To understand this fully, let's look at the program execution process of an example source code.

```c
#include <stdio.h>

#define A 10

int main() {
    int number = A;
    printf("The number is: %d\n", number);
    return 0;
}
```

1. The preprocessor will process the directives. #include <stdio.h> will include the contents of the <stdio.h> file, which includes the printf function definition. #define A 10 will substitute the letter A with the number 10 wherever it appears.

```c
// Contents of stdio.h (simplified)
extern int printf(const char *format, ...);

int main() {
    int number = 10; // A has been replaced with 10
    printf("The number is: %d\n", number);
    return 0;
}
```

2. The compiler will transform the preprocessed c code to assembly.

```
section .data
    format db "The number is: %d", 10, 0
    number dd 0

section .text
    global _start

_start:
    ; Assign value to number
    mov eax, A
    mov [number], eax

    ; Print the number
    push dword format
    push dword [number]
    call printf
    add esp, 8

    ; Exit the program
    mov eax, 1
    xor ebx, ebx
    int 0x80
```

3. The assembler turns assembly language into machine code, which is still not executable since it contains references to symbols such as printf that are not yet associated with their real memory addresses.

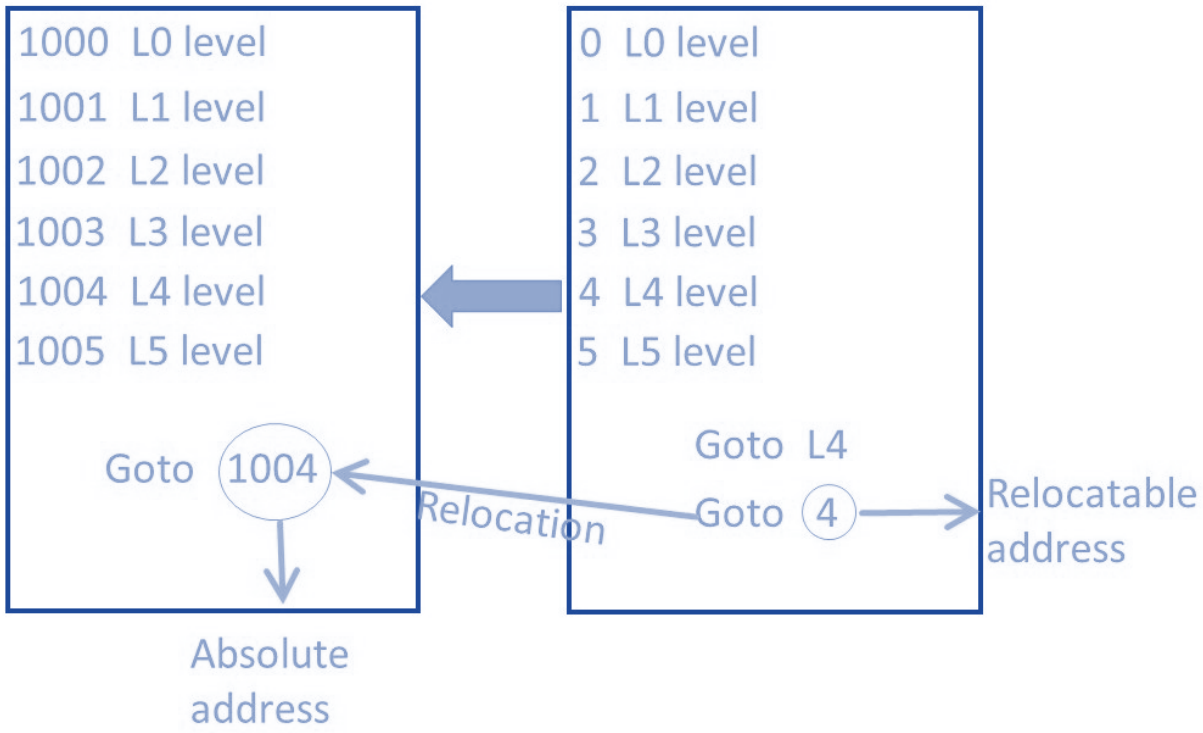
   Note: The relocatable machine code is binary and therefore unreadable by humans, but it contains information such as:

   - Instructions in binary.
   - The printf function is still not linked to its actual code in the C library; it's just a reference at this point.

4. Linker/Loader: The linker resolves the printf reference by scanning the C library for the real machine code and linking it with the relocatable machine code. It also specifies all variables' and functions' final memory addresses. The result is a file that can be executed.

   Note: The executable file is also binary and not human-readable, but it contains:

   a. Completely resolved computer instructions with absolute addresses.
   b. All necessary library functions and system calls.



Example of relocatable machine code vs absolute machine code.

Note: this is not connected to the example and is simply to clarify the difference between relocatable and absolute machine code.

The internal phases of the compilation process

Now that we've looked at the execution process of a program, let's look more closely into the compiler's parts and all the stages.

Compilation consists of two parts:

1. **Analysis** (front end): determines the operations of the source program, which are stored in a tree structure. (Breaking down) it is machine independent.
2. **Synthesis** (back end): takes the tree structure and translates the operations within it into the target program. (Collection) it is machine dependent,

To make it clearer here's a practical example:

Consider baking a cake, where the recipe is like a source program. Analysis is like reading and understanding a recipe, breaking each step down into simple activities such as measuring flour and beating eggs. This is like how a computer examines code, structuring it into a tree structure for easier processing. Synthesis, on the other hand, is like baking the cake using the prepared ingredients, following the recipe. In programming, this is the process by which the computer takes the ordered code and converts it into a working program, much as combining ingredients and processes results in a cooked cake.

These are the internal phases of the compilation process:

1. **Lexical analyzer (analysis):**

   It is the compiler's first phase. It examines the source code character by character (character stream), identifying essential units known as tokens, removing spaces and can tell errors like invalid characters. Tokens are the fundamental building elements of programming languages, and they might comprise keywords, operators, identifiers, and literals.

   The analyzer examines the source code during this phase and recognizes lexemes, which are character sequences that create specific tokens. For example, if the lexeme (+) appears in the source code, the analyzer recognizes it as an operator token indicating addition. The tokens, together with their related information, are then recorded in a symbol table which is a data structure that contains information about the identifiers of the program. Following that, the tokens are sent on to the next phases of the compiler.

   Let's assume we have this source code snippet:

```
int x = 5;
char y = "hello";
int sum = x + y;
```

   The analyzer would identify the following tokens (keywords, operators, identifiers):

int	x	=	5	;	char	y	=	hello	;	int	sum	=	x	+	y	;

Splitting the source code into tokens allows parsing and semantic analysis to work with a structured representation of the program, which aids understanding and translation of the code into executable instructions.

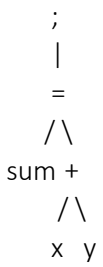
2. Syntax analyzer (analysis):

   Tokens are passed to a syntax analyzer, which analyzes them based on the language grammar and organizes them into a hierarchical structure, often represented as a parse or an abstract syntax tree (AST) if the syntax is correct. This ensures that the source code satisfies the programming language's grammatical rules.

   Ex. This line of the code

```
int sum = x + y;
```

   Will generate this AST:



   Which shows that this line follows the C grammar.

3. Semantic analyzer (analysis):

   It uses the syntax tree from the previous phase, as well as the symbol table from the lexical analyzer and examines the relationships between the AST's elements to ensure that the given source code is semantically consistent. The Semantic Analyzer will look for type mismatches, incompatible operands, scope analysis, an undeclared variable, and so on.

   Examples:

   1) Type mismatch: The operands in an expression must be compatible with the operator being used. In an arithmetic operation, such as addition (+), the operands on both sides must be of compatible type, such as both integers or floating-point values.

   2) Variable declaration: If you want to use the variable x in your program, you must declare it first, as in int x; similarly, you can't use a function without declaring it first.

3) Array bound checking: Array access must be within the array's limits. So, if you declare an array int arr[5];, you can only access elements between arr[0] and arr[4]. Accessing arr[5] or above would result in unpredictable behavior.

Let's look at the code snippet again:

```
int x = 5;
char y = "hello";
int sum = x + y;
```

The syntax checker (parser) will form the AST since it is only concerned with the arrangement and combination of tokens in the program. There are, however, semantic problems in the code that a syntax checker cannot discover.

1) Type incompatibility: In the line char y = "hello";, we assign the string literal "hello" to the char variable y resulting in a type mismatch.

2) Addition of incompatible types: in the line int sum = x + y;, we are attempting to add variables x and y of different types (integer variable, character) which is not allowed in C.

These problems would go undiscovered without a semantic checker during the syntax checking step, and potentially resulting in unexpected behavior or runtime issues. If a semantic error is discovered during this analysis, the semantic checker may continue to create the AST until the error is found. It can locate the fault in the AST and provide an error message detailing the nature of the issue.

4. Intermediate code generation (analysis):

Following syntax and semantic analysis of the source code, the compiler generates an intermediate code representation of the source code. This code is independent of the target machine architecture, should be quick and simple to produce and easy to translate into the target machine.

Example of a code:

```
total = count + rate * 5
```

After this phase:

```
t1 := int_to_float(5)
t2 := rate * t1
t3 := count + t2
total := t3
```

5. Code optimization (synthesis);

The compiler optimizes the intermediate code in order to increase the efficiency and speed of the produced executable. Such as deleting unneeded code lines (such as unreachable code and unused variables) and rearranging the sequence of statements to speed up program execution without wasting resources. The major purpose of this step is to improve the intermediate code so that it runs faster and takes up less space.

Example of a code:

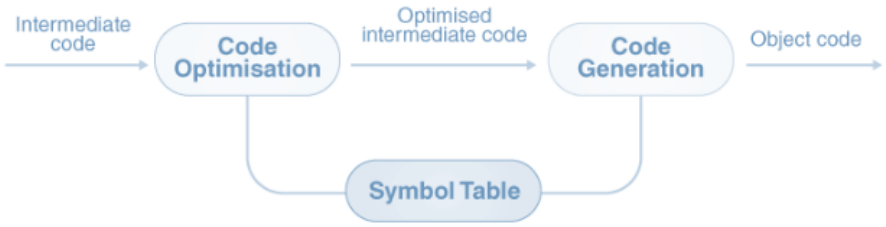
```
a = intofloat(10)
b = c * a
d = e + b
f = d
```

Can be optimized to:

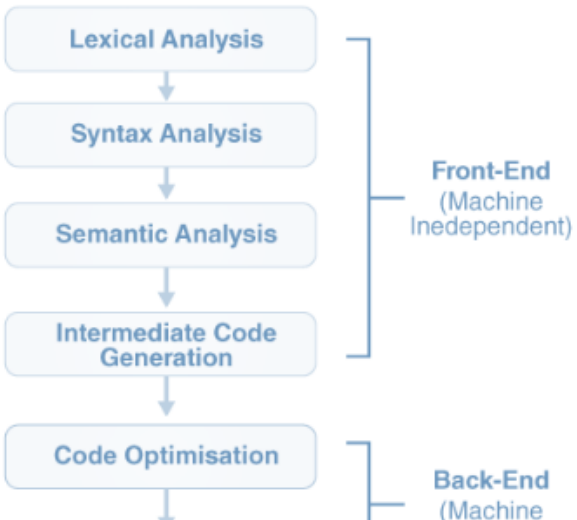
```
b =c * 10.0
f = e+b
```

6. Code generation (synthesis);

The optimized intermediate code is translated by the compiler into machine code or another low-level representation (specific to the target architecture) creating the final executable code that can execute on the target computer. It also sets aside memory for the variable.



Overall process diagram:



The lexical analyzer

Lexical analyzer role in terms of its inputs and outputs

Input: The lexical analyzer first scans the program source code's raw input, which is a stream of characters that includes letters, numbers, symbols, and other sorts of whitespace (e.g. spaces and newlines).

This input stream is processed by the lexical analyzer in a series of steps:

1. Scanning: Before tokenization, the lexical analyzer performs several operations to clean and organize the character stream:
    - Deletion of comments: Comments will not be executed (only intended for developers) so the lexical analyzer recognizes and removes comment patterns (such as // for single-line comments or /*... */ for block comments in C) from the code.
    - Whitespace compaction: Consecutive whitespace characters are used in source code to improve readability; however, they have no impact on code execution, so the lexical analyzer reduces these to a single space.

3. Tokenization: The cleaned code is an input in this stage and is transformed into tokens. A token is a string that has a meaning that has been assigned to it. It is organized as a pair of a token name and an optional token value. Tokenization follows these steps:
    - Pattern Matching: The lexical analyzer recognizes character sequences in the input string using patterns defined by regular expressions. These patterns determine what identifiers, keywords, literals, operators, and punctuation are valid tokens for the language.
    - Token Creation: The lexical analyzer generates a token after a pattern is recognized. Example, if the pattern matches a sequence of alphabetic characters followed by digits (like var1), it will consider it an identifier and output a token appropriately.
    - Symbol Table Entries: The lexer then adds symbol table entries for some tokens, such as identifiers and literals. This table contains data regarding variable names, data types, scope, and other details which will be utilized later in the compilation process.

Output: The lexical analyzer produces a stream of tokens, which is then sent on to the next phase of the compiler which is the parser (syntax analyzer). The parser uses these tokens to build a syntax tree, which describes the program's grammatical structure.

Note: The lexical analyzed should handle an error if it finds an invalid character or sequence of characters that does not match any known pattern. For example, it might ignore it, give an error message, or attempting to recover by locating the nearest valid token are all possible error handling options.

Data structures, and techniques to build.

Data structures:

Transition table: These are data structures used in the development of deterministic finite automata (DFA) in lexical analyzers for token recognition. A transition table represents the DFA's states and transitions between them based on input characters. Each row is a state, and each column is an input. Based on the current state and the input character, the table entries suggest the next state.

Buffers: Used to temporarily store data. A typical technique is double buffering, which includes employing two buffers in code it can can be simply implemented as two variables. Buffers enable lookahead which is a technique used in lexical analysis to preview upcoming characters without consuming them, enabling accurate differentiation and interpretation of similar-looking sequences.

Tokens follow the form: <token-name, attribute-value>.

Hash tables: Symbol-Table stores a record for each variable name and fields for the name's attributes (characteristics). Attributes can be name, type, scope, for procedure names attributes include number and types of arguments, by value/referance, type returned, etc. Tokens can be stored in the hash table since an entry in a hash table maps a key (unique entry) to a value which works well with the form of tokens as mentioned above. Moreover, the hash table data structure provides fast access time with a time complexity of O(1) which is suitable for tokens that are regularly read in compilers. It can efficiently handle scalability.

I used an input stream to read the data written on the file. I also tokenized this file and saved it in a tokens array list so that I can pass these tokens .

Techniques:

Regular Expressions to DFA: Most lexical analyzers define token patterns using regular expressions. For efficient token recognition, these are transformed into deterministic finite automata (DFA).

Hand-Coding: Sometimes, lexical analyzers are hand-coded (ex. simple languages) using programming languages such as C or Java.

Buffer Management: Efficient buffer management is critical for handling the source code stream to provide a constant and efficient flow of data. A technique that is usually used is double-buffering where two buffers are used together: while one buffer is being processed, the other is filled with the next character. This method reduces wait times for disk I/O operations and assures a continuous data feed for the analyzer. Additionally, it is used to achieve character lookahead to aid in token recognition, which might require knowing the next characters.

Error handling: Lexical analyzer should detect, report, and recover from errors, to ensure parsing reliability. The analyzer must discover these errors and produce descriptive error messages that specify the position of the issue in the source code or recover from them.

Its advantages and disadvantages

Advantages:

1. Speed: Lexical analyzers are often fast and can process source code quickly, contributing to the compilation's overall speed.
2. Simplification for parsing: Lexical analysis enhances parsing performance by breaking down the input into smaller, more manageable tokens. This allows the parser to focus on the code structure rather than individual characters. Moreover, it creates and populates the symbol table, which is required for later stages like parsing, semantic analysis and code generation.
3. Flexibility: Lexical analysis allows the use of keywords and reserved terms which makes the development of new programming languages easy (or modifying current ones).
4. Error Detection: 4. Error Detection: The lexical analyzer may detect errors like missing semicolons, invalid characters, undefined variables, and other simple grammatical mistakes which can save time throughout the debugging process and early on.
5. Code Optimization: lexical analyzer can detect common patterns and replace them with more efficient code which optimizes code and possibly boosts the program's performance.

Disadvantages:

1. Complexity: Lexical analysis can be complicated and computationally intensive which can make implementation hard in some programming languages. To add, determining the token can be difficult at times and may necessitate lookahead or backtracking.
2. Limited error detection: While lexical analysis may detect certain faults, it cannot detect all of them. For example, it is incapable of detecting logic and type errors.
3. Increased code size: The addition of keywords and reserved words will make the code larger and more difficult to read. Moreover, for very simple scripting or programming languages, the overhead of a lexical analysis phase may be unnecessary.
4. Maintenance Issues: If the syntax of the programming language changes, the lexical analyzer must be updated, which can be a complex and error-prone procedure.

Why its separated from the other parts of the phases

1) Modularity: This modular approach to compiler design, in which each module performs a specialized role, makes the process more efficient. Because of the task separation of the lexical analyzer, the parser can work with a clean stream of tokens that is independent of the complexity of the initial text processing. Furthermore, partitioning allows for separate debugging and testing of modules, making it easier to discover and resolve errors within certain portions of the compiler without affecting the overall system.

2) Simplicity and efficiency in parsing: Lexical analysis provides a sequence of tokens rather than raw text which is critical in speeding the parsing process. This distinction is especially important for LL(1) and LR(1) parsers, which rely on a single lookahead token to make parsing decisions.
   Parsers would have to deal with both raw text and tokenization if lexical analysis was not available which increases complexity and could involve the use of more than one lookahead token, complicating parsing algorithms making them inefficient.

3) Optimization: Having a dedicated lexical analyzer which can be fine-tuned and optimized for the purpose of reading input text and recognizing tokens can potentially improve compiler performance.

4) Portability and reusability: The design of lexical analyzers, which are created to be independent of specific language syntax, considerably improves portability and reusability. Because of this independence, they are portable among compilers. Furthermore, lexical analyzers may handle a wide range of system-specific text formats, such as non-standard symbols and other character encodings (e.g., UTF8), allowing such complications to be kept isolated from the rest of the compiler. This separation not only helps the compilers' capacity to adapt to new languages, but it also contributes to the system's general portability.

   ??Furthermore, within the lexical analyzer, input-device-specific characteristics can be properly handled and restricted, further optimizing the process and ensuring that these features do not interfere with other portions of the compiler.

5) Error handling: separating lexical analysis makes it easier to manage errors at different stages of the compilation process related to character and token formation, while parsers can focus on syntactical errors.

Example to demonstrate how it works.

Let's consider this input in c:

```
/* Calculate the distance traveled */
distance = speed * time + 0.5 * acceleration * time * time;// final calculation
```

1. **Scanning phase**: The block comment /* Calculate the distance traveled */ at the beginning of the line would be removed and the line comment // final calculation at the end of the line would also be removed. Also, all sequences of more than one whitespace character between tokens would be compacted into a single space.

   This is the result of the scanning phase of the compiler:

   ```
   distance = speed * time + 0.5 * acceleration * time * time;
   ```

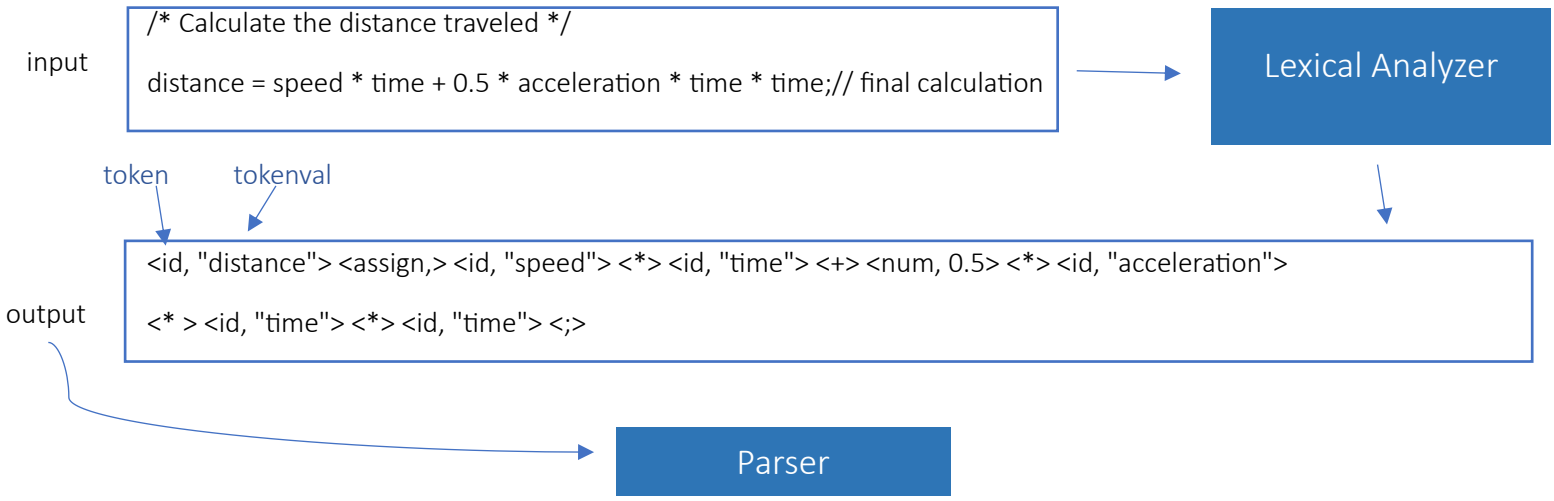
2. **Tokenization phase**: The lexical analyzer will read the input from cleaning phase and classify it as follows:
   - Identifies distance, speed, time, and acceleration as identifiers.
   - It recognizes =, *, and + as operators used for assignment, multiplication, and addition, respectively.
   - Identifies 0.5 as a numeric constant.
   - Identifies (;) as a delimiter.

Output:

<id, "distance"> <assign,> <id, "speed"> <*> <id, "time"> <+> <num, 0.5> <* > <id, "acceleration">

<*> <id, "time"> <*> <id, "time"> <; >

Overall:



Relationship between lexical analyzer, source language, finite state automata and regular expression.

Well define all concepts first, the **lexical analyzer**, as explored in previous parts of the report, it is a component of a compiler that breaks down source code into tokens (smallest units with semantic meaning in a programming language). The **Source Language** is the programming language that is currently being compiled. The source language in my example is C. The **regular expression** is a sequences of characters that define a pattern which are used for pattern matching (identifying tokens in source code). Lastly the **Finite state automata** is a mathematical model of computation used in lexical analyzers to recognize patterns in the source code (tokens) in other words they are recognizers; they simply say yes or no about each possible input string. It can be represented as an NFA or DFA.

Now, let's examine the relationships using an example:

Source language: C

An identifier is a name in C that is used to identify a variable, function, or other user-defined entity. In C, an identifier should begin with a letter (either uppercase or lowercase) or an underscore and can be followed by letters, numbers, or underscores.

Regular Expression for C Identifier: According to this rule, the regular expression for an identifier in C is: (L|_)(L|D|_)* where L denotes an uppercase or lower case letter, D denotes Digit.

In another way: [a-zA-Z_][a-zA-Z0-9_]*

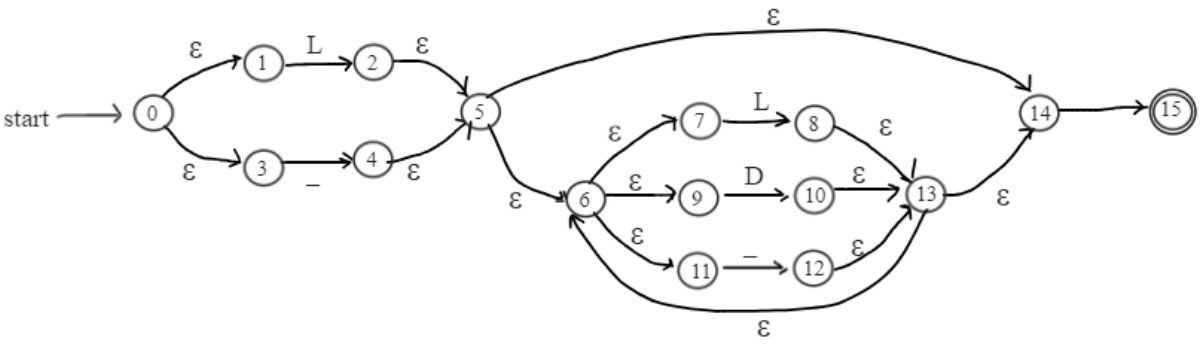
- [a-zA-Z_]: The identifier must start with any uppercase or lowercase letter or an underscore.
- [a-zA-Z0-9_]: After the first character, the identifier can have any combination of letters, digits, and underscores. Where the asterisk denotes zero or more occurrences of the preceding element.

Valid input based on the regular expression:

We can choose the underscore from the first parentheses then form the second parentheses we can get L then L then _ then D then again D → _Ab_13

We transform the RE **(**L|_)(L|D|_)* into the finite state automata to NFA/DFA to identify whether the token is a valid identifier or not:

NFA:



Now let's test the input "_Ab_13" on the NFA to test whether it is a valid identifier or not. First, we go from state 0 to 3 through epsilon then state 3 to 4 through _ then 4 to 5 through epsilon the 5 to 6 through epsilon then 6 to 7 through epsilon the 7 to 8 through L then 8 to 13 through epsilon then 13 to 6 through epsilon then 6 to 7 through epsilon the 7 to 8 through L then 8 to 13 through epsilon then 13 to 6 through epsilon then 6 to 11 through epsilon then 11 to 12 through _ then 12 to 13 through epsilon then 13 to 6 through epsilon then 6 to 9 through epsilon then 9 to 10 through D then 10 to 13 through epsilon then 13 to 6 through epsilon then 6 to 9 through epsilon then 9 to 10 through D then 10 to 13 through epsilon then 13 to 14 through epsilon then 14 to 15 through epsilon which is a final state. This proves that the input is valid as checked by the NFA. We know that this input is valid because the

Below is The **Code** that matches regular expression: In my example, it is a piece of Java code designed to identify identifiers in C code. The code reads characters one by one and checks if they form valid identifiers.

```
int readChar;
while ((readChar = reader.read()) != -1) {
    ch = (char) readChar;

    // check for identifiers (starts with letter or underscore)
    if (Character.isLetter(ch) || ch == '_') {
        id.append(ch);
        ch = (char) reader.read();

        // can be followed by letter digit or underscore
        while (Character.isLetter(ch) || Character.isDigit(ch) || ch == '_') {
            id.append(ch);
            ch = (char) reader.read();
        }

        id.delete(0, id.length());
    }
}
```

The Java code snippet is a practical example which reads characters and creates identifiers that begin with a letter or underscore and is followed by any combination of letters, digits, or underscores. This corresponds to the regular expression defined above.

The source language (C in this case) defines the standards for what constitutes a valid identifier. This rule is expressed using the regular expression. The lexical analyzer carries out this logic in order to tokenize the source code. The code written in java in my example acts as the lexical analyzer. And the FSA checks if an identifier is valid against the language or not.

In general, to process the source language, the lexical analyzer employs FSA and regular expression, identifying and tokenizing valid patterns to identify identifiers, keywords, literals, and so on.

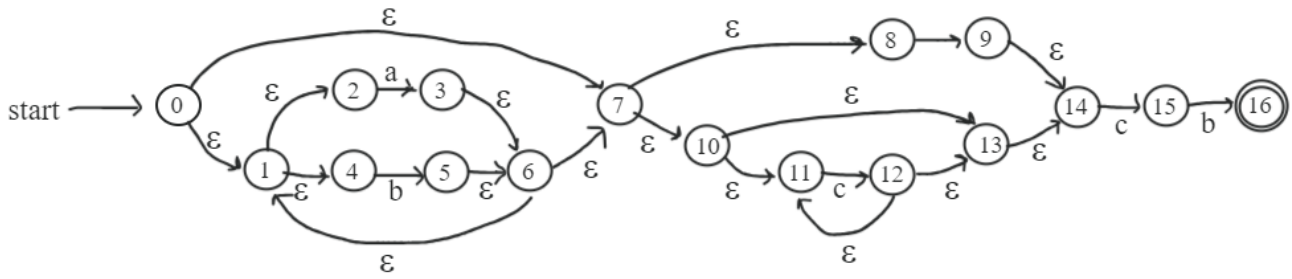
Conversion

Between regular expression to nondeterministic finite automata to deterministic finite automata.

RE:

(a|b)*(d|c*)bc

RE to NFA:



NFA to DFA:

1) Initialize DFA state set (e-closure of start state)

E-closure: {0,1,2,4,7,8,10,11,13,14}

2) For each DFA state, ech input (a,b) determin NFA states reachable and find e-closure (move, e-closure functions)

E-closures and moves of DFA state {0,1,2,4,7,8,10,11,13,14}

E-closure: {0,1,2,4,7,8,10,11,13,14} State
move({0,1,2,4,7,8,10,11,13,14},a) → {3}
E-closure({3}) →{1,2,3,4,6,7,8,10,11,13,14} State
move({0,1,2,4,7,8,10,11,13,14},b) → {5,15}
E-closure({5,15})→{1,2,4,5,6,7,8,10,11,13,14,15} State
move({0,1,2,4,7,8,10,11,13,14},c) → {12}
E-closure({12}) →{11,12,13,14} State
move({0,1,2,4,7,8,10,11,13,14},d) → {9}
E-closure({9}) →{9,14} State

E-closures and moves of DFA state {1,2,3,4,6,7,8,10,11,13,14}

move({1,2,3,4,6,7,8,10,11,13,14},a) → {3}
E-closure already calculated
move({1,2,3,4,6,7,8,10,11,13,14},b) → {5,15}
E-closure already calculated
move({1,2,3,4,6,7,8,10,11,13,14},c) → {12}
E-closure already calculated
move({1,2,4,5,6,7,8,10,11,13,14},d) → {9}
E-closure already calculated

E-closures and moves of DFA state {1,2,4,5,6,7,8,10,11,13,14,15}

move({1,2,4,5,6,7,8,10,11,13,14,15},a) → {3}
E-closure already calculated
move({1,2,4,5,6,7,8,10,11,13,14,15},b) → {5,15}
E-closure({5,15}) →{1,2,4,5,6,7,8,10,13,14,15} State
move({1,2,4,5,6,7,8,10,11,13,14,15},c) → {12,16}
E-closure({12,16}) → {11,12,13,14,16} State
move({1,2,4,5,6,7,8,10,11,13,14,15},d) → {9}
E-closure already calculated

E-closures and moves of DFA state {11,12,13,14}

move({11,12,13,14},a) → {}
move({11,12,13,14},b) →{15}
E-closure({15}) →{15} State
move({11,12,13,14},c) → {12}
E-closure already calculated
move({11,12,13,14},d) → {}

E-closures and moves of DFA state {9,14}

move({9,14},a) → {}
move({9,14},b) → {15}
E-closure already calculated
move({9,14},c) → {}
move({9,14},d) → {}

E-closures and moves of DFA state {11,12,13,14,16}

move({11,12,13,14,16},a) → {}
move({11,12,13,14,16},b) → {15}
E-closure already calculated
move({11,12,13,14,16},c) → {12}
E-closure already calculated
move({11,12,13,14,16},d) → {}

E-closures and moves of DFA state {15}

move({15},a) → {}
move({15},b) → {}
move({15},c) → {16}
E-closure({16}) → {16} State
move({15},d) → {}

E-closures and moves of DFA state {16}

move({16},a) → {}
move({16},b) → {}
move({16},c) → {}
move({16},d) → {}

3) identify DFA states, accept states.

A: {0,1,2,4,7,8,10,11,13,14}

B: {1,2,3,4,6,7,8,10,11,13,14}

C: {1,2,4,5,6,7,8,10,11,13,14,15}

D: {11,12,13,14}

E: {9,14}

F: {11,12,13,14,16} (accept)

G: {15}

H: {16} (accept)

4) Draw transition table

DFA states	a	b	c	d
A	B	C	D	E
B	B	C	D	E
C	B	C	F	E
D	-	G	D	-
E	-	G	-	-
F	-	G	D	-
G	-	-	H	-
H	-	-	-	-

4) Construct DFA from transition table

RE to DFA

RE: (same as RE to DFA)
(a|b)*(d|c*)bc

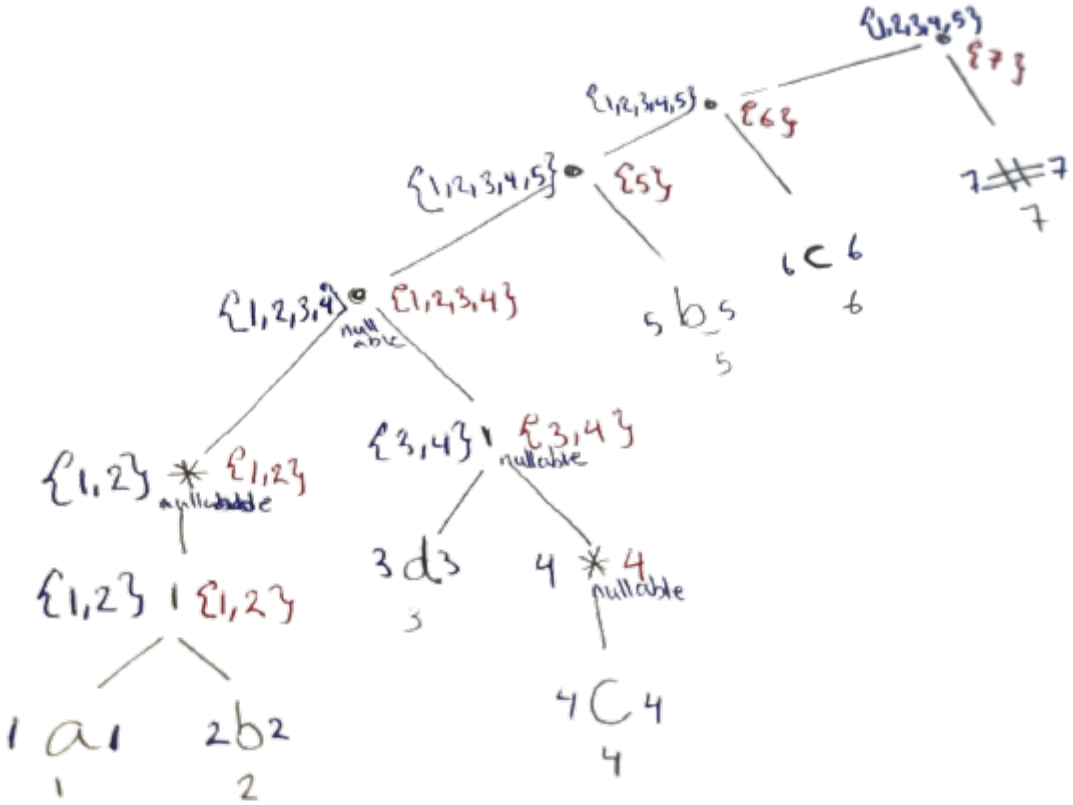
Augmented RE:

(a|b)*.(d|c*).b.c.#

Nodes numbered:

(a|b)*.(d|c*).b.c.#
1 2 3 4 5 6 7

Build syntax tree then calculate first position, last position, and nullable function from leaf to node:



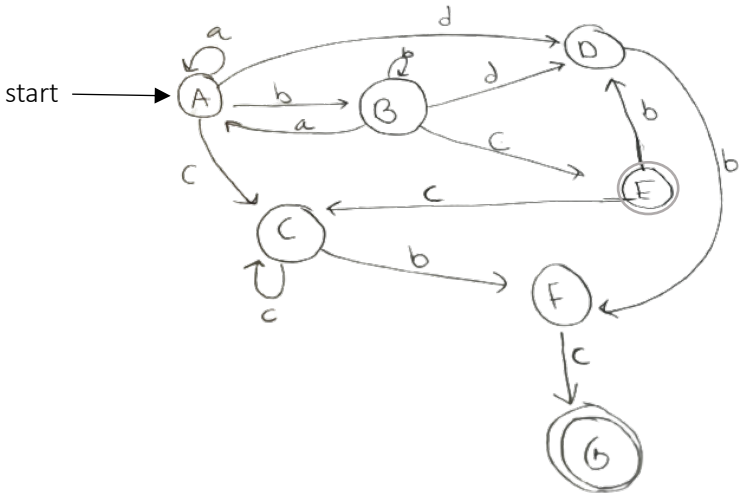
Calculate follow position:

	Node	Follow position
a	1	{1,2,3,4,5}
b	2	{1,2,3,4,5}
d	3	{5}
c	4	{4,5}
b	5	{6}
c	6	{7}
#	7	{}

DFA states transition table:

DFA	a	b	c	d
A = {1,2,3,4,5}	{1,2,3,4,5} = A	{1,2,3,4,5,6} = B	{4,5} = C	{5} = D
B={1,2,3,4,5,6}	{1,2,3,4,5} = A	{1,2,3,4,5,6} = B	{4,5,7} = E	{5} = D
C={4,5}	-	{6} = F	{4,5} = C	-
D={5}	-	{6} = F	-	-
E={4,5,7} **final**	-	{5} = D	{4,5} = C	-
F={6}	-	-	{7} = G	-
G={7} **final**	-	-	-	-

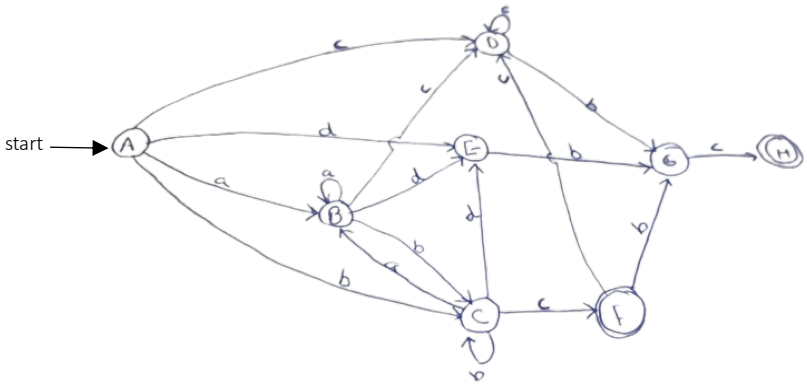
Draw DFA:



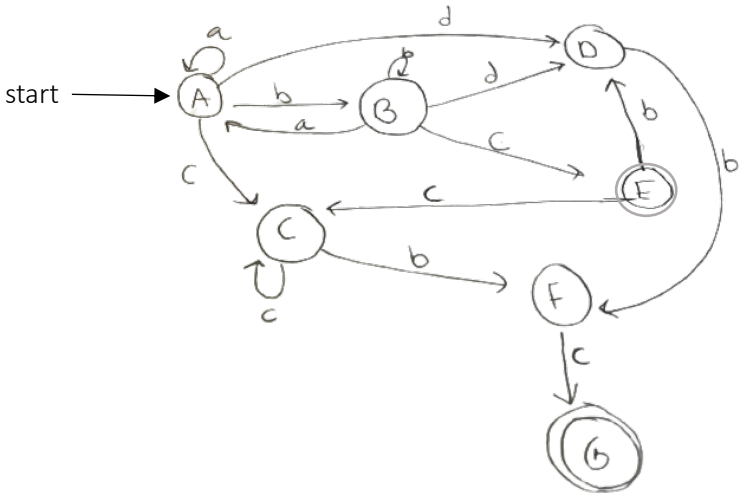
Note that the two DFAs test the validity of an input string to the same regular expression, however the states and the number of these states are different.

Generating and testing input strings using both DFAs produced.

RE: (a|b)*(d|c*)bc



DFA 1



DFA 2

Trying to generate aabbc:

DFA 1: From node **A** to **B** to **B** to **C** to **C** to **F**

DFA 2: From node **A** to **A** to **B** to **B** to **E**

I was able to start input at start node and end it at a final node. Both DFAs generated the aabbc input.

Test valid input: abaabcccbc

DFA 1: A $\xrightarrow{a}$ B $\xrightarrow{b}$ C $\xrightarrow{a}$ B $\xrightarrow{a}$ B $\xrightarrow{b}$ C $\xrightarrow{c}$ F $\xrightarrow{c}$ D $\xrightarrow{c}$ D $\xrightarrow{b}$ G $\xrightarrow{c}$ H

DFA 2: A $\xrightarrow{a}$ A $\xrightarrow{b}$ B $\xrightarrow{a}$ A $\xrightarrow{a}$ A $\xrightarrow{b}$ B $\xrightarrow{c}$ E $\xrightarrow{c}$ C $\xrightarrow{c}$ C $\xrightarrow{b}$ F $\xrightarrow{c}$ G

I was able to start input at start node and end it at a final node, so the input is valid for both DFAs.

Test invalid input: ababdccbc

DFA 1: A $\xrightarrow{a}$ B $\xrightarrow{b}$ C $\xrightarrow{a}$ B $\xrightarrow{b}$ C $\xrightarrow{d}$ E $\xrightarrow{c}$ NO TRANSITION. Invalid.

DFA 2: A $\xrightarrow{a}$ A $\xrightarrow{b}$ B $\xrightarrow{a}$ A $\xrightarrow{b}$ B $\xrightarrow{d}$ D $\xrightarrow{c}$ NO TRANSITION. Invalid.

The Parser

Role of the parser in terms of its inputs and outputs

Input: The syntax analyzer (parser) receives a string of tokens from the previous stage (lexical analyzer) which breaks down the source code into tokens.

This input stream is processed by the lexical analyzer in a series of steps:

1. **Verifies string of tokens:** The parser should verify if the string of tokens passed can be formed by the source language's grammar (grammatically correct). This is done by comparing the tokens to the syntax rules of the language.

2. **Reports syntax errors:** It will report syntax errors if it finds a string of tokens that does not correspond to the grammar. The parser is also in charge of resolving common problems so that the compilation process can continue (error handling) which is critical for giving the programmer useful feedback.

3. **Constructs parse tree:** It may generate a parse tree depending on the type of parser. Top-down parsers begin at the root and progress to the leaves, whereas bottom-up parsers begin at the leaves and progress to the root.

It also performs several other tasks:

1. It collects and saves information about tokens (e.g. type and scopes) in the symbol table, which is later used for semantic checks and code generation.
2. To verify that operations are executed on compatible types, the parser does type checking.
3. The parser can generate intermediate representation (lower-level representation of the source code) using syntax-directed translation methods. This intermediate representation could be in many forms:
    a. Abstract Syntax Trees (ASTs) which as its name suggests is a tree representations of the source code's abstract syntax.
    b. Control-flow Graphs (CFGs) which are graphs that represent the sequence in which various pieces of code are executed, frequently accompanied by three-address code.
    c. Different Levels of IR: Some compilers, such as the SGI Pro64, may support many levels of IR, each of which serves a different level of abstraction and optimization.

The parser is built as a recognizer for strings generated by context-free grammar (CFG). It acts as a syntax checker, reports syntax errors (e.g. misplaced semicolons), performs semantics checking (type checking of expressions, functions, etc.) and generates an intermediate representation (IR).

Output for the next step: Depending on the compiler's design, the AST (syntax tree) or another intermediate representation (IR) is produced by the parser and sent on to the next step of the compiler, which could be semantic analysis or an intermediate code generation phase.

Data structures, and techniques to build.

Data structures:

Parse Trees: This is the data structure generated by parsers. It is a tree that shows the syntax structure of the source code.

Abstract Syntax Trees (ASTs): A simplified parse tree that omits certain elements in syntax that are useless for semantic analysis.

Stacks: Used in shift-reduce parses (bottom-up parser) that keeps track of symbols and states during parsing.

Tables: The parsing tables are required for table-driven parsers (such as LR and shift reduce parsers). They drive the parsing process by specifying which rule to use based on the current state and the next input token.

Symbol table: it is a table where the parser saves important information such as the scope and type which is used for later semantic check stage.

Lexical Tokens: Since the parser works with the lexical analyzer's output, it should understand the tokens produced by the lexical.

Techniques:

Top-down parsing techniques:

Recursive descent: It processes the input using a collection of recursive operations. Each non-terminal in the grammar is usually associated with a procedure.

LL: is a table-driven parsing approach where the action is determined by a parsing table.

Bottom-up parsing techniques:

LR: It reads the input from left to right and produces a rightmost derivation in reverse.

SLR, LALR, and CLR: They are variants of the LR parser having distinct methods of constructing the parsing table, each with its own set of performance and capability trade-offs.

Error recovery techniques: Techniques such as panic mode, phrase level recovery, error productions, and global correction that allow the parser to continue processing when an error has occurred.

Management of symbol table: since the parses interacts with the symbol table data structure.

Its advantages and disadvantages

Advantages:

1. Error Detection: Parsers can detect syntax mistakes in source code and provide documentation to the programmer which is critical for debugging and guaranteeing the correctness of the code.
2. Provides structure: Parsers give a structured representation of the source code by generating a parse tree or an abstract syntax tree, which is necessary for further stages of the compilation.
3. Enabling optimization: Because the output provided by the parser is structured which allows for optimization techniques in next stages of compilation increasing the efficiency of the generated code.
4. Makes translation easier: makes the translation to an intermediate representation or straight into target code easier.
5. Semantic analysis: Some parsers integrate with semantic analysis to allow early identification of certain semantic mistakes, such as type incompatibilities.

Disadvantages:

1. Complexity: Creating a parser for complex grammar is difficult especially for languages with a lot of exceptions.
2. Performance: It is time-consuming, especially when dealing with big source files or handling more sophisticated grammars.
3. Error recovery: depending on the error recovery plan, it might not be suitable in some cases and might change the goal of the code.
4. Maintenance: As the language specification changes, the parser must be updated, which can be time-consuming.
5. Handling ambiguity: Some grammars are ambiguous, making it difficult for the parser to know which rule to apply. There are techniques to deal with ambiguity, such as utilizing a more advanced parsing strategy which can make the parser more complex.

Why its separated from the other parts of the phases

1. Modularity: Separating the parser from the rest of the compiler is suitable since each module performs a specific function making the compiler is easy to understand, maintain, and adapt.

2. Abstraction: The parser deals with the source code's syntactic structure, changing a flat sequence of tokens into a hierarchical structure without having to worry about features like semantic analysis or code generation.

3. Reusability: The same parsing component can be used in several compilers for different languages that share the same syntactic structures, with only minor modifications.

4. Efficiency: By separating the parser, the compiler can resolve grammatical mistakes before moving on to more time-consuming tasks like semantic analysis and optimization, saving resources.

5. Parallel development: Different teams can work on different phases of the compiler at the same time, which can speed up the development process.

6. Testing: Since each phase has its responsibilities, issues can be identified and resolved within that phase.

7. Less complication: avoid mixing different types of logic and data structures, which could complicate the compiler's design.

8. Grammar modification: The grammar of the language can be adjusted independently without affecting the lexical, semantic analysis or code generation phases.

In summary, separating the parser from other compiler phases contributes to a cleaner, more efficient, and more maintainable compiler design. It allows each phase to focus on a specific aspect of the compilation process, which helps manage the complexity inherent in translating human-readable code into machine-executable instructions.

Example to demonstrate how it works.

Let's take this example in c:

```
speed= distance/ time;
```

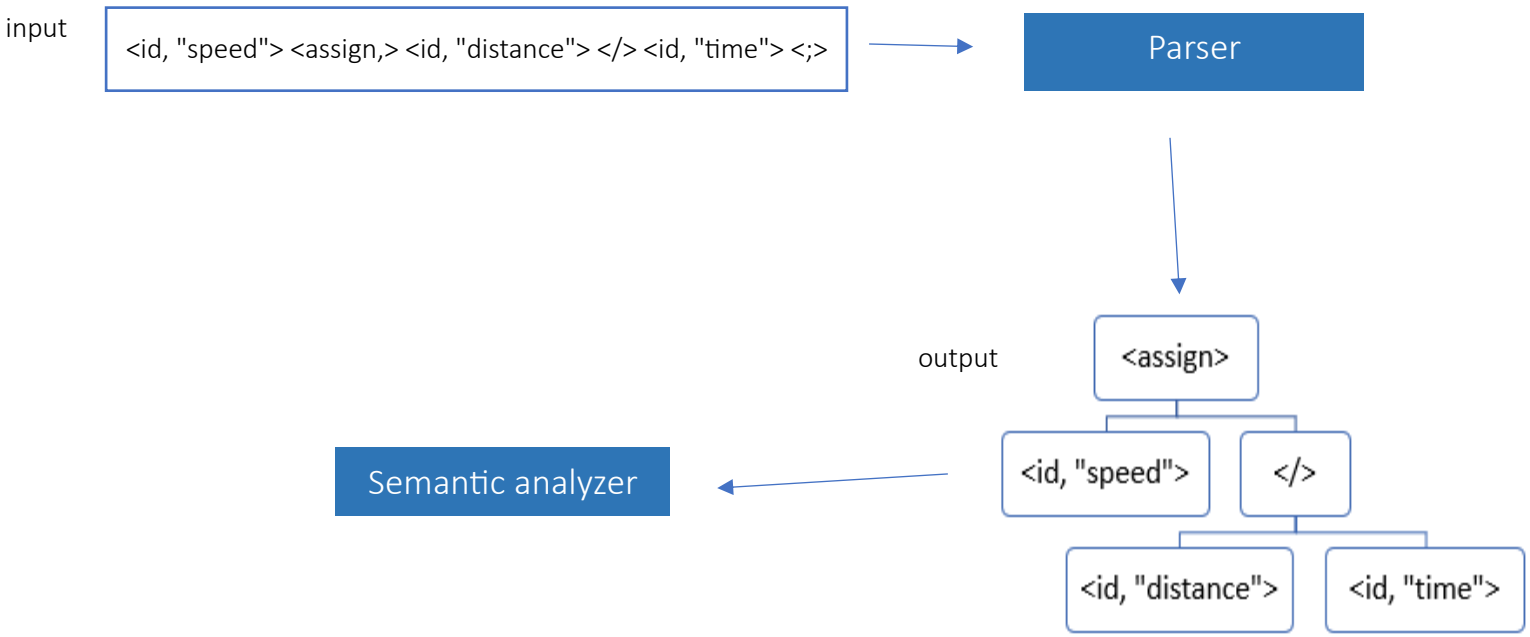
The lexical analyzer produces the output stream which is the input of the parser:

```
<id, "speed"> <assign,> <id, "distance"> </> <id, "time"><;>
```

This input stream is processed by the lexical analyzer which verifies the string of tokens, reports syntax errors (none will be found here as it is grammatically correct) and construct the parse tree if it is grammatically correct.

It also collects and saves information about tokens (e.g. type and scopes) in the symbol table, verifies operands and operations are compatibility.

Overall:



The relationships between the parser and the grammar for the source language.

Overall

Grammar is the set of rules that define the structure of a programming language. This includes syntax rules, which control how statements are produced, as well as semantic rules, which determine what these statements mean. Essentially, grammar is the blueprint that explains how valid expressions, or code statements, can be constructed in a certain programming language. The parser (which uses the top-down/bottom up techniques) is responsible for evaluating and interpreting source code written in a programming language. Its major function is to process the code and determine whether it conforms with the defined grammar of the language. This involves breaking down the code into tokens (lexical analysis) and then creating a parse tree or abstract syntax tree to reflect the code's syntactic structure (syntax analysis). The parser's relationship with the grammar is interdependent. The parser uses the grammar to guide its examination of the source code. It uses the grammar as a reference to check the code's structure, ensuring that it follows the language's syntax and semantic rules. If the code deviates from these guidelines, the parser usually flags errors, informing the programmer that changes are required or does some type of error recovery. However, the complexity and clarity of the grammar have a significant impact on a parser's performance. A well-defined and unambiguous grammar enables more efficient parsing. Grammar that are overly complicated or contain ambiguities can cause parsing issues, such as the difficulty to determine which grammatical rule applies in specific instances (parsing conflicts ex. shift reduce conflict). Furthermore, the grammar's design influences the parsing technique used. For example, context-free grammars are commonly employed in programming languages, depending on the grammar's characteristics, different parsing techniques, such as top-down or bottom-up parsing, may be more effective. Furthermore, the relationship between the parser and the grammar is always changing. Grammar updates or extensions are frequently required as programming paradigms evolve and new language capabilities are introduced. As a result, parsers must be modified or redesigned to accommodate these changes, ensuring that they can correctly read the language's developing syntax and semantics.

Below is an example showing the relationships between the parser and the grammar for the source language.

Grammar

Let's assume the grammar which is a subset of the arithematic precedency rules in C encompassing addition, subtraction, multiplication, division, modulus, postfix and prefix increment and decrement of ids.

Grammar:

Addition production: E → E + E

Multiplication production: E → E * E

Subtraction production: E → E- E

Division production: E → E / E

Parenthesis production: E → (E)

Identifier production: E → id

Postfix increment E → id ++

Postfix decrement E → id--

Prefix increment E → ++ id

Prefix decrement E →-- id

However, the above grammar does not explicitly define the precedence rules. In a context-free grammar, precedence is often handled through the production structure. For example, to guarantee that multiplication and division have precedence over addition and subtraction, we use different non-terminals for high-precedence and low-precedence operations, such as this:

E → E + T | E- T | T

T → T * F | T / F | T % F | F

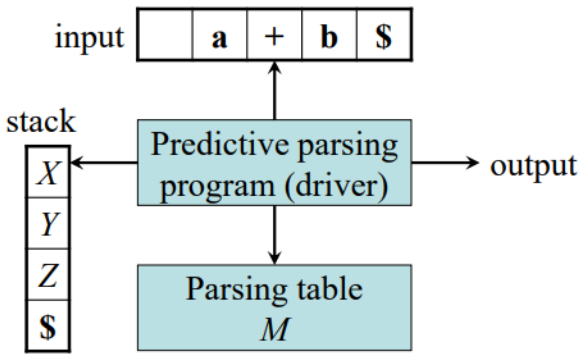
F → (E) | I

I → id | id ++ | id-- | ++ id |-- id

From this context free grammar that describes the syntax of the arithematic operations, we can generate the following valid statement:

1. Start with E, apply E → T so E becomes T
2. Apply T → F: T becomes F.
3. Apply F → (E): F becomes (E).
4. Inside the parentheses, apply E → T again: We get (T).
5. Apply T → F inside the parentheses: We get (F).
6. Apply F → I inside the parentheses: we have (I).
7. Apply I → id: It becomes (id)

The parser employs techniques such as top-down and bottom-up parsing to determine whether an expression matches the language's specific grammar rules. A prominent technique is the "top-down left to right leftmost derivation with one lookahead (LL(1))." This method employs a parsing table, in which rows correspond to non-terminals, columns to terminals, and intersections indicate the applicable production rule. In addition, a stack is employed to navigate the parsing process. The purpose is to build a parse tree from the top down, starting with the start symbol and proceeding sequentially through the input string.



We need to eliminate grammar ambiguity (it can produce several parse trees for the same sentence). Two significant solutions for resolving this issue are to eliminate left recursion and left factoring.

First Left-recursion: This happens when a production refers to itself on the left side of, which can result in infinite recursion in top-down parsing methods. This is the grammar after elimination of left recursion:

E → T E'

E' → + T E' | - T E' | ε

T → F T'

T' → * F T' | / F T' | % F T' | ε

F → (E) | I

I → id | id ++ | id-- | ++ id | -- id

Second Left-factoring: This happens when the beginning of several productions for a nonterminal are the same, causing ambiguity and making it impossible for a top-down parser to determine which production to use without reading further into the input.

E → T E'

E' → + T E' | - T E' | ε

T → F T'

T' → * F T' | / F T' | % F T' | ε

F → (E) | I

I → id I' | ++ id | -- id

I' → ++ | -- | ε

Third, we will calculate the first and the follow for each non-terminal.

Non-terminal	First	Follow
E	{(,id,++,--}	{$,)}
E’	{+,-, ε}	{$,)}
T	{(,id,++,--}	{+,-,$,)}
T’	{*,/,%, ε}	{+,-,$,)}
F	{(,id,++,--}	{*,/,%,+,-,$,)}
I	{id,++,--}	{*,/,%,+,-,$,)}
I’	{++,--, ε}	{*,/,%,+,-,$,)}

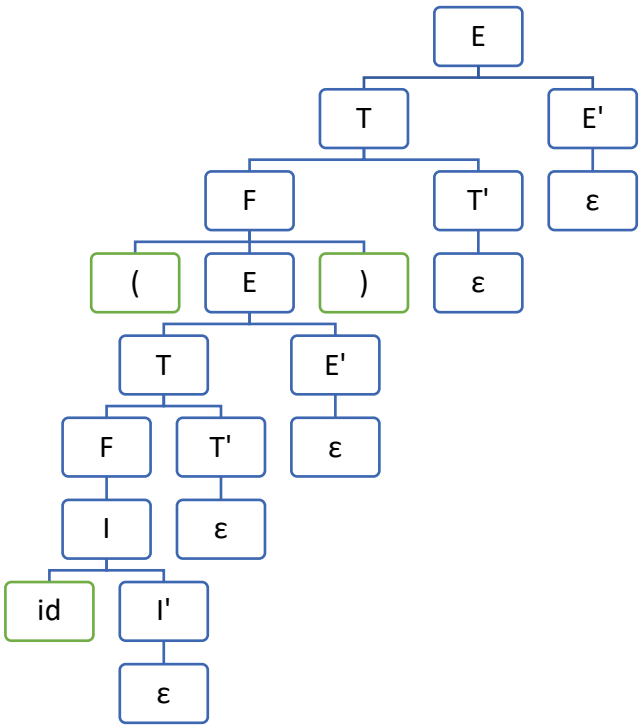
Fourth, based on this we will fill the predictive parsing table:

	+	-	*	/	%	(	)	id	++	--	$
E						E → T E'		E → T E'	E → T E'	E → T E'	
E'	E' → + T E'	E' → - T E'					E' → ε				E' → ε
T						T → F T'		T → F T'	T → F T'	T → F T'	
T'	T' → ε	T' → ε	T' → * F T'	T' → / F T'	T' → % F T'		T' → ε				T' → ε
F						F → (E)		F → I	F → I	F → I	
I								I → id I'	I → ++ id	I → -- id	
I'	I' → ε	I' → ε	I' → ε	I' → ε	I' → ε		I' → ε		I' → ++	I' → --	I' → ε

Fifth, Driver: Now to check if the input string for example (id) is valid, we can parse it:

Stack	Input	Production Rule
$ E	(id) $	
$ E' T	(id) $	E → T E'
$ E' T' F	(id) $	T → F T'
$ E' T') E (	(id) $	F → (E)
$ E' T') E	id) $	POP MATCH
$ E' T') E' T	id) $	E → T E'
$ E' T') E' T' F	id) $	T → F T'
$ E' T') E' T' I	id) $	F → I
$ E' T') E' T' I' id	id) $	I → id I'
$ E' T') E' T' I'	) $	POP MATCH
$ E' T') E' T'	) $	I' → ε
$ E' T') E'	) $	T' → ε
$ E' T')	) $	E' → ε
$ E' T'	$	POP MATCH
$ E'	$	T' → ε
$	$	E' → ε
ACCEPTED		

Sixth, parse tree built:



I wrote a Java implementation of the fourth and fifth step of the LL(1) parser for the basic arithmetic grammar of the c language (grammar specified above) to check the inputs. In the code, the parser parses expressions using a predictive parsing table. Let's look at the code's essential components:

To implement the driver (5th step) we need to reference the predictive parsing table created in the 4th step. The parsing table is a two-dimensional string array (String [][] table) that represents the grammatical rules, with each cell defining a distinct production rule that associates a non-terminal with a terminal. The arrays (nonTers) and (terminals) are responsible for storing the grammar's non-terminals and terminals and are used to index into the parsing table, which was done manually in the driver stage.

```
//Table of rules
String [][] table=
{
    {null,null,null,null,null,"TP",null,"TP","TP","TP",null},
    {"+TP","-TP",null,null,null,null,"",null,null,null,""},
    {null,null,null,null,null,"FB",null,"FB","FB","FB",null}   ,
    {"","","*FB","/FB","%FB",null,"",null,null,null,""},
    {null,null,null,null,null,"(E)",null,"I","I","I",null},
    {null,null,null,null,null,null,null,"iK","vi","wi",null},
    {"","","","","","",null,"","v","w",""}
};

//set of terminals and non terminals to get rule
String [] nonTers={"E","P","T","B","F","I","K"};
String [] terminals={"+","-","*","/","%","(",")","i","v","w","$"};
```

LL(1) parsers make parsing decisions using a single lookahead symbol, which implies they only look one symbol ahead of the input at a time. That is why the terminals and non-terminals should be one character:

Non-terminals: E' is P , T' is F, I' is K

Terminals: id is i, ++ is v, -- is w (This change is handled in the read function)

Stack and Input Initialization: The parser initializes with a stack containing the start symbol (E in this case) and the input string (the code to be parsed). The stack is used to keep track of the symbols that need to be processed.

```
//input
public String input="";

//will go through the input
private int indexOfInput=-1;

//Stack
Stack <String> strack=new Stack<String>();

 push(this.input.charAt(0)+"");/
 push("E");
```

The read() function handles the retrieval and conversion of tokens (for example, handling increment and decrement operators).

```
private String read() {
    indexOfInput++;

    char ch = this.input.charAt(indexOfInput);
    String str = String.valueOf(ch);

    // Check for i++
    if (ch == '+' && indexOfInput + 1 < input.length()) {
        if (input.charAt(indexOfInput + 1) == '+') {
            str = "v";//set str to the correct terminal defined
            //skip the next +
            indexOfInput++;
        }
    }
    // Check for ++i
    else if (ch == '+' && indexOfInput - 1 >= 0) {
        if (input.charAt(indexOfInput - 1) == '+') {
            str = "v"; //set str to the correct terminal defined
            // Do not increment (looking backwards so index correct)
        }
    }
    //check for i--
    else if (ch == '-' && indexOfInput + 1 < input.length()) {
        if (input.charAt(indexOfInput + 1) == '-') {
            str = "w"; //set str to the correct terminal defined
          //skip the next -
            indexOfInput++;
        }
    //check for i++
    } else if (ch == '-' && indexOfInput - 1 >= 0) {
        if (input.charAt(indexOfInput - 1) == '-') {
            str = "w";//set str to the correct terminal defined
         // Do not increment (looking backwards so index correct)
        }
    }

    return str;
}
```

The main algorithm

The parser repeatedly removes the top symbol from the stack and compares it to the current input symbol. If the top symbol is a non-terminal, the parser uses the parsing table to determine which rule to apply. This rule is then pushed onto the stack in reverse order (with the leftmost symbol on top). If the top symbol represents a terminal, it is compared to the current input symbol. If they match, the input pointer advances to the next symbol; otherwise, an error is returned. This operation will continue until the stack is empty (successful parsing) or an exception occurs (unsuccessful parsing).

Grammar and Parsing

Grammar in Parser Design

A Programming Language Grammar is a set of instructions for writing statements that are correct for the programming language. The instructions are in the form of rules that govern how characters and words can be arranged one after the other to make acceptable statements (also known as sentences). In compiler design, a language's grammar is often defined using context-free grammars (CFGs). A CFG is a set of production rules that specify how terminal and non-terminal symbols can be combined to generate sequences in the language. The choice of grammar significantly influences the parsing strategy.

Context free grammar is often represented as a 4-tuple. G={N,T,P,S}. Where T is a finite set of tokens (terminals), N is a finite set of nonterminal, P is a finite set of productions of the form $\alpha \rightarrow \beta$ and S belongs to the set of non-terminals and is a designated start symbol. For the grammar mentioned in the previous task:

N: {E,T,E',F,T',I,I'}

T:{+,-,*,/,%,(,),++,--}

P: { E → T E'

E' → + T E'

E' → - T E'

E' → ε

T → F T'

T' → * F T'

T' → / F T'

T' → % F T'

T' → ε

F → (E)

F → I

I → id I'

I → ++ id

I → -- id

I' → ++

I' →--

I' → ε}

S : {E}

The grammar of a language acts as a blueprint for the parser, directing it in detecting valid strings (code) and building the matching parse trees. By adhering to this grammar, the parser can successfully comprehend and analyze the code, recognizing not only the fundamental elements but also their relationships.

Parsing Techniques

Top-Down Parsing:

Top-down parsing is an approach that begins with the grammar's highest-level rule (the start symbol or root) and seeks to turn it into the desired input string. This transformation is accomplished by applying production rules in a way that tries to predict the structure of the supplied input string without first inspecting its content. Top-down parsers construct parse trees from the root down to the leaves. Top-down parsers are categorized into two types: recursive descent and non-recursive descent.

Recursive descent: also known as the Brute force parser or the backtracking parser. It employs the grammar's production rules, aiming to match the input string. Basically, it creates the parse tree using brute force and backtracking. They are easy to implement making them an appropriate choice for simple language grammars, as they are simple to construct and comprehend. These parsers are also adaptable and expandable, allowing simple changes to accommodate new or extended grammars. However, there are some significant drawbacks. Recursive descent parsers have difficulty with left-recursive grammars, resulting in endless recursion, and can be expensive since it will keep backtracking a lot in complex grammars. Furthermore, their predictive skills are limited, making them unsuitable for increasingly complicated grammars, especially when lookahead is restricted.

Non-recursive descent also known as LL parsing is a more advanced kind of top-down parsing. It scans the input from left to right and generates the sentence's leftmost derivative. The notation "LL(k)" defines the number of lookahead tokens used by the parser to make syntax tree decisions, where "k" is the number of tokens looked ahead. The parser utilizes a parsing table to determine which rule to apply based on the current input token and the stack's top position. It uses a parsing table to generate the parse tree instead of backtracking. They can handle a wider range of grammars, including those requiring lookahead, resulting in more deterministic and efficient parsing for grammars. Furthermore, they often offer greater error detection and reporting capabilities. They do, however, share the disadvantage of recursive descent parsers in that they cannot handle left-recursive grammars without transformation. The number of lookahead tokens used by an LL parser influences its effectiveness but also makes it more complicated. Furthermore, LL parsers are not suitable for grammars with a high level of ambiguity.

Bottom-Up Parsing:

The parser constructs the parse tree for the given input string using grammatical productions by compressing the terminals, i.e. the parse tree is built from the input symbols and works its way up to the grammar's start symbol. It applies the reverse of the rightmost derivation. Bottom-up parsers are categorized into two types: shift-reduce parsers and LR parsers.

Shift-reduce: This approach has two primary operations: shifting and reducing. Shifting involves moving symbols from the input buffer to a stack. In reduction, a sequence of symbols on the stack that corresponds to the right side of a production rule is replaced by the non-terminal on the left side of the rule. This process continues until the grammar's start symbol is derived, indicating that the input string has been parsed successfully. Shift-reduce parsers can parse a wider range of grammars than top-down parsers and are particularly good at managing left-recursive grammar rules, which are sometimes a challenge for top-down parsers. However, shift-reduce parsers are naturally more complicated. They are more likely to meet parsing conflicts like reduce-reduce and shift-reduce.

LR Parsing: it is an advanced form of shift-reduce parsing it reads input from left to right and produces a rightmost derivation. The 'k' in LR(k) parsing represents the number of lookahead tokens utilized to make parsing decisions. LR parsers are well-known for their ability to efficiently parse a wide range of grammar by handling lookahead tokens in a systematic manner. LR parsers are known for their efficiency and robustness, allowing them to parse practically all forms of programming language grammar. They are especially adept at handling complex syntax while providing exact error reporting. However, their biggest problem is implementation; creating LR parsing tables, particularly for canonical LR parsers, is hard and resource intensive. This complexity might make it difficult to implement and understand, especially if done manually.

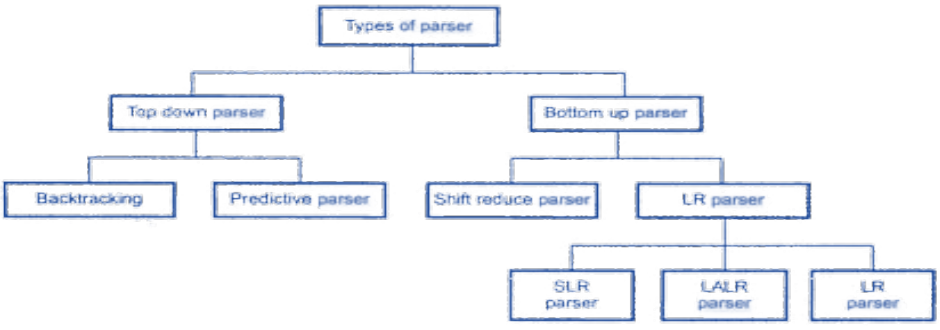
Variations of LR parsing include:

SLR (simple LR): This is a reduced version of LR parsing and only includes reductions according to the follow (explained later). It constructs its parsing table using a simpler way, resulting in lower memory requirements which also reduces the power of full LR parsers.

LR (1): LR(1) represents left-to-right parsing with a single token lookahead. It is an extension of the LR parsing approach. LR(1) parsers make parsing decisions using a single lookahead token, which helps reduce ambiguities that simpler parsers such as SLR can encounter. LR(1) parsing tables are often larger than SLR or LALR, but less than CLR (Canonical LR).

LALR (Lookahead LR parsing): LALR parsing is a compromise between SLR and CLR parsing. It provides more power than SLR by adding more lookahead information while retaining a more manageable table size.

CLR: This is the most extensive and effective method of LR parsing. It can handle a broader range of grammars with more precision, but it requires much larger parsing tables, making it more complex and memory intensive.



Comparison of techniques through solving an example:

Example: Below is the grammar for the logical operations in c including AND (&&) and OR (||)

B → B || M | M

M → M && N | N

N → ! N | (B) | bool

Top down:

LL(1)

Step 1: check and eliminate left recursion.

B → B || M | M

M → M && N | N

Grammar after elimination of left recursion :

B → M B'

B' → || M B' | ε

M → N M'

M' → && N M' | ε

N → ! N | (B) | bool

Issue: LL(1) can't handle left recursion or left factoring.

Step 2: check and eliminate left factoring. This happens when there are grammatical rules that begin with the same prefix, which might cause complications in LL(1) parser.

No left factoring in this grammar.

Step 3: generate first and follow.

Nonterminal	FIRST	FOLLOW
B	{!,(,bool}	{$,)}
B'	{||, ε }	{$,)}
M	{!,(,bool}	{||,$,)}
M'	{&&, ε }	{||,$,)}
N	{!,(,bool}	{&&,||,$,)}

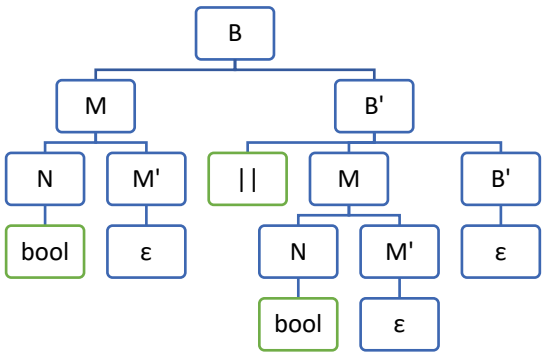
Step 4: Predictive parsing table:

	||	&&	!	(	)	bool	$
B			B → M B'	B → M B'		B → M B'	
B'	B' → || M B'				B' → ε		B' → ε
M			M → N M'	M → N M'		M → N M'	
M'	M' → ε	M' → && N M'			M' → ε		M' → ε
N			N → ! N	N → (B)		N → bool	

Step 5: Driver

Stack	Input	Production Rule
$ B	bool || bool $	
$ B' M	bool || bool $	B → M B'
$ B' M' N	bool || bool $	M → N M'
$ B' M' bool	bool || bool $	N → bool
$ B' M'	|| bool $	POP MATCH
$ B'	|| bool $	M' → ε
$ B' M ||	|| bool $	B' → || M B'
$ B' M	bool $	POP MATCH
$ B' M' N	bool $	M → N M'
$ B' M' bool	bool $	N → bool
$ B' M'	$	POP MATCH
$ B'	$	M' → ε
$	$	B' → ε
ACCEPTED		

Step 6: Draw parse tree.



Bottom up:

Shift-reduce:

B → B || M | M

M → M && N | N

N → ! N | (B) | bool

Stack	Input	Action
$	bool || bool $	shift
$ bool	|| bool $	reduce N → bool
$ N	|| bool $	reduce M → N
$ M	|| bool $	reduce B → M
$ B	|| bool $	shift
$ B ||	bool $	shift
$ B || bool	$	reduce N → bool
$ B || N	$	reduce M → N
$ B || M	$	reduce B → B || M
$ B	$	Accept

This method like LL(1) causes shift-reduce conflict and/or reduce-reduce conflict. For example, this input (bool || bool) may cause shift-reduce conflict because the parser may not immediately known whether to apply the rule N-> bool to reduce the bool or to shift the || token and wait for more input. This issue is resolved by backtracking which causes overhead which is why we opt to other options like LR, SLR, LALR.

Left Reduce(0):

Grammar: B → B || M | M

M → M && N | N

N → ! N | (B) | bool

Step 1: Find augmented grammar:

r0 B'-> B

r1 B-> B || M

r2 B-> M

r3 M-> M && N

r4 M-> N

r5 N-> (B)

r6 N-> ! N

r7 N-> bool

Step 2: I_0, closure and got to (using . operator) on initial state

B'-> .B

B-> .B || M

B-> .M

M-> .M && N

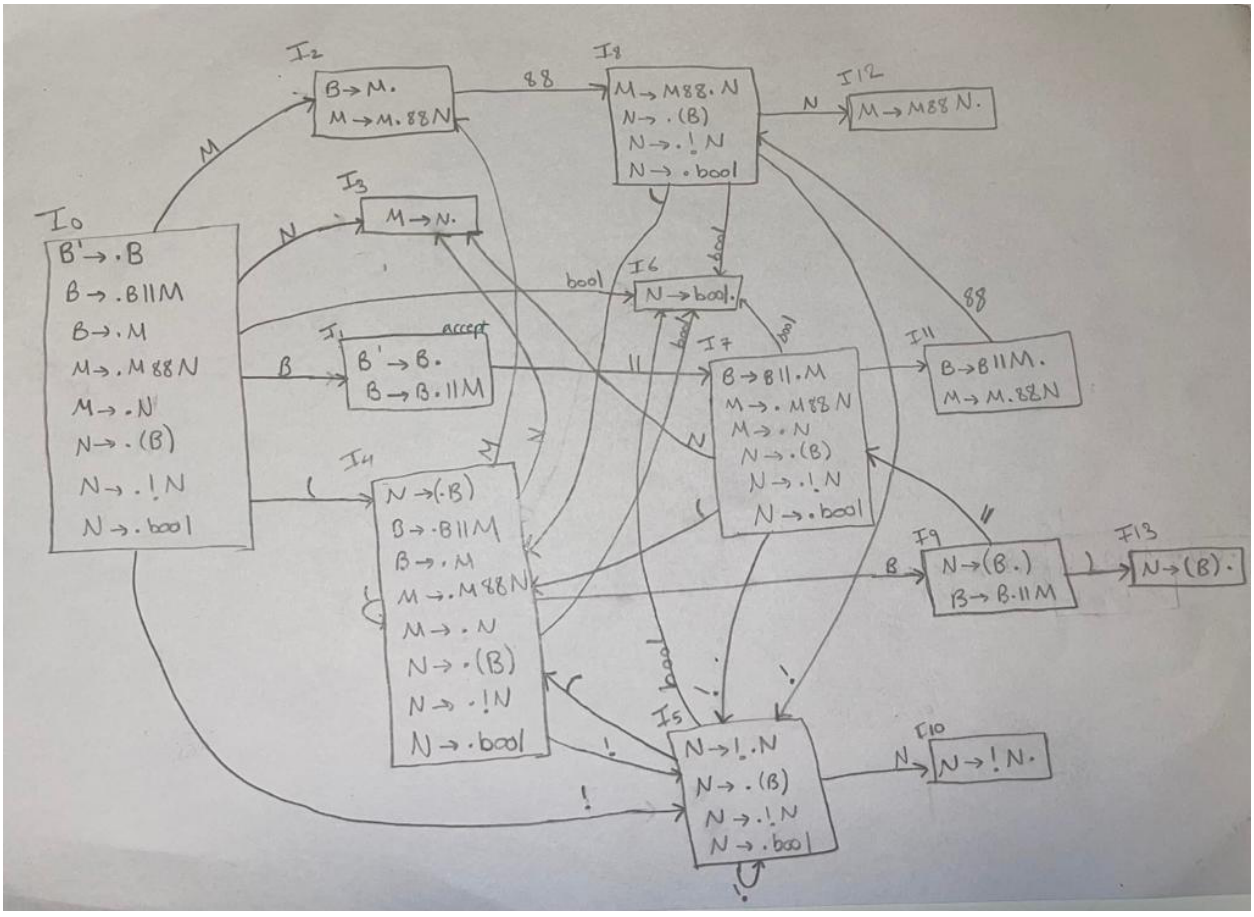
M-> .N

N-> .(B)

N-> .! N

N-> .bool

Step 3: Calculate Go-to and closure functions for each state.



Step 4: build the parser table based on the graph shown above.

State	Terminals (Actions)							goto			
	||	&&	(	)	!	bool	$	B'	B	M	N
0			s4		s5	s6			1	2	3
1	s7						accept				
2	r2	s8	r2	r2	r2	r2	r2				
3	r4	r4	r4	r4	r4	r4	r4				
4			s4		s5	s6			9	2	3
5			s4		s5	s6					10
6	r7	r7	r7	r7	r7	r7	r7				
7			s4		s5	s6				11	3
8			s4		s5	s6					12
9	s7			s13							
10	r6	r6	r6	r6	r6	r6	r6				
11	r1	s8	r1	r1	r1	r1	r1				
12	r3	r3	r3	r3	r3	r3	r3				
13	r5	r5	r5	r5	r5	r5	r5				

Notice: reductions are applied to all terminals. T

This method handles left recursion and left elimination, and in the case of this grammar there is no shift-reduce or reduce-reduce conflicts. which shows that the grammar is unambiguous and well-defined for LR(0) parsing. While LR(0) is adequate for these grammars, I chose to apply SLR which only takes a few extra steps. This is because SLR parsers provide better flexibility by handling a wider range of grammars, which is especially useful for potential future grammar modifications. It also has enhanced error reporting and recovery capabilities, which are critical in real-world applications to improve user experience. Furthermore, it provides memory and computational efficiency optimizations.

SLR:

For the same grammar to remove the issue of reduction being applied on all non-terminals, we need to calculate the follow for the non-terminals and use them to specify under which non-terminals to apply reductions.

Nonterminal	Follow
B'	{$}
B	{$,||,)}
M	{$,||,&&,)}
N	{$,||,&&,)}

Here is the updated table:

State	Terminals (Actions)							goto			
	||	&&	(	)	!	bool	$	B'	B	M	N
0			s4		s5	s6			1	2	3
1	s7						accept				
2	r2	s8		r2			r2				
3	r4	r4		r4			r4				
4			s4		s5	s6			9	2	3
5			s4		s5	s6					10
6	r7	r7		r7			r7				
7			s4		s5	s6				11	3
8			s4		s5	s6					12
9	s7			s13							
10	r6	r6		r6			r6				
11	r1	s8		r1			r1				
12	r3	r3		r3			r3				
13	r5	r5		r5			r5				

Let's test the same input, bool||bool:

Stack	Input	Action
0	bool || bool $	s6
0 bool 6	|| bool $	r7
0 N	|| bool $	go to 3
0 N 3	|| bool $	r4
0 M	|| bool $	go to 2
0 M 2	|| bool $	r2
0 B	|| bool $	go to 1
0 B 1	|| bool $	s7
0 B 1 || 7	bool $	s6
0 B 1 || 7 bool 6	$	r7
0 B 1 || 7 N	$	go to 3
0 B 1 || 7 N 3	$	r4
0 B 1 || 7 M	$	go to 11
0 B 1 || 7 M 11	$	r1
0 B	$	go to 1
0 B 1	$	accept

As we can see this grammar is unambiguous and doesn't lead to any shift reduce conflicts (no shit/reduce in the same cell) so we don't need to use other more advanced parsing techniques like LR(1) left reduce with 1 lookahead or the LALR.

Overall comparison:

Starting with the LL(1) parsing strategy, which examines the input from left to right and generates a leftmost derivation with one lookahead token, we run into obvious problems. LL(1) parsers are not designed to handle left recursion, as demonstrated by the products B-> B || M and M-> M && N in the above language. This creates an infinite loop in the parsing process. To make this grammar compatible with an LL(1) parser, I applied restructuring to remove left recursion and left factoring to resolve ambiguities. The shift-reduce technique was more capable of handling left recursion than LL(1) parsers. However, it experiences shift-reduce and reduce-reduce conflicts. While they improve upon LL(1) in terms of recursion, their conflict resolution capabilities remain limited and did not fully accommodate the complexities of the grammar. To decide between reductions or between reducing and shifting, I had to do a lot of backtracking and trials which took a lot of time despite the input tested was short. This problem is further complicated if it is to be coded as backtracking overhead is high and machines don't have the logic of humans. To try and avoid the shift-reduce or reduce-reduce conflicts, I opted to using Left to right in reverse (rightmost techniques) and its variations. I started by employing the LR(0) while it was simple to implement (compared to other LR variations), it has limitations that make the SLR (Simple LR) parsing method a better option in many cases. The main problem with LR(0) is that it offers reductions to all terminals, which increases the possibility of ambiguity and false parsing actions. This method cannot distinguish between different contexts where a reduction should occur, which frequently results in inaccurate parses in grammars with more sophisticated structures (results in many reduce-reduce and shift-reduce conflicts, particularly in grammar with a high level of complexity or ambiguity). In contrast, SLR improves on LR(0) by guiding the reduction process using the follow for the set of non-terminals. This means that SLR reductions are only executed when the next symbol in the input string is in the follow set of the grammar's non-terminal on the left side of the reduction rule. As a result, SLR parsing tables have fewer conflicts and can handle a broader range of grammar than LR(0) parsers, making them more robust and versatile in practical applications.

The above example doesn't have a shift reduce conflict, now let's consider another grammar to demonstrate the need for LR(1) and LALR.

SLR to prove shift reduce conflict of grammar

Grammar: S → L = R | R

L → * R | id

R → L

Step 1: Find augmented grammar:

r0 S' → S

r1 S → L = R

r2 S → R

r3 L → * R

r4 L → id

r5 R → L

Step 2: I_0, closure and got to (using . operator) on initial state

S' → . S

S → . L = R

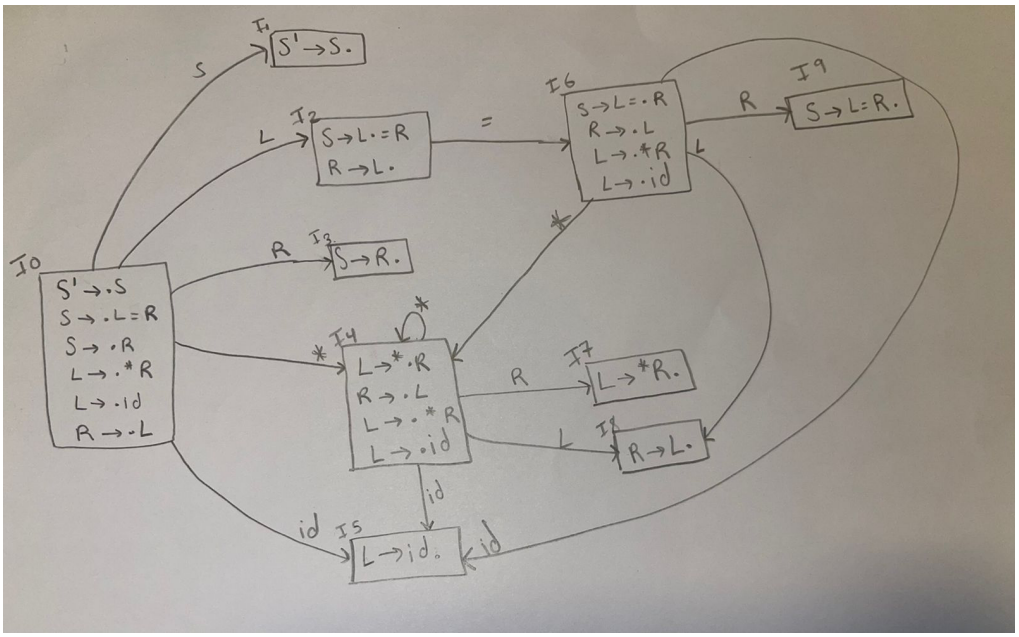
S → . R

L → . * R

L → . id

R → . L

Step 3: Calculate Go-to and closure functions for each state.



Step 4: Calculate follow.

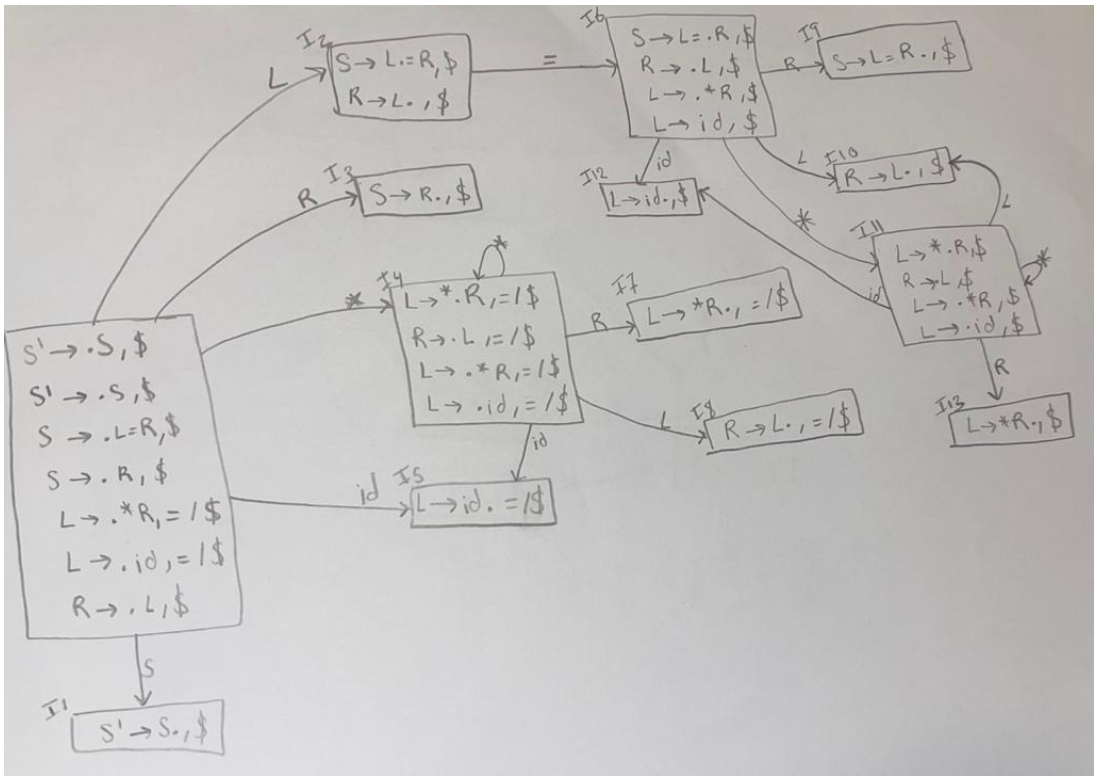
Nonterminal	Follow
S'	{$}
S	{$}
L	{=,$}
R	{$,=}

Step 5: build the parser table based on the graph shown above and follow calculated.

State	Terminals (Actions)				goto			
	=	*	id	$	S'	S	L	R
0		s4	s5			1	2	3
1				accept				
2	s6 / r5 (shift reduce conflict)			r5				
3				r2				
4		s4	s5				8	7
5	r4			r4				
6		s4	s5				8	9
7	r3			r3				
8	r5			r5				
9				r1				

LR(1): Left reduce conflict with a single token lookahead

LR(1) eliminates the shift reduce conflict using lookahead. It produces more states than SLR since it considers the same production sets with different lookaheads a different state.



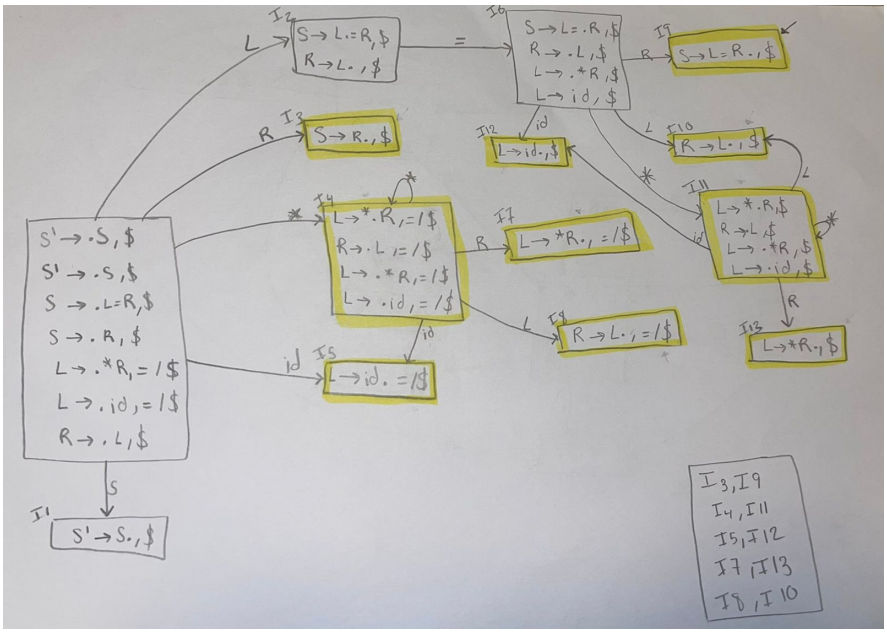
The parse table: The number of states is 13 compared to 9 in SLR.

State	Terminals (Actions)				goto			
	=	*	id	$	S'	S	L	R
0		s4	s5			1	2	3
1				accept				
2	s6			r5				
3				r2				
4		s4	s5				8	7
5	r4			r4				
6		s11	s12				10	9
7	r3			r3				
8	r5			r5				
9				r1				
10				r5				
11		s11	s12				10	13
12				r4				
13				r3				

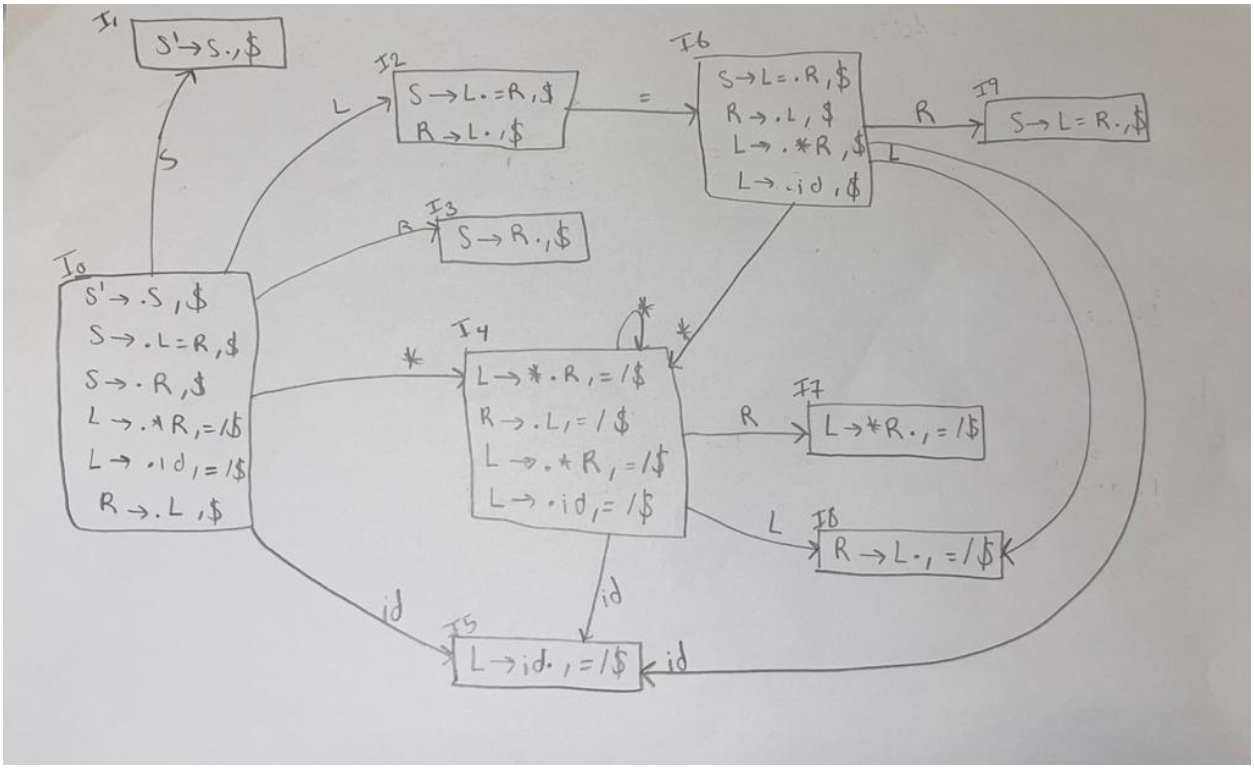
LALR:

Decreases the number of states in LR(1) but maintains the advantages (decreases complexity)

Check for the similar states:



Transform the states

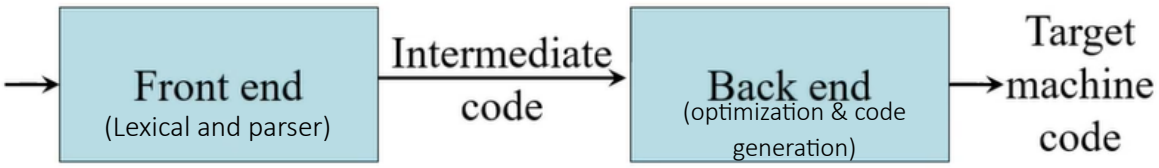


Number of states is decrease and the shift reduce conflict is resolved.

state	Terminals (Actions)				goto			
	=	*	id	$	S'	S	L	R
0		s4	s5			1	2	3
1				acc				
2	s6			r5				
3				r2				
4		s4	s5				8	7
5	r4			r4				
6		s4	s5				8	9
7	r3			r3				
8	r5			r5				
9				r1				

Code generation and optimization

Previously when the compilation process was outlined, I mentioned how after the semantic analysis, an intermediate code is generated (quick to produce, easy to translate, and portable on different machines), then this form is optimized to improve the performance and then it is translated into the target machine language. Below are the code generation and optimization processes techniques described to give a clearer idea.

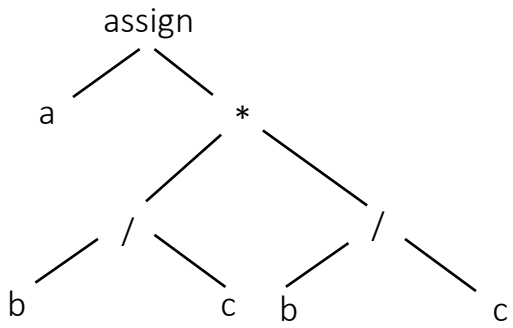


Code generation process

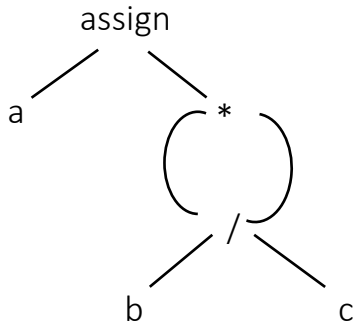
Ways of intermediate code generation:

1. **Graphical representations**: Syntax trees and Directed Acyclic Graphs (DAGs) are critical for understanding code structure and optimizing it. Syntax trees represent the grammatical structure of a code by breaking it down into its component parts. DAGs, on the other hand, provide the same information in a more compact manner and are used to detect and remove common subexpressions which optimizes the overall code. They can describe expressions where each node is an operator and the edges represent operands, therefore minimizing duplicate calculations.

   For example: consider this expression a : = b/c * b/c



Syntax graphical representation



DAG graphical representation

   Note: Dag reduces the number of registers needed to perform the same calculation, which decreases the CPU needed, optimizing the code.

   Programmers can build data structure to represent syntax tree/ DAG to help in code generation through two techniques either by building a linked list where each operator node point to its two children unless it is a leaf or by building a lookup table.

2. Postfix notation: Postfix notation is a linearized representation of a syntax tree; it is a list of the tree's nodes, with each node appearing immediately after its children (operators follow their operands). It is a way to write expressions that eliminates the requirement for parentheses to indicate operation priority. This notation is useful because it streamlines the parsing process and may be used directly to create code for stack-based computers.

   For example, the statement **a : = b/c * b/c** becomes: **a bc/ bc/ * assign** in postfix.

3. Three-address code: is an intermediate code that helps to simplify translation and optimization. It divides a given expression into multiple independent instructions that may be readily translated into assembly language. As the name implies, each TAC instruction normally performs a single operation (ex. Arithmetic, logical, and Boolean) with no more than three operands (names, constants, or compiler generated temporaries). TAC simplifies complex arithmetic and control flow structures, making it appropriate for creating and optimizing target code. Unlike postfix notation, it employs named intermediate values, facilitating code rearrangement and optimization. This improves readability and allows for more efficient compiler transformations, bridging the gap between high-level code and machine instructions effectively.

   Forms of three-address instructions:
   1. Assignment Statement: operator is a binary arithmetic or logical operation. Assignments can work with unary operators' example, unary minus, logical negation, and conversion operators. It assigns the result obtained by solving the right-side expression of the assignment operator to the left side operator.
   2. Copy Statement: It copies and assigns the value of the right operand to the left operand.
   3. Conditional Jump (If x relop y goto X): If the condition "x relop y" is met, the control is transferred to the labeled location (X), all statements in between are skipped. If condition x relop y fails, the next statement in the usual order is executed.
   4. Unconditional Jump (goto X): On executing the statement, the control is sent directly to the location specified by label X.
   5. Procedure Call (param x call p return y): Here, p is a function which takes x as a parameter and returns y.

   For example, **a : = b/c * b/c** might be translated into a sequence:
   t1 : = b / c
   t2:=b / c
   a : = t1* t2

   which can be further optimized into:
   t1 : = b / c
   a : = t1* t1

   The three-address code that represents the DAG will be more concise than the syntax tree leading to more concise assemble code and therefore a more concise machine language. The greater the number of the three address code statements the more registers are required to calculate the

expression leading to a longer time of execution. (We need to minimize the number of variables to decrease the number of registers needed since the operations (Store, load) done on the registers are expensive.)

Data structures for three-address code:
Again let's look at the three-address code of our example:
t1 : = b / c
a : = t1* t1

1. Quadruples have four components: op (which contains the operator's internal code), arg1, arg2, and result. arg1 and arg2 often include pointers to symbol-table entries representing the operands. The result field represents the destination (also a pointer to a symbol table entry.) Quadruples efficiently represent operations by structuring the data to reflect the operation's parts and is very useful for optimizing and editing intermediate code since it clearly distinguishes operands, operators, and result locations.

op	arg1	arg2	result
/	b	c	t1
*	t1	t1	a

2. Triples, unlike quadruples, only have three fields: op, arg1, and arg2. They eliminate the requirement for temporary variables in the symbol table by referring to operations' results directly based on their position in the triple structure. This structure consists of the operator and pointers to the operands, which could be symbol table entries or references to other triples (temporary values). This style simplifies the encoding of intermediate code by directly employing the outcomes of earlier actions, eliminating the need for additional temporary variables (some form of optimization).

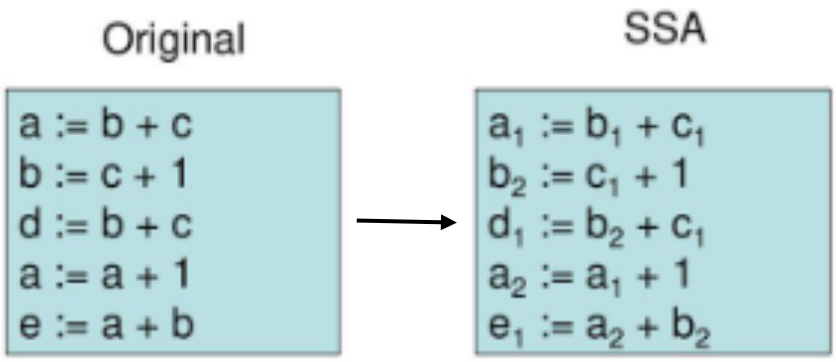
	op	arg1	arg2
0	/	b	c
1	*	(0)	(0)

3. Indirect triples: Instead of listing the triples, this structure uses a list of pointers to triples. This method allows for more flexibility and efficient change of the intermediate code because reordering code or adjusting operations can be accomplished by changing pointers rather than the triples themselves. This is especially useful in optimizations and transformations.

	Op
35	(0)
36	(1)

	op	arg1	arg2
0	/	b	c
1	*	(0)	(0)

4. **Static single assignment (SSA):** a feature of an intermediate representation (IR) that requires each variable to be allocated exactly once and defined before use. This form simplifies and enhances the efficiency of many compiler optimizations, including dead code reduction and constant propagation. The main feature of SSA is the use of versioning for variables, which makes it easier to follow the flow and modification of data through the program.
   Example:

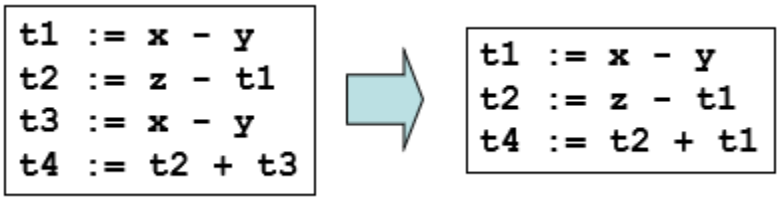


Code optimization process

A control flow graph (CFG) is a visual tool that depicts the flow of a program's execution. It is made up of basic blocks, which are code units that contain a sequence of instructions. These basic blocks are unique in that the flow of control enters at the start and exits at the end, without branching or interruptions in between, except at the final instruction. This representation is critical for code optimization because it gives a clear and organized perspective of the program's execution paths. CFG analysis allows compilers to identify and optimize code portions for efficiency.

The compiler employs several techniques that help with code optimization:

1. **Common-subexpression elimination (CSE)**: It identifies instances where the same expression is evaluated multiple times and replaces these with a single computation. This not only reduces the number of operations but also minimizes memory access.
   For example, if x- y is calculated in multiple places and the values of a and b remain unchanged, the compiler can calculate this expression once and use the result wherever needed.



2. **Dead code elimination**: This technique removes code that has no effect on the program's output, such as instructions after an unconditional jump or variables that are never accessed after being written. This decreases the amount of the generated code while increasing cache utilization.

3. Renaming temporary variables: this technique reuses storage places for variables that are not active at the same moment. This minimizes the number of registers or memory locations required, resulting in better memory use and access times.

4. Interchange of statements: This approach rearranges independent statements to enhance the flow of execution. For example, it can restructure loops or conditional statements to reduce the number of transitions or combine related operations together.

5. Efficient register allocation and assignment: Graph coloring and live variable analysis are important techniques to achieve that. Live variable analysis recognizes variables that are 'alive' at the same point in the program, suggesting that their values are required for future actions. This analysis generates a graph in which each node represents a variable and edges connect nodes with variables that are active at the same moment. Graph coloring is then used to assign colors (representing registers) to each node, guaranteeing that no nearby nodes (variables alive at the same time) have the same color (register). This effectively reduces register utilization, ensuring that the restricted number of registers are used efficiently, decreasing the requirement for slower memory access.

6. Peephole optimization: it improves short sequences of machine code by identifying patterns that can be replaced with more efficient ones. This includes reducing instructions, removing redundant loads or stores, and adopting shorter machine instructions whenever practical.

7. Algebraic Transformations: changing expressions into algebraically equivalent less complicated version. Example: a-a is equivalent to 0. a^2 is equivalent to a*a which is less expensive and equivalent to a<<1 left shift which is the least expensive. Similarly, a+0 is the same as a. This reduces heavy operations and still produces the same results.

8. Local code optimizations: This involves optimizing smaller parts or blocks of code without considering the overall flow of the program and includes a range of techniques here are some. Constant folding involves calculating constant expressions at compile time rather than at runtime, which shortens execution time. Constant combining combines identical constants across the code into a single definition, which improves memory use and processing. Strength reduction speeds up computing by replacing more complex operations, such as multiplication, with simpler ones, such as addition. Constant propagation sends the values of known constants to their usage sites in the code, reducing redundant calculations. Common subexpression elimination detects and reuses already computed expressions rather than recalculating them, hence conserving computational resources. Finally, backward copy propagation improves code by replacing variables with their values if they have not changed, so speeding the execution cycle.

9. Global code optimization: These techniques are advanced and used to optimize across entire programs. Examples include dead code elimination and common subexpression elimination which were mentioned before. Constant propagation which identifies and replaces variables with constant values, resulting in simplified expressions. Forward copy propagation replaces variable occurrences with known values to optimize data flow. Code motion which transfers code out of

loops or to more appropriate spots, lowering the number of executions. Loop strength reduction simplifies costly operations within loops, such as substituting multiplication with addition, to improve loop efficiency. Finally, induction variable elimination simplifies or eliminates variables that fluctuate reliably within loops, resulting in faster loop execution.

Using fewer registers typically results in more efficient use of the processor's resources, which improves overall speed. Registers are the fastest accessible memory regions in a CPU and are limited in quantity, so their proper use is critical. When a program uses fewer registers, it reduces the requirement for register shifting or access to slower memory regions such as cache or RAM, which can drastically slow down performance. Furthermore, employing fewer registers can improve code scalability and adaptability to processors with fewer registers, boosting compatibility.

Referances

hpc-wiki.info. (n.d.). *Compiler - HPC Wiki*. [online] Available at: https://hpc-wiki.info/hpc/Compiler.

Chandra, A. (2022). *Scaler Topics*. [online] www.scaler.com. Available at: https://www.scaler.com/topics/c/compilation-process-in-c/.

Guru99.com. (2019). *Phases of Compiler with Example*. [online] Available at: https://www.guru99.com/compiler-design-phases-of-compiler.html.

BYJUS. (n.d.). *Code Optimization in Compiler Design - GATE CSE Notes*. [online] Available at: https://byjus.com/gate/code-optimization-in-compiler-design-notes/.

Toppr-guides. (2021). *What is Compiler? Definition, Structure, Types, Applications*. [online] Available at: https://www.toppr.com/guides/computer-science/computer-fundamentals/system-software/compiler/.

www.codeconvert.ai. (n.d.). *CodeConvert AI - Convert code with a click of a button*. [online] Available at: https://www.codeconvert.ai/c-to-assembly-converter

GeeksforGeeks. (2017). *Introduction of Compiler Design - GeeksforGeeks*. [online] Available at: https://www.geeksforgeeks.org/introduction-of-compiler-design/.

GeeksforGeeks (2015). *Introduction of Lexical Analysis*. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/introduction-of-lexical-analysis/.

www.youtube.com. (n.d.). *Compiler Design Lec - 06 -Why Lexical Analysis phase is separated from Parser by Deeba Kannan*. [online] Available at: https://youtube.com/watch?v=TTbEpV8oVts

GeeksforGeeks. (2019). *Types of Parsers in Compiler Design*. [online] Available at: https://www.geeksforgeeks.org/types-of-parsers-in-compiler-design/.

Tutorchase.com. (2023). Available at: https://www.tutorchase.com/answers/a-level/computer-science/what-is-the-role-of-a-parser-in-a-compiler.

Team, B.-C.S.T. & S.M., Programming, Tech, Business & IT (n.d.). *Three Address Code in Compiler Design*. [online] BtechVibes- Computer Science Tutorials & Study Material, Programming, Tech, Business & IT. Available at: https://www.btechvibes.com/2023/07/three-address-code.html#:~:text=Implementation%20of%20Three%20Address%20Code&text=The%20internal%20code%20for%20the [Accessed 30 Jan. 2024].

Hero Vired. (n.d.). *Increment and Decrement Operators in C*. [online] Available at: https://herovired.com/learning-hub/blogs/increment-and-decrement-operators-in-c/

Pietro (n.d.). *What Is A Programming Language Grammar?* [online] Compilers. Available at: https://pgrandinetti.github.io/compilers/page/what-is-a-programming-language-grammar/#:~:text=A%20Programming%20Language%20Grammar%20is.

GeeksforGeeks. (2019). *Types of Parsers in Compiler Design*. [online] Available at: https://www.geeksforgeeks.org/types-of-parsers-in-compiler-design/.