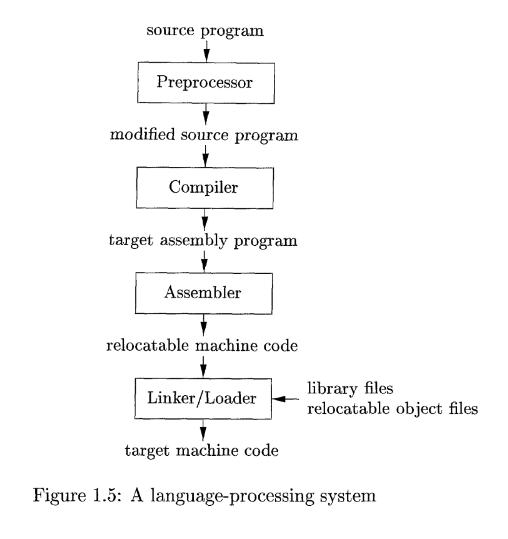
**Introduction to Compiler**

**A language-processing system**



**PREPROCESSOR**: Takes the original *source code* and prepares it by handling special instructions (like including other files, contents of libraries mentioned in the code or setting constants). **This step produces *modified source code*.**

**COMPILER**: Translates the *modified source code* into *assembly code*, a simpler, more hardware-friendly version of the code.

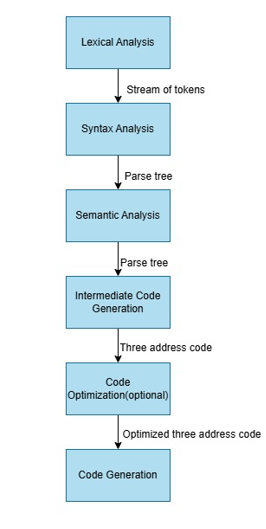
**ASSEMBLER**: Changes the *assembly code* into *relocatable machine code*, which is made up of machine-readable instructions**(binaries)** but still needs to be combined with other code to run.

**LINKER/LOADER**: Combines the *relocatable machine code* from different files, filling in any missing parts, to create *machine code*, which is the final program that can be executed by the computer.

**The Structure of a Compiler**

1. **Compiler Structure**: A compiler translates a source program into an equivalent target program, which involves two main parts: **analysis** and **synthesis**.
2. **Analysis (Front End)**: This part breaks down the source code, organizes it into a structure, and checks for errors:
   * It parses the code into components and imposes a grammatical structure.
   * It creates an *intermediate representation* of the program, used to convert it into the target language.
   * If there are syntax or logic issues, it provides error messages for correction.
   * It collects and stores program details in a **symbol table** to help during later stages.
3. **Synthesis (Back End)**: This part generates the final target code:
   * It uses the intermediate representation and symbol table information to construct the final target program.
4. **Phases of Compilation**: The compiler works through multiple phases, each transforming the code from one form to another. Sometimes phases are grouped, and intermediate forms may not always be explicitly created.

In short, the compiler's front end (analysis) handles the initial breakdown and error-checking, while the back end (synthesis) generates the final output program.

Diagram represents phases of a compiler:

**Lexical Analysis**

The first phase of a compiler is called lexical analysis or scanning. The **lexical analyzer** reads the characters of the source program and groups them into meaningful sequences called **lexemes**

For each lexeme, it generates a token in the form of **<token-name, attribute-value>**, which it then passes to the next phase, syntax analysis.

**Token-name** is an abstract symbol (e.g., **id** for identifier) used during syntax analysis.

**Attribute-value** points to an entry in the **symbol table** for this token.

**For example,** suppose a source program contains the assignment statement

**position = initial + rate \* 60**

**position** becomes **<id, 1>** with **id** indicating an identifier and **1** pointing to its symbol table entry. **id** is an abstract symbol standing for identifier and **1** points to the symbol-table entry for position. The symbol-table entry for an identifier holds information about the identifier, such as its name and type.

The **=** symbol is an **assignment operator**.

It is converted directly into a **token** represented as **<=>**. This token doesn’t need an attribute value, so it appears as just **<=>**. In other cases, a descriptive name like **assign** could be used for the token name, but here the actual lexeme **=** is used for simplicity.

**initial** is another **lexeme** identified as an identifier, mapped to the **token** **<id, 2>**. Again, **id** represents "identifier," and **2** points to **initial**’s entry in the symbol table.

The **+** symbol represents the **addition operator**.

It is mapped directly to the **token** **<+>**, which also doesn’t need an attribute value.

**rate** is recognized as another **identifier**.

It is mapped to the **token** **<id, 3>**, where **3** points to **rate**’s symbol table entry

The **\*** symbol represents the **multiplication operator**. It is mapped to the **token** **<\*>**, with no attribute value.

**60** is recognized as a **numeric constant**.

It is mapped directly to the **token** **<60>**, representing the constant itself.

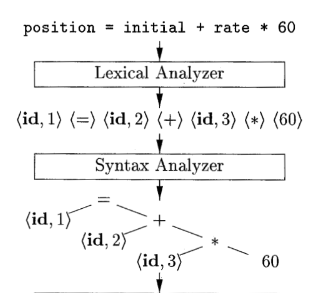
Blanks(white space) separating the lexemes would be discarded by the lexical analyzer.

After **lexical analysis**, the assignment statement **position = initial + rate \* 60** is represented as a sequence of tokens:

**<id,1> <=> <id,2> <+> <id,3> <\*> <60>**

**Syntax Analysis**

The second phase of the compiler is syntax analysis or parsing. The compiler takes the **token stream** produced by the **lexical analyzer** and organizes it into a **syntax tree**. The parser uses the first components **“token-name”** of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that shows the grammatical structure of the code and the order of operations, which is essential for generating the final program. In a typical representation a syntax tree, each interior node represents an **operation** and the children of the node represent the

**arguments** of the operation.

In the example of **position = initial + rate \* 60:**

The tree has an interior node labeled **\*** with **<id, 3>** as its left child and the integer **60** as its right child. The node **<id, 3>** represents the identifier **rate**. The node labeled **\*** makes it explicit that we must first **multiply** the value of **rate** by **60**. The node labeled **+** indicates that we must **add** the result of this multiplication to the value of **initial**. The root of the tree, labeled **=**, indicates that we must store the result of this addition into the location for the identifier **position**. This ordering of operations is consistent with the usual conventions of arithmetic which tell us that **multiplication** has higher **precedence** than **addition**, and hence that the multiplication is to be performed before the addition**.**

**Semantic Analysis**

In the Semantic Analysis, semantic analyzer uses the syntax tree and the information in the symbol table to check the source program ensuring that the program's operations and expressions align with the rules of the programming language, making logical sense and behaving as expected.

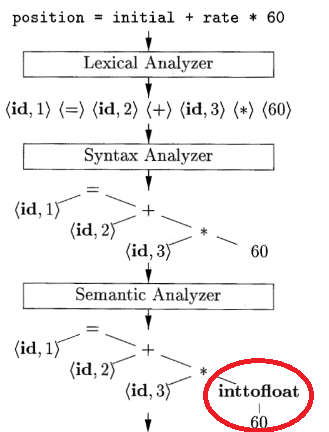
**Consistency Checks**: checks if the program's operations, variables, and data types are logically valid.

**Type Information**: collects **type information** (e.g., integer, floating-point) for each variable or expression. This information is saved in either the syntax tree or the symbol table to help in later stages of the compilation process.

**Type Checking**: A crucial part of semantic analysis is **type checking**. The compiler verifies that each operation in the program has operands of the expected type. For example, if an array index is used, the compiler checks that the index is an integer (since many languages don’t allow non-integer array indices). If an inappropriate type is used, the compiler reports an error.

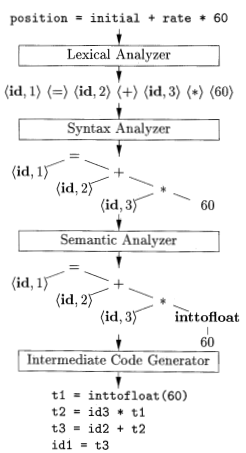
**Type Conversion (Coercion)**: Some languages allow certain **type conversions** (or **coercions**), where the compiler automatically changes one type to another to match the expected type. For example, if a multiplication operation involves a floating-point number and an integer, the compiler might convert (or coerce) the integer to a floating-point number so that both operands match.

In the previous example **(position = initial + rate \* 60),** suppose that **position**, **initial**, and **rate** have been declared to be **floating-point numbers**, and that the lexeme **60** by itself forms an integer. The type checker in the semantic analyzer discovers that the operator **\*** is applied to a **floating-point numbe**r **rate** and an integer **60**. In this case, the integer may be converted into a floating-point number. In the picture below we can notice that the output of the semantic analyzer has an extra node for the operator **inttofloat**, which explicitly converts its integer argument into a floating-point number.



**Intermediate Code Generation**

In the **Intermediate Code Generation** phase, the compiler converts the syntax tree into a form of low-level code known as an **intermediate representation.** This **intermediate representation** looks somewhat like assembly code because it uses simple, step-by-step instructions. This **intermediate representation** is useful because it’s both easy to generate and easy to translate into actual machine code later.



One common type of IR is called **three-address code**. There are several points worth noting about three-address instructions. Firstly, each instruction in three-address code has a maximum of **one operator** on the **right side**, with up to **three operands** in total. Thus, these instructions fix the order in which operations are to be done; the multiplication precedes the addition in the source program. Secondly, the compiler must generate a temporary name to hold the value computed by a three-address instruction. Third, some "three-address instructions" like the first and last line in the sequence below have fewer than three operands.

**t1 = inttofloat (60)**---->Line1

**t2 = id3 \* t1**------------->Line2

**t3 = id2 + t2**------------>Line3

**id1 = t3**------------------>Line4

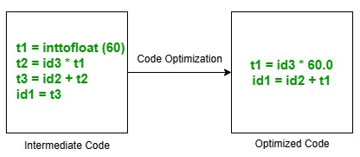
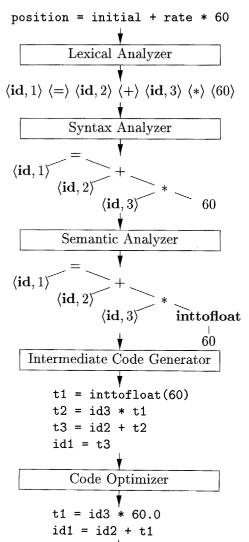
**Code Optimization**

**Code Optimization** is a phase in the compiler that tries to improve the code to make it faster, shorter, or more efficient in terms of memory or power usage. The machine-independent code-optimization phase attempts to improve the intermediate code. A straightforward algorithm generates the intermediate code shown above, using an instruction for each operator in the tree representation that comes from the semantic analyzer.

A simple intermediate code generation algorithm followed by code optimization is a reasonable way to generate good target code. In the case of **60** being converted to a floating-point number, the optimizer realizes it only needs to do this conversion once. Instead of doing it repeatedly, it changes **60** to **60.0** in the code shown below, making it faster and more efficient. Also, **t3** is used only once to transmit its value to **id1** so the optimizer can transform the above code into the shorter sequence as shown below

**t1 = id3 \* 60.0**-------->Line1

**id1 = id2 + t1**--------->Line2



**Code Generation**

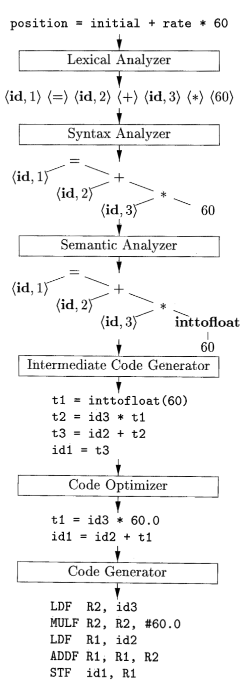
In the **code generation** phase, the compiler translates the intermediate representation of the source program into target machine code. The main tasks in this phase include selecting registers or memory locations for each variable and translating each intermediate instruction into a sequence of machine instructions that perform the same task.

**LDF R2, id3**

**MULF R2, R2, #60.0**

**LDF Rl, id2**

**ADDF R1, R1, R2**

**STF id1, R1**

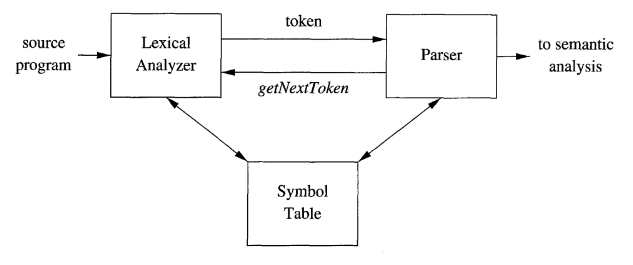
**Introduction to Lexical Analysis**

**The Role of the Lexical Analyzer**

The **lexical analyzer** is the first phase in a compiler, responsible for reading the characters in a source program, grouping them into lexemes (basic code components), and then outputting tokens for each lexeme. These tokens are then passed to the parser for syntax analysis.

The lexical analyzer also interacts with the symbol table. When it finds an identifier, it records it in the symbol table, and in some cases, reads information from the table to help determine the correct token to pass to the parser.

Typically, the parser calls the lexical analyzer using a command like **getNextToken**, which makes the analyzer read characters until it finds the next lexeme, identifies it, and returns the corresponding token to the parser.



Since it reads the source text, the lexical analyzer may handle additional tasks:

* Removing comments and unnecessary whitespace (like blanks and newlines).
* Keeping track of line numbers for accurate error reporting, potentially inserting errors directly into the source code.
* Expanding macros if the source program has a macro-preprocessor.

The lexical analyzer might be split into two parts:

1. **Scanning**: Handles basic processes like removing comments and compacting whitespace.
2. **Lexical analysis:** lexical analysis itself is the more complex portion, where the scanner produces the sequence of tokens as output.

**Tokens, Patterns, Token Attributes**

When discussing lexical analysis, we use three related but distinct terms:

**Token:** A token is a pair made up of a token name and an optional attribute value. The **token name** represents the type or kind of lexical unit. For example, a keyword like **if** might be represented by a token named **KEYWORD**, or a sequence of characters representing an identifier might have the token name **IDENTIFIER**. The token names are the input symbols that the parser processes. we shall generally write the name of a token in boldface. We will often refer to a token by its token name.

**Patterns:** A pattern is a description of how the lexemes (the actual text) of a token should look. For example, the pattern for the **KEYWORD** token might simply be the string "if," as the keyword is just the exact word "if". The pattern for a keyword token (like **if, while**, or **return**) is simple because the token name directly matches a specific word. The token **if** only matches the lexeme **if**, so its pattern is straightforward: just the string **"if".** The pattern for identifiers (which are variable or function names like **x, sum**, or **main**) is more complex because there are many valid combinations that can match the pattern. An identifier could be a string of letters, digits, and possibly underscores, but it must start with a letter or underscore (not a digit). So, for identifiers, the pattern is not just one specific string, but rather a **set of rules** that allow many different strings (like **main, x1, \_sum,** etc.) to match the identifier token.

**Lexemes:** A lexeme is the actual sequence of characters found in the source program that matches a specific token pattern and is identified by the lexical analyzer as an instance of that token. For example, in the statement **int main()**, the lexeme **main** is recognized as an instance of the token **IDENTIFIER**.

In Short:

**Token**: The type or category of a lexical unit.

**Pattern**: The rule or description of the form that a token's lexeme must follow.

**Lexeme**: The actual string of characters in the source code that matches a token's pattern.

**Example:**

**int main()**

**Tokens:  
  
Token 1: (KEYWORD, "int")**

**Token 2: (IDENTIFIER, "main")**

**Token 3: (PUNCTUATION, "(")**

**Token 4: (PUNCTUATION, ")")**

**Pattern:**

**int** → exact match string

**main** → sequence of letters

**( and )** → single-character match

**Lexemes:**

**int**

**main**

**(**

**)**

**Token Attributes**

A token attribute is additional information attached to a token that provides details about the specific lexeme (a sequence of characters in the source code) matched by the token. While the token name indicates the general category of the lexical unit (e.g., NUMBER, ID, KEYWORD), the attribute gives specific details about the exact instance of that category.

**Regular Definitions**

Regular definitions allow us to define complex patterns in a structured way by breaking them into simpler components. They use **regular expressions** to specify rules for valid strings, and we can assign names to these rules to make them reusable and easier to read.

**What are the steps of a compiler?**

The steps of a compiler are:

1. **Lexical Analysis:** Breaks the source code into tokens.
2. **Syntax Analysis:** Checks for grammatical structure and generates a parse tree.
3. **Semantic Analysis:** Ensures correctness of meaning and type-checking.
4. **Intermediate Code Generation:** Produces an abstract representation of the source program.
5. **Optimization:** Improves the intermediate code for efficiency.
6. **Code Generation:** Converts optimized code into target machine code.
7. **Code Linking and Loading:** Combines modules and prepares the executable for execution.

**What happens in lexical analysis?**

In **lexical analysis**, the source code is scanned to identify **lexemes** (basic units like keywords, identifiers, and symbols). These are converted into **tokens** (e.g., <id, x>, <number, 10>), while comments and whitespace are removed.

**What happens in syntax analysis?**

In **syntax analysis** (parsing), the sequence of tokens from lexical analysis is checked against the grammar of the programming language. A **parse tree** is generated to represent the syntactic structure if the tokens follow the rules; otherwise, **syntax errors** are reported.

**What happens in semantic analysis?**

Semantic analysis ensures the program is logically correct by checking that variables are defined before use, data types are compatible, and functions are called with the correct arguments. It verifies rules like assigning the right data type to variables and keeps track of program elements to catch logical errors.

**What are the types of errors lexical, syntax and semantic analysis remorts?**

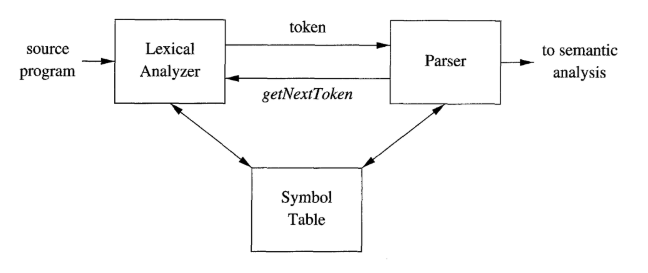
### **Types of Errors:**

* **Lexical Errors**: Invalid tokens or unrecognized characters (e.g., illegal symbols).
* **Syntax Errors**: Incorrect code structure (e.g., missing brackets, misplaced semicolons).
* **Semantic Errors**: Logical inconsistencies (e.g., type mismatches, undeclared variables).

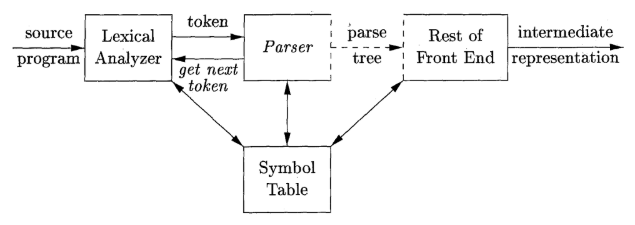
**what is lexeme?**

A **lexeme** is a sequence of characters in the source code that matches the pattern for a token.

**Diagram of Lexical analyzer**

****

**Diagram of parser**

****

**What is token, pattern and lexemes?**

**Tokens**: Pairs of token names and optional attribute values representing language units.

**Patterns**: Descriptions of the form that lexemes of a token can take.

**Token Attributes**: Extra information associated with a token, such as type or value.

**Syntax Analysis**

**The Role of the Parser**

The parser’s main job in a compiler is to take a sequence of tokens from the lexical analyzer (which breaks down code into basic elements) and check if they follow the grammar rules of the programming language. Think of it as verifying that the sentence structure of the code is correct. If there’s a syntax error, the parser should notify the user in a way that’s easy to understand, and ideally, it should be able to handle minor errors so it can keep processing the rest of the code.

For code that is correctly structured, the parser can build a "parse tree," a diagram that shows the relationships between different parts of the code based on grammar rules. However, it doesn’t always need to create this tree explicitly; instead, it can carry out other checks and translations while parsing, which makes the process more efficient.

There are three general types of parsers for grammars: **universal, top-down, and bottom-up**. As implied by their names, top-down methods build parse trees from the top (root) to the bottom (leaves), while bottom-up methods start from the leaves and work their way up to the root. In either case, the input to the parser is scanned from left to right, one symbol at a time.

**Universal parsiers:** Parsing methods such as the Cocke-Younger-Kasami algorithm and Earley's algorithm can parse any grammar. These general methods are, however, too inefficient to use in production compilers.

**Top-Down Parsers**: Build the parse tree from the top (root) down to the (leaves). They work well for LL grammars, a grammar type that handles most modern programming language constructs.

**Bottom-Up Parsers**: Start from the (leaves) and build up to the (root). These are often used with LR grammars, which are powerful enough to handle many complex language constructs and are usually constructed using automated tools.