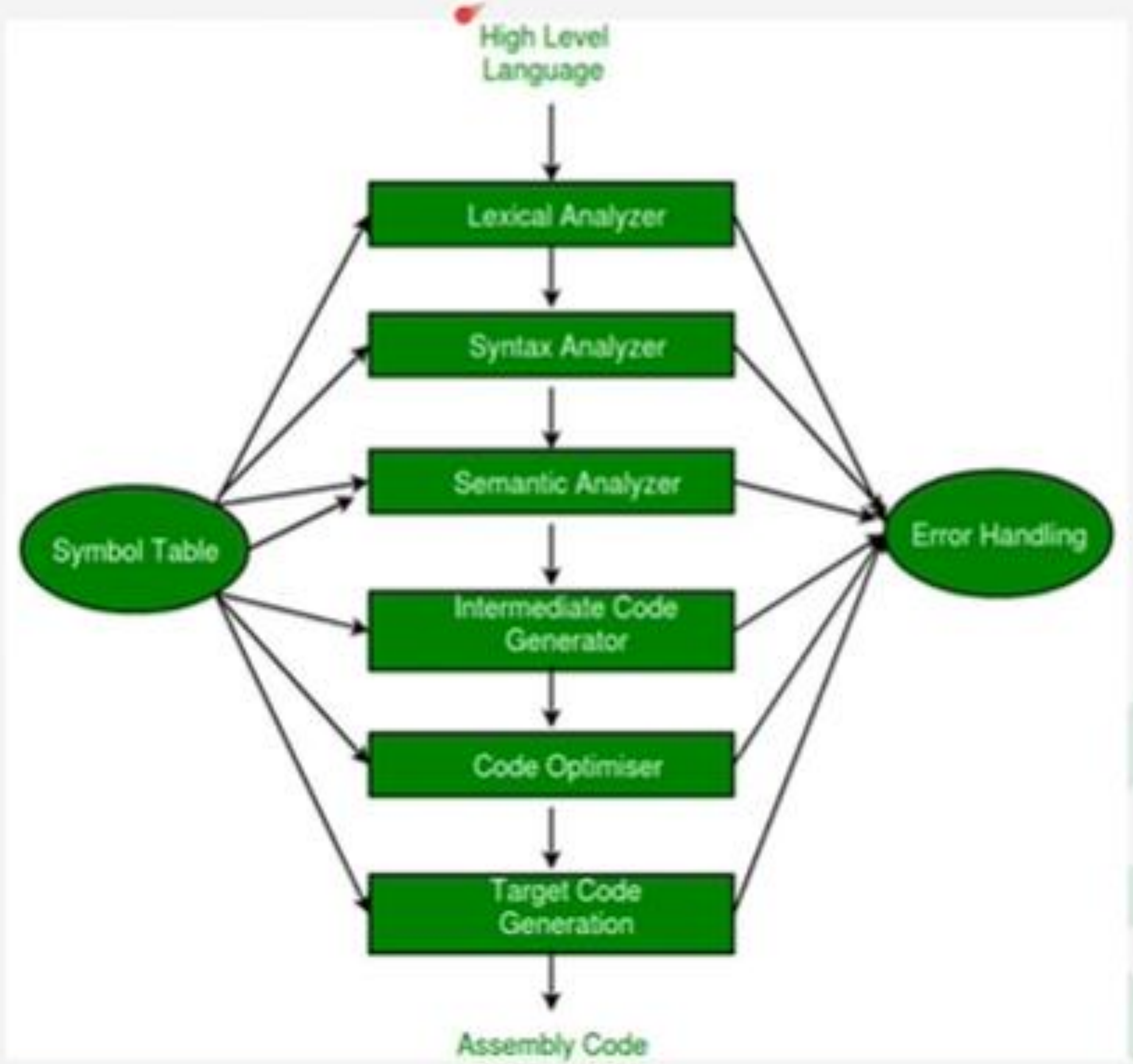
Phases of Compiler

In Compiler Design

What is the need of Phases of Compiler?

- The main task of Compiler is to convert high level programming language into low level language or machine code.
- Compiler operates in various phases where each phase transforms the source code/ program from one representation to another (higher level to lower level).
- There are six phases in a compiler. Each phase takes input from its previous phase and gives output to the next phase. Each of this phase helps in converting the high-level language into machine code.

List of Phases of Compiler



Phases of Compiler

There are two phases of compilers: Analysis phase and Synthesis phase.

Analysis phase: Creates an intermediate code from given source code.

Synthesis phase: Creates the final target code from the intermediate code.

- These two phases are also commonly termed as the **front end (analysis phase)** and the **back end (synthesis phase)**.
- Front-end consists of the Lexical analyzer, semantic analyzer, syntax analyzer, and intermediate code generator. On the other hand, backend part consists of code optimizer and target code generator.

Lexical Analyzer

In the Lexical Analysis phase entire source code gets scanned by the compiler. This phase takes pure high-level language as input, reads its characters and groups them into lexemes/ tokens.

It makes the entry of the corresponding tokens into the symbol table and passes that token to next phase.

The primary functions of lexical analyzer are:

- Identifying lexical units in a source code
- Ignoring comments in the source program
- Removing white space, tab space, new-line character

Example:

If this enters as an input: **x=a+ b*c;**

No. of tokens = 8

id=id+id*id

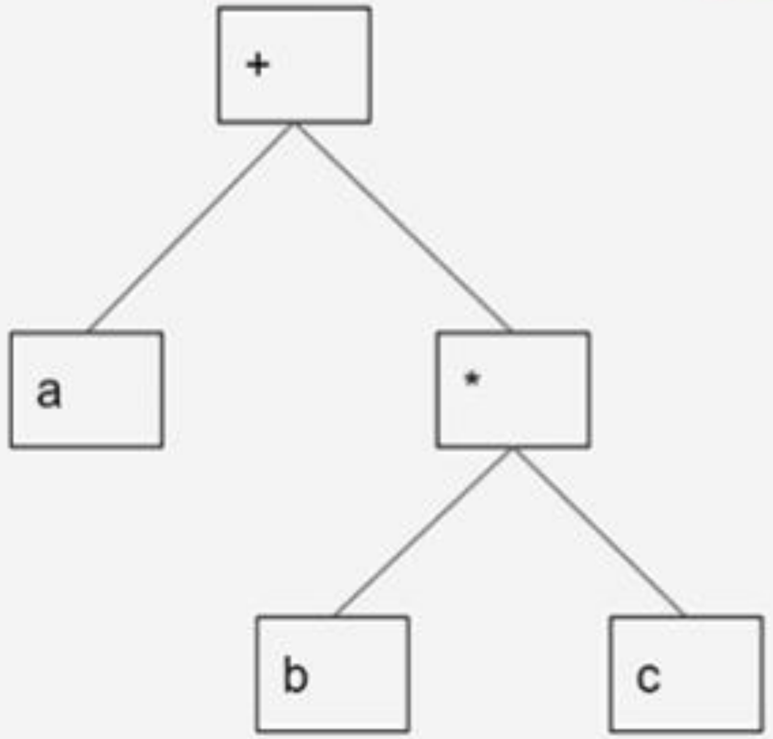
Syntax Analyzer

It takes all the tokens one by one from Lexical Analyzer and uses Context-Free Grammar to construct the parse tree.
The following is a list of tasks performed in syntax analysis phase:

- Take tokens from lexical analyzer as input.
- Checks the expression whether it is syntactically correct or not.
- Reports syntax errors, if any
- Construct hierarchical structure known as a syntax tree or parse tree

Syntax Analyzer

Example of Syntax tree for “a+b*c”

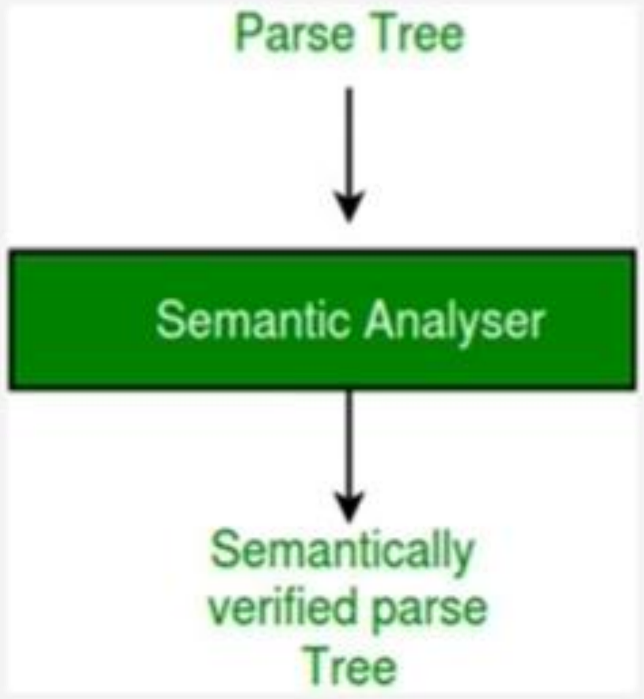


Semantic Analyzer

- Semantic Analyzer verifies if the parse tree is semantically correct or not, meaningful or not. If not, it produces a verified parse tree.

- It will check for mismatching data types, incompatible operands, a function called with improper arguments, undeclared variables, etc.

- In our example, it will give same parse tree in output as it is already semantically correct.



Intermediate Code Generator

- Once the semantic analysis phase is over, the compiler generates intermediate code for the target machine. Intermediate code is something between high-level and machine level code. This intermediate code needs to be generated in such a manner that makes it easy to be translated into the target machine code.

- We have many variations of intermediate codes. Three address code is one of the most commonly used one.
- Intermediate code is converted into machine language by using the last two platform dependent phases.

Intermediate Code Generator

- The intermediate code for our example **x=a+b*c** would be:

  T1 = b*c

  T2 = a+T1

  X = T2

Code Optimizer

- The main aim of code Optimizer is to transform the code so that it consumes fewer resources and produces more speed.

- For the intermediate code:

```
T1 = b*c
T2 = a+T1
X = T2
```

- The optimized code can be:

```
T1 = b*c
x = a+T1
```

Target Code Generator

- Target Code generator generates a machine understandable code.

- This phase is the final stage in compilation. The optimized code is converted into relocatable machine code. For example, let's take Assembly code as our Target code.

- For our previous code:

  ```
  T1 = b*c
  x = a+T1
  ```

- The target code / Assembly code can be:

  ```
  a→R0, b→R1, c→R2
  Mul R1,R2
  Add R0,R2
  Mov R2,x
  ```

Symbol Table

- Symbol Table is a data structure which is created and maintained by the compiler to keep track of variables i.e. it stores information about the scope and binding of various entities such as variable and function names, classes, objects, etc.
- A symbol table makes it easier for the compiler to search the identifier record and retrieve it quickly.
- The information in Symbol Table are being collected by the analysis phases of the compiler and are used by the synthesis phases of the compiler.

Error Handling

Similar to Symbol Table, all the phases of compiler are also connected to Error Handler. The tasks of the Error Handler are to detect each error, report it to the user, and then make some recovery strategy and implement them to handle the error.

Three primary functions of Error Handler are:

- Error Detection
- Error Reporting
- Error Recovery

Error Handling

Error Handlers can detect following categories of Errors:

1. **Lexical** : It detects error when sequence of any input doesn't match any token pattern. So, lexical error includes misspellings of identifiers, operators, keywords, etc.
2. **Syntactical** : It detects error during syntax phase if something is missing. It includes missing parenthesis or unbalanced parenthesis, when some operators like semicolons are missing, etc.

Error Handling

3. **Semantical** : These errors are detected during semantic analysis phase. Such errors are the results of wrong matching of operators with operands, incompatible value assignment to any variable, etc.

4. **Logical** : Any error that is not specific enough to be included in above three categories and that includes logical issues in problem are logical errors. For example: Code not reachable, infinite loop, etc.

Computer Language Representation

- Refers to the way programming languages and data are structured and interpreted by both humans and machines.
- 1. Low-Level vs. High-Level Languages
- Low-Level Languages:
  - Machine Code: Binary instructions (0s and 1s) directly executed by the CPU.
  - Assembly Language: Human-readable mnemonics (e.g., MOV, ADD) that map to machine code. Requires an assembler to translate.
- High-Level Languages (e.g., Python, Java, C++):
  - Abstracted from hardware, focusing on readability and productivity.
  - Require compilers or interpreters to convert code into machine-executable form.
- 2. Language Syntax and Semantics
- Syntax: Rules governing code structure (e.g., grammar, punctuation).
- Semantics: Meaning of the code (e.g., how loops or functions behave).

- **3. Compiled vs. Interpreted Languages**
- **Compiled Languages** (e.g., C, Go):
    - Translated entirely to machine code before execution (via a **compiler**).
    - Faster execution but platform-dependent.
- **Interpreted Languages** (e.g., Python, JavaScript):
    - Executed line-by-line by an **interpreter** at runtime.
    - Platform-independent but slower.
- **4. Intermediate Representations (IR)**
- Used by compilers/interpreters to optimize code before generating machine code.
- Examples:
    - **Bytecode** (e.g., Java JVM, Python .pyc): Portable intermediate code.
    - **LLVM IR**: A universal IR for optimization across languages.

- **5. Data Representation**
  - **Primitive Data Types**: Integers, floats, characters, booleans.
  - **Composite Types**: Arrays, structs, objects.
  - **Memory Addressing**: How data is stored (e.g., big-endian vs. little-endian).
- **6. Control Structures**
  - **Sequential Execution**: Code runs line-by-line.
  - **Conditionals**: if-else, switch statements.
  - **Loops**: for, while, do-while.

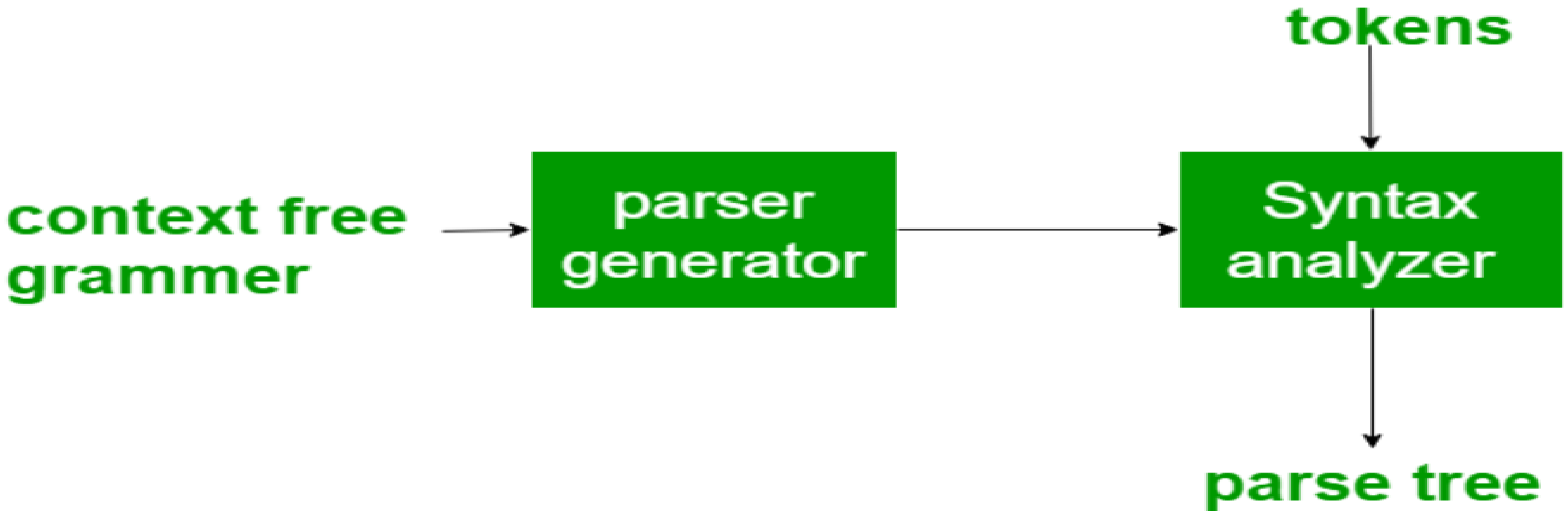
- **7. Abstraction Layers**
- **Hardware Abstraction**: Operating systems manage hardware interactions.
- **APIs/Libraries**: Pre-written code for common tasks (e.g., math functions).
- **Virtual Machines** (e.g., JVM, CLR): Execute intermediate code across platforms.
- **8. Domain-Specific Languages (DSLs)**
- Tailored for specific tasks (e.g., SQL for databases, HTML/CSS for web design).
- **9. Formal Language Theory**
- **Regular Expressions**: Pattern matching in text.
- **Context-Free Grammars**: Describe programming language syntax (e.g., BNF notation).

- **10. Tools for Language Processing**
- **Lexers**: Break code into tokens (e.g., keywords, operators).
- **Parsers**: Analyze token structure against grammar rules.
- **Optimizers**: Improve code efficiency (e.g., removing redundancies).
- **Example Workflow (C Program):**

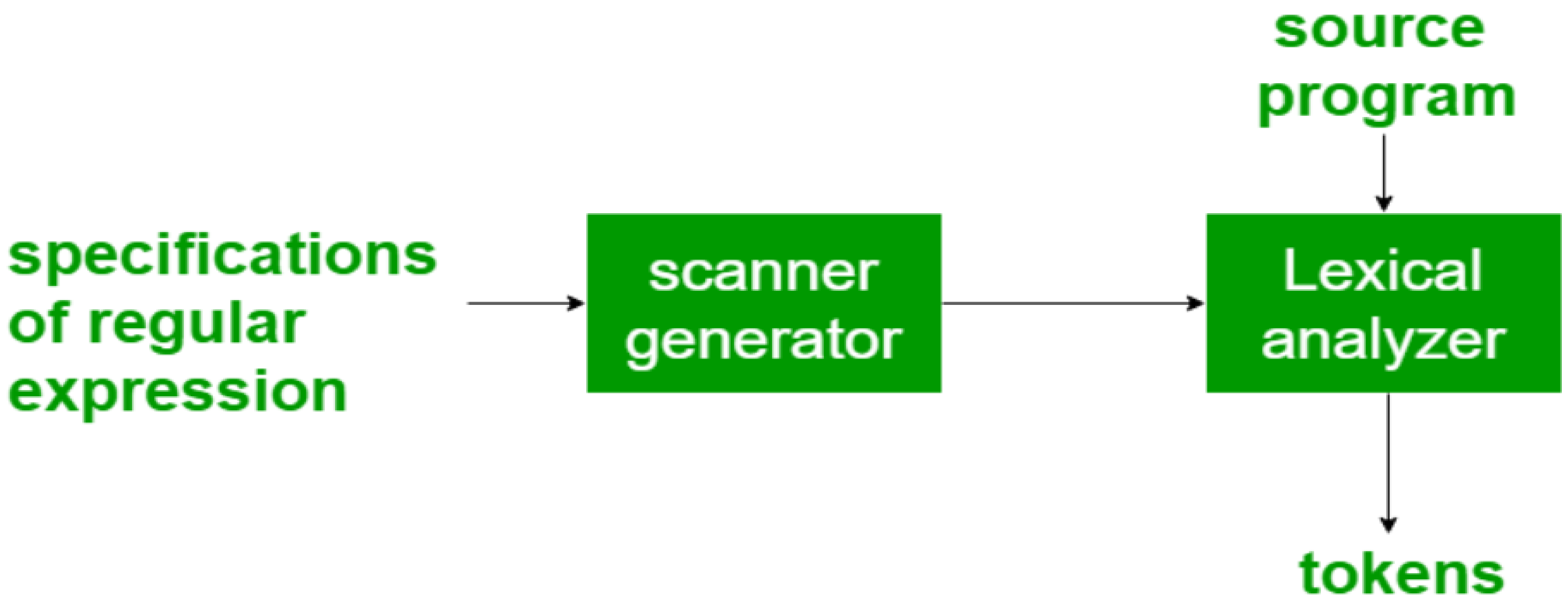
1. **Source Code**: main.c (high-level code).
2. **Compiler**: Translates to assembly (main.s).
3. **Assembler**: Converts assembly to machine code (main.o).
4. **Linker**: Combines object files into an executable (a.out).

Compiler construction tools

1. **Parser Generator** – It produces syntax analyzers (parsers) from the input that is based on a grammatical description of programming language or on a context-free grammar. It is useful as the syntax analysis phase is highly complex and consumes more manual and compilation time.
   Example: PIC, EQM



2. **Scanner Generator** – It generates lexical analyzers from the input that consists of regular expression description based on tokens of a language. It generates a finite automaton to recognize the regular expression. Example: Lex



3. **Syntax directed translation engines** – It generates intermediate code with three address format from the input that consists of a parse tree. These engines have routines to traverse the parse tree and then produces the intermediate code. In this, each node of the parse tree is associated with one or more translations.
4. **Automatic code generators** – It generates the machine language for a target machine. Each operation of the intermediate language is translated using a collection of rules and then is taken as an input by the code generator. A template matching process is used. An intermediate language statement is replaced by its equivalent machine language statement using templates.

5. **Data-flow analysis engines** – It is used in code optimization.Data flow analysis is a key part of the code optimization that gathers the information, that is the values that flow from one part of a program to another. Refer – data flow analysis in Compiler
6. **Compiler construction toolkits** – It provides an integrated set of routines that aids in building compiler components or in the construction of various phases of compiler.

THANK
YOU

Q&A