LEXICAL ANALYSIS = SCANNING = LINEAR ANALYSIS

- The source program in HLL is a stream of characters read from left to right
- Scanner reads characters from that stream and groups them into tokens.
- Tokens are sequences of characters which form the low-level constructs of the HLL (e.g. variable names, keywords, labels, operators)
- Example:

```
Position = initial + step * 60
```

- Whitespace (blanks, tabs, returns) and comments are eliminated
- Scanner outputs a stream of tokens

SYNTAX ANALYSIS = PARSING = HIERARCHICAL ANALYSIS

- Parser reads a stream of tokens,
- recognizes the grammatical phrases that they form,
- and outputs a parse tree showing the grammatical structure of the program
- Example:

```
Position = initial + step * 60
```

SEMANTIC ANALYSIS

- Checks source program for *semantic errors* (errors in meaning) i.e. enforces the rules of the language which are not defined in its syntax.
- Detects errors in syntactically legal but meaningless statements. Example:

```
int f() {int i; …}
i = 1;
```

- Makes sure that variables and functions are properly defined.
- Makes sure that variables and functions are properly used:
  - Performs type checking and type conversion when needed
  - Checks that structured variables and functions are used correctly.
  - Enforces scoping and visibility rules
- Works with Symbol Table Manager
- Pseudo-stage, often integrated with other stages such as evaluation or intermediate code generation →output depends on requirements of next stage.

SYMBOL TABLE MANAGEMENT

- A symbol table is a table of all identifiers used in the program and their attributes.
- For example,

- Symbol Table Manager manages that table. It initiates it during scanning and uses it throughout translation process.

ERROR MANAGEMENT

- Manages errors throughout translation process:
  - Reports errors to user
  - Tries to recover from errors as well as possible to continue translation.

INTERMEDIATE CODE GENERATION

- Creates intermediate representation of source program.
- Should be easy to produce
- Should be easy to translate
- Example:

```
Position = initial + step * 60
```

INTERPRETATION ONLY: EVALUATION

- Takes modified syntax tree and evaluates (i.e. calculates values) it recursively working with symbol table.
- Example:

```
Position = initial + step * 60
```

INTERMEDIATE CODE OPTIMISATION (OPTIONAL)

- Attempts to improve intermediate code to produce faster running code.
- Ad-hoc approaches: some trivial changes, some complicated.
- Examples:
  - loop unrolling
  - optimisation of loop invariants
  - removal of loop invariants

CODE GENERATION

- Generation of target or assembly code

CODE OPTIMISATION (OPTIONAL)

- Optimisation of code for specific architecture, e.g.
  - Special instructions
  - Usage of registers

ASSEMBLY (OPTIONAL)

- Translation of assembly code into machine code

LINKING

- Searches external libraries for definitions of identifiers which are not defined locally.
  - Resolves the references to undefined variables by getting an external address.
  - Resolves the calls to undefined functions by adding the function definitions to the code. → adds more code to program

LOADING

- Copies a compiled program (in machine code) into RAM
- Gives it all the memory it needs to run (for variables, stack, and heap)
- Starts executing the first instruction of that program.