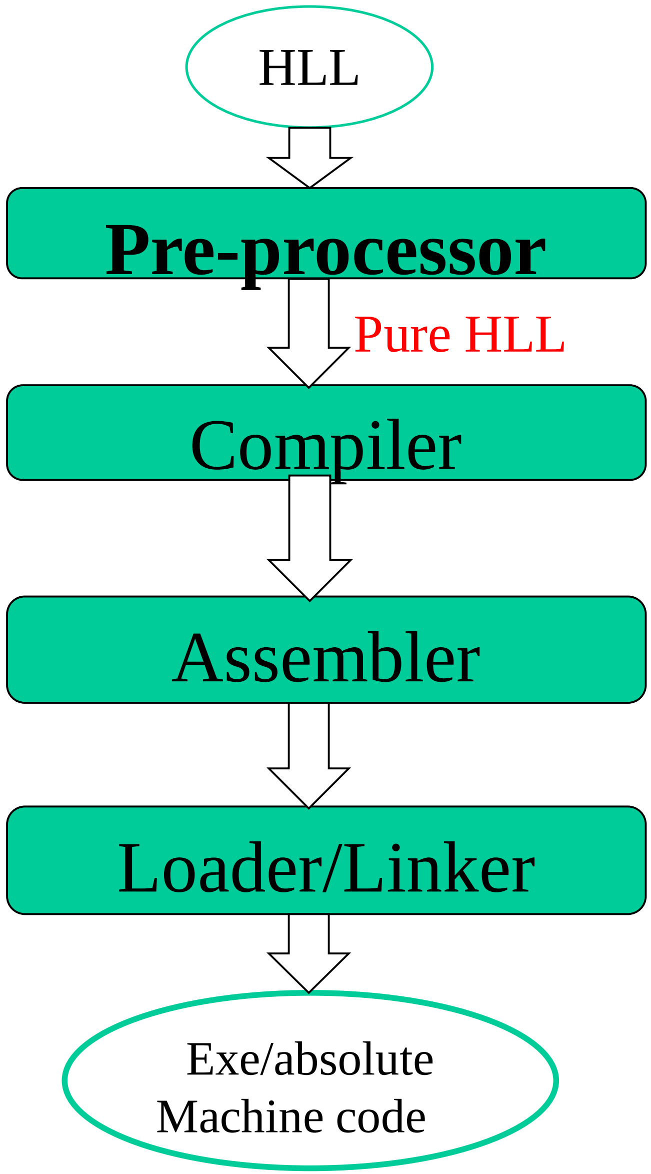
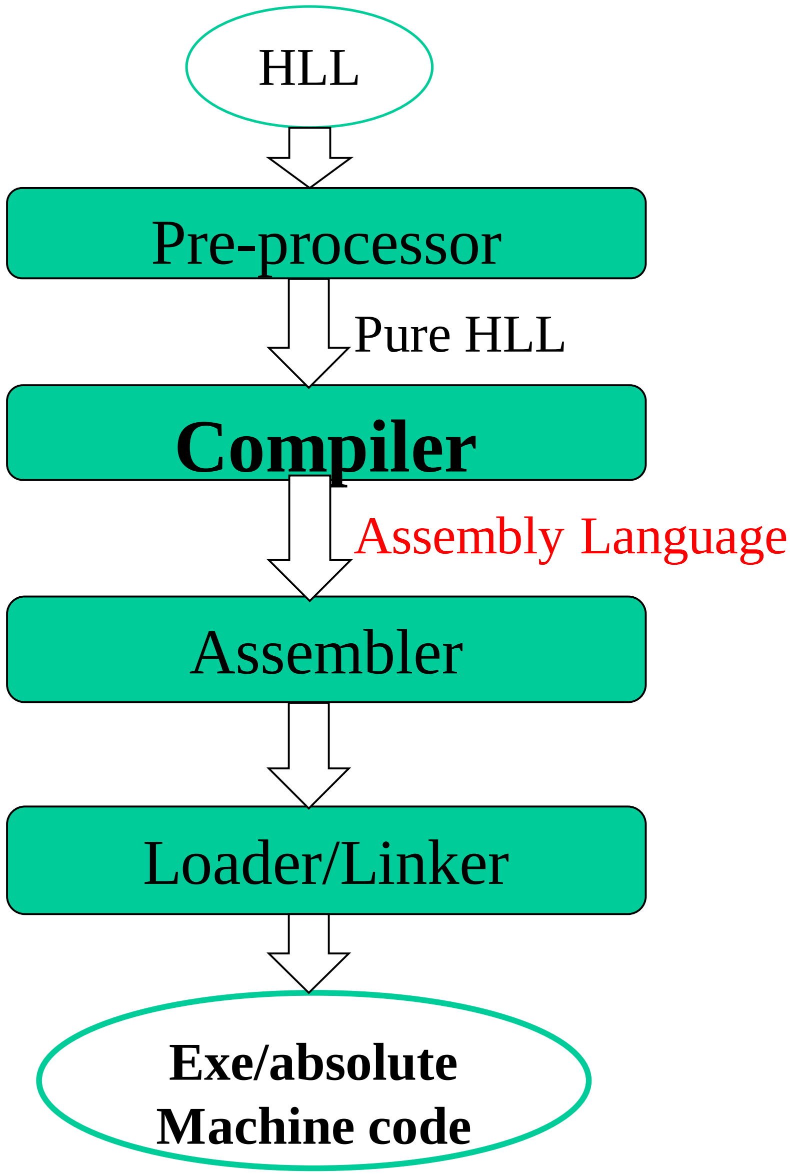
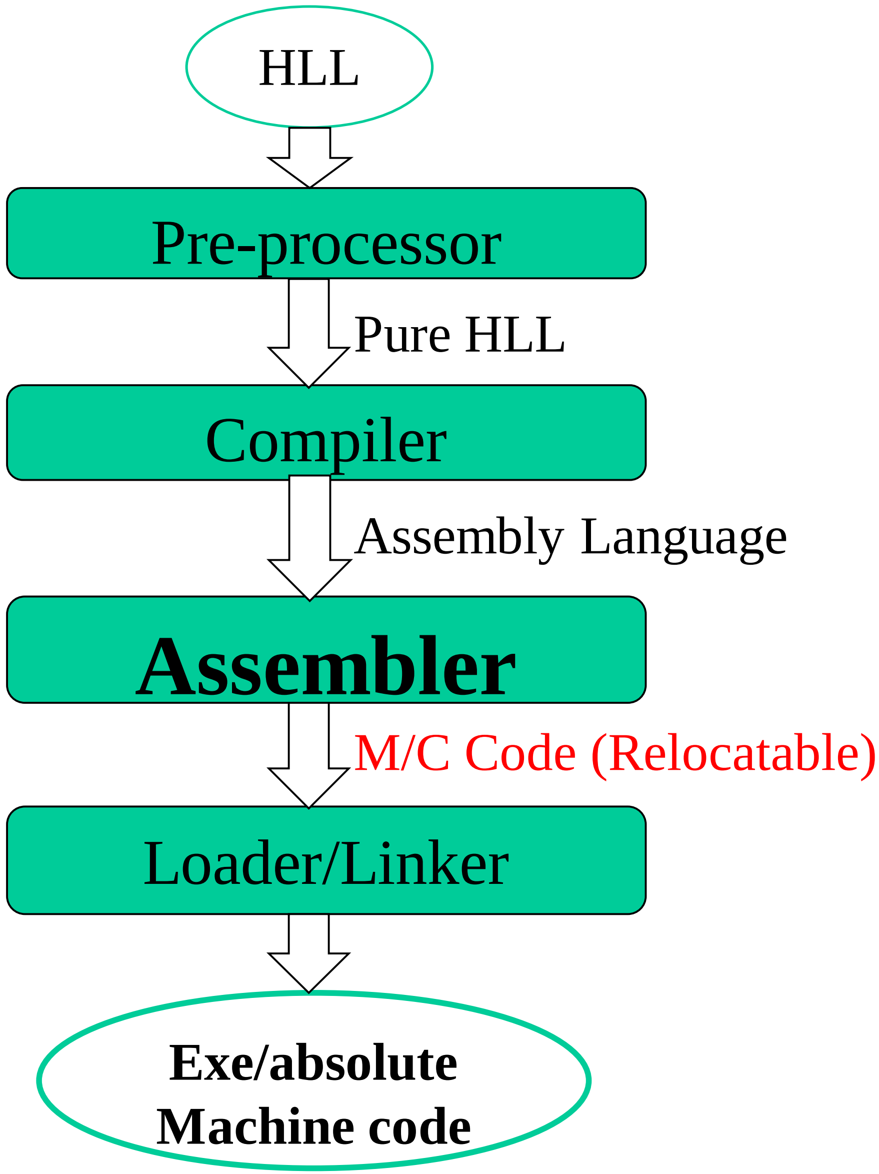
Execution of a Program



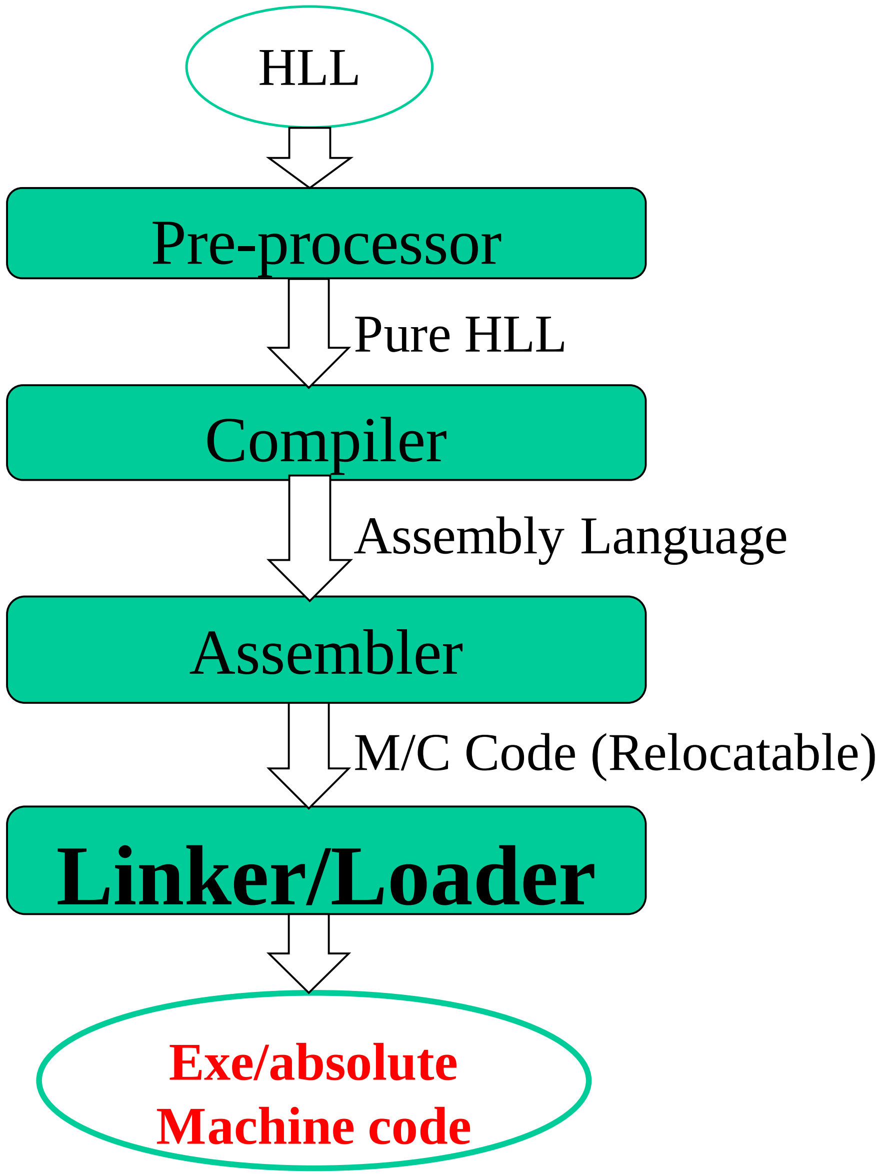
Execution of a Program



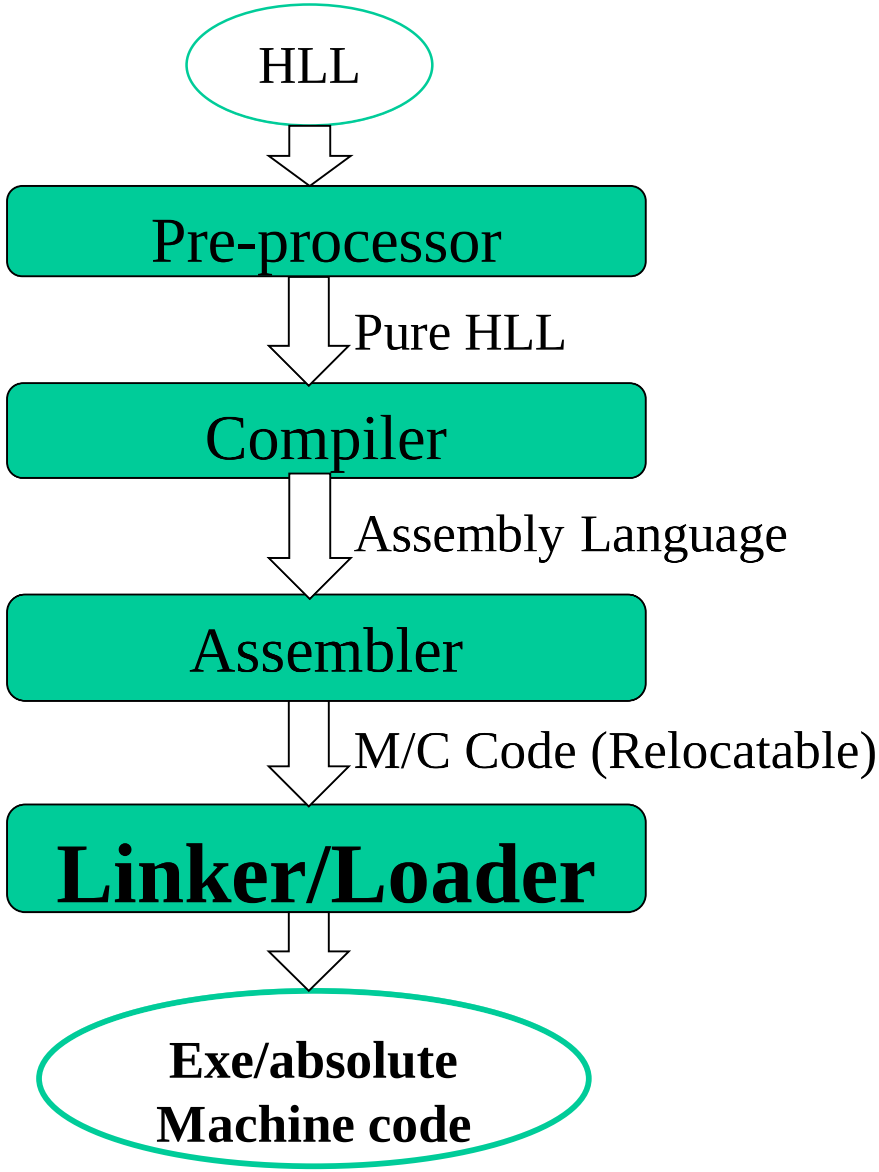
Execution of a Program



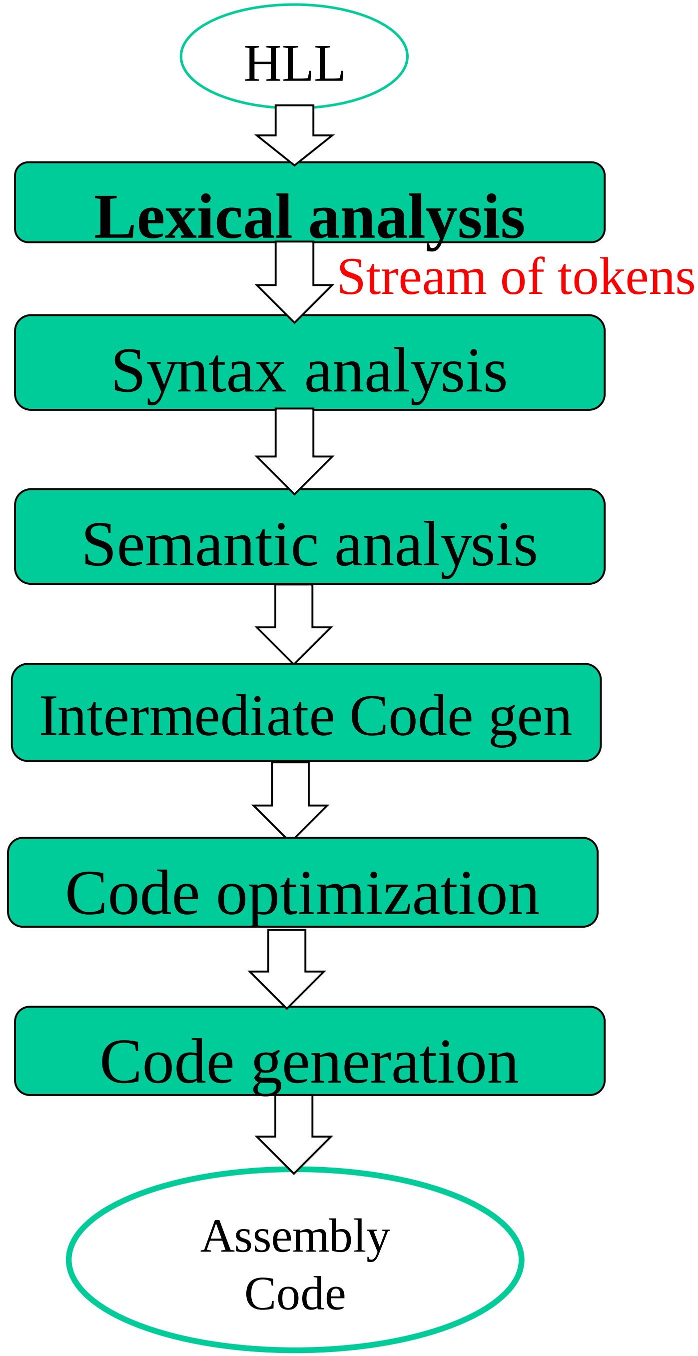
Execution of a Program

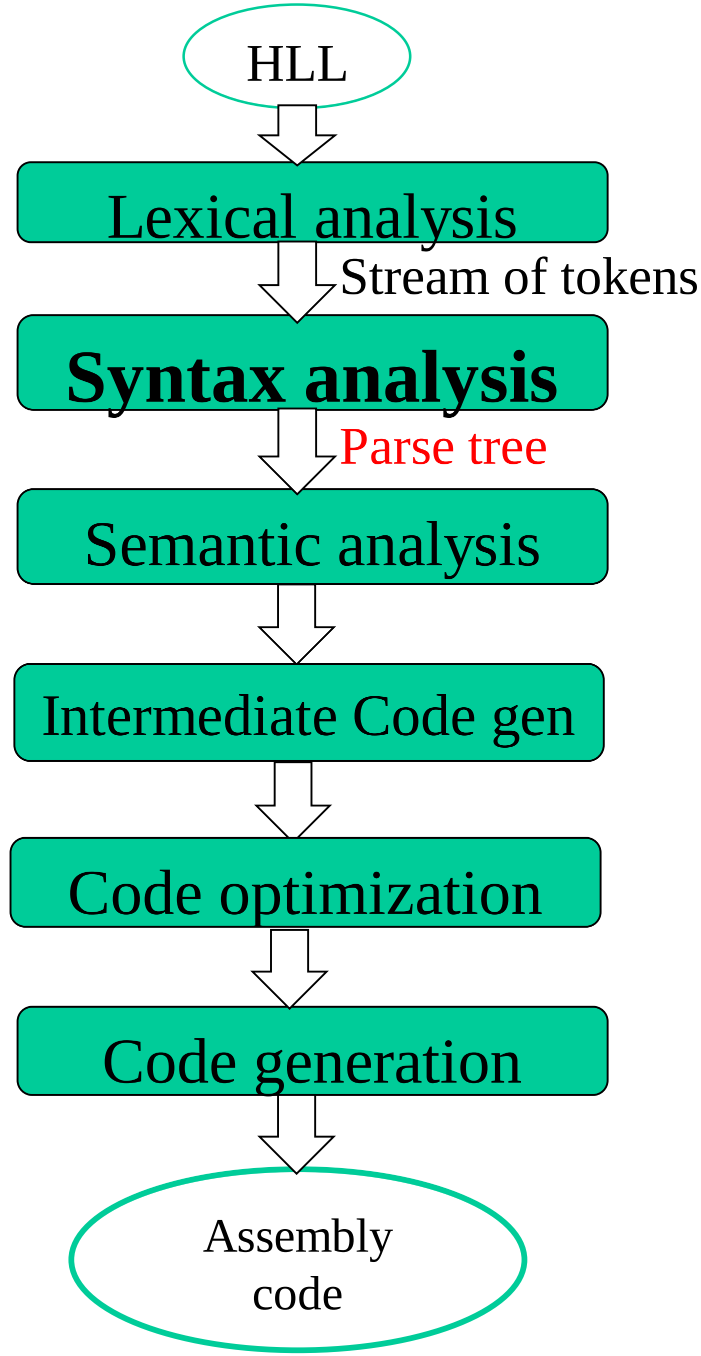


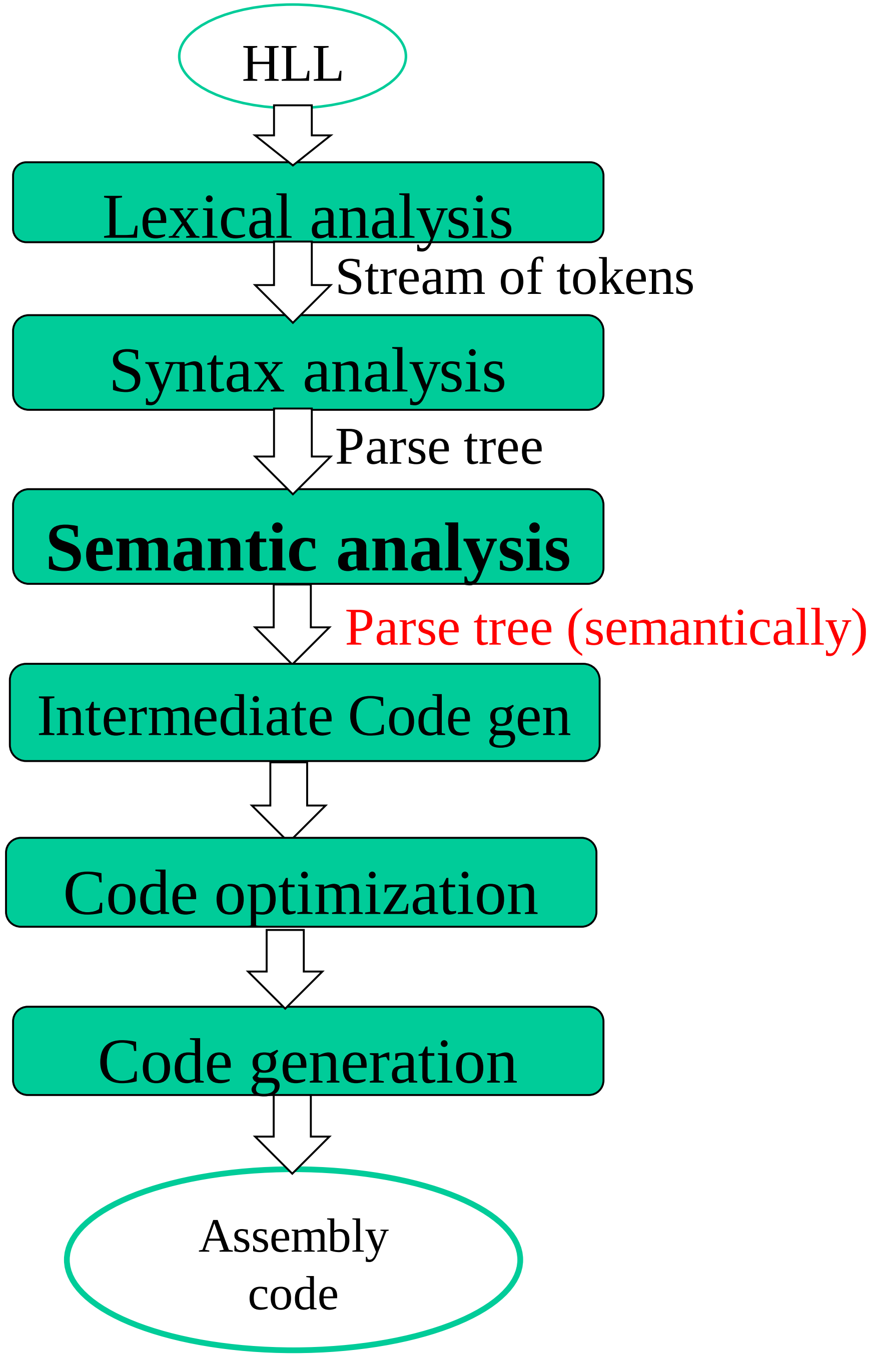
Execution of a Program

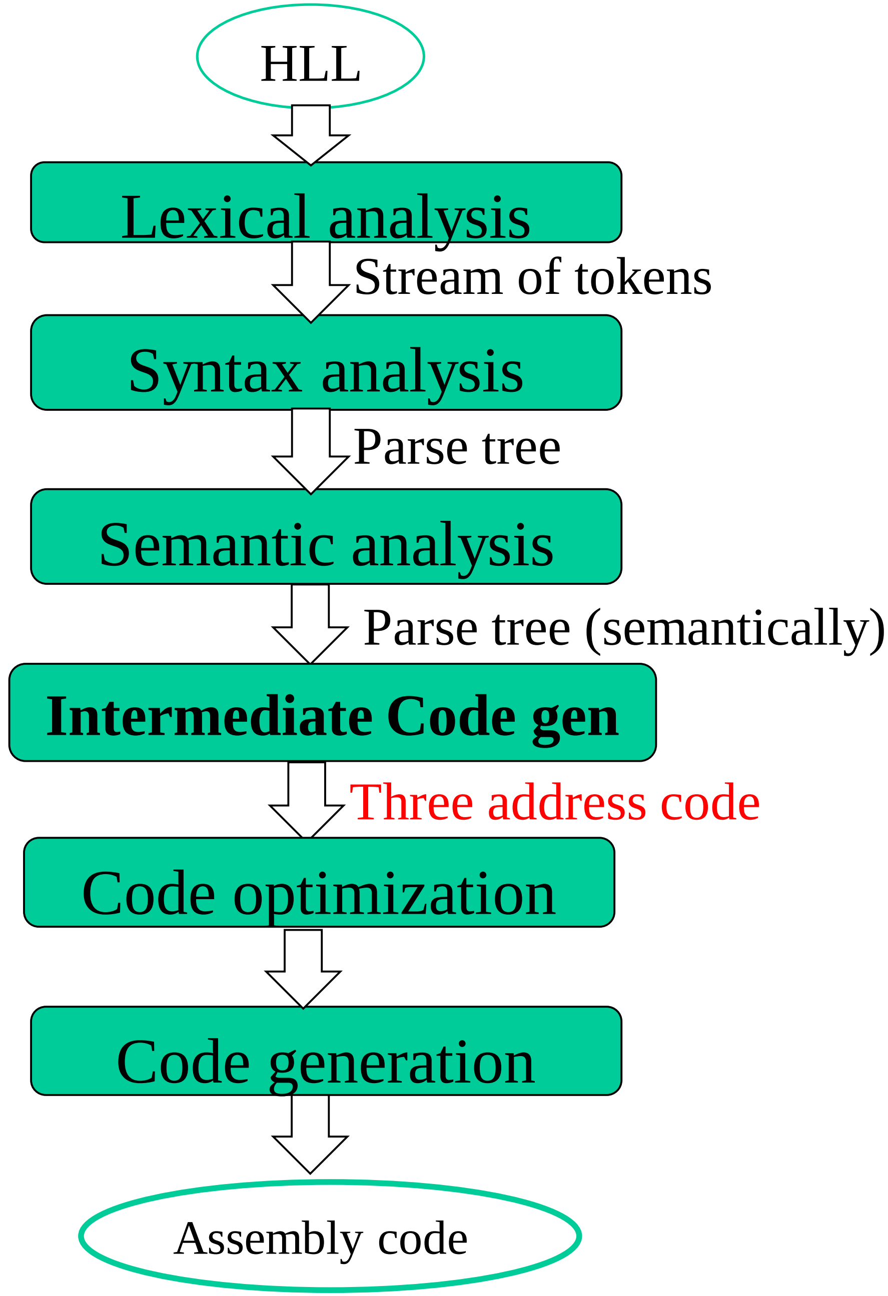


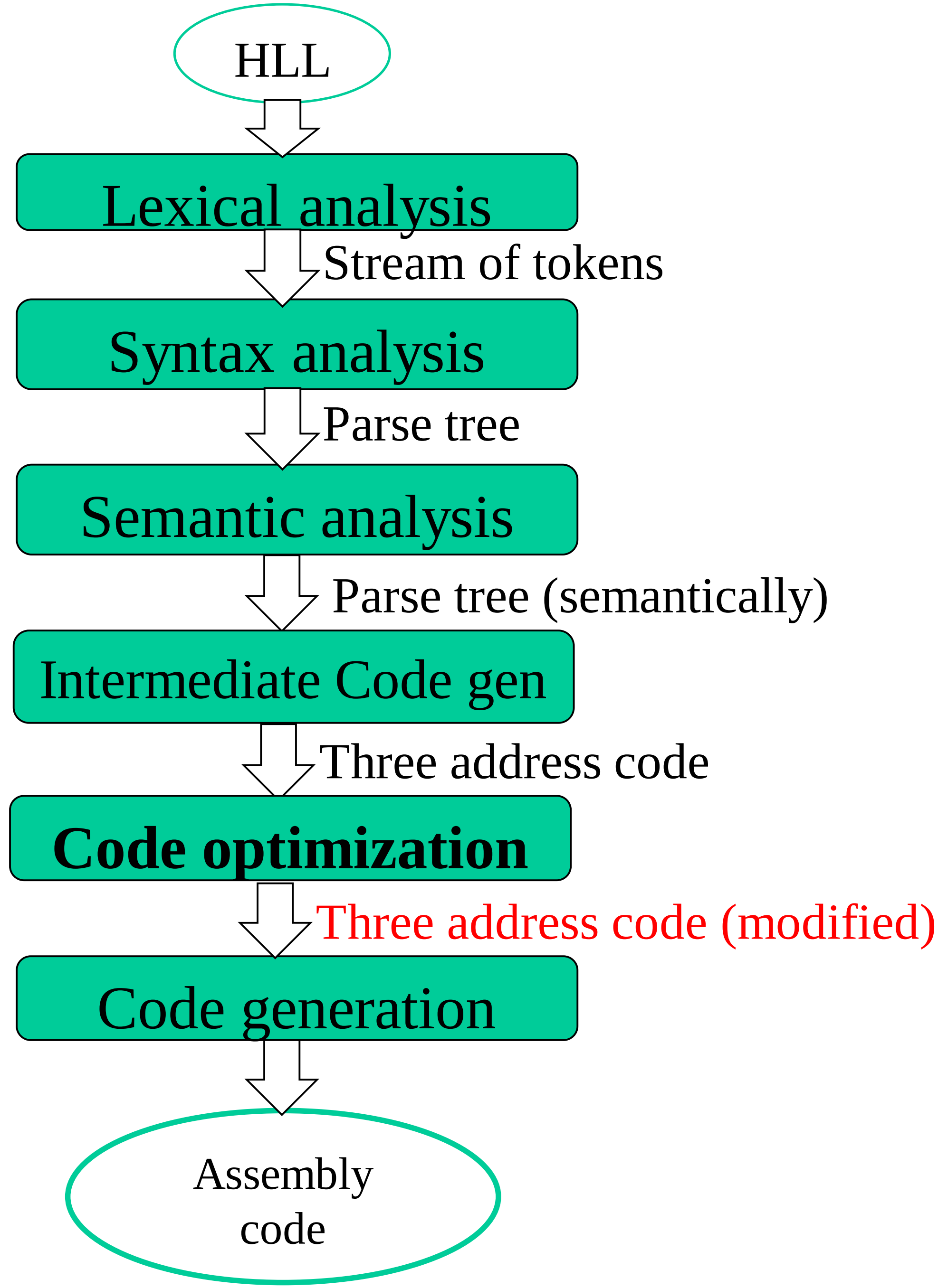
Phases (or structure) of Compiler

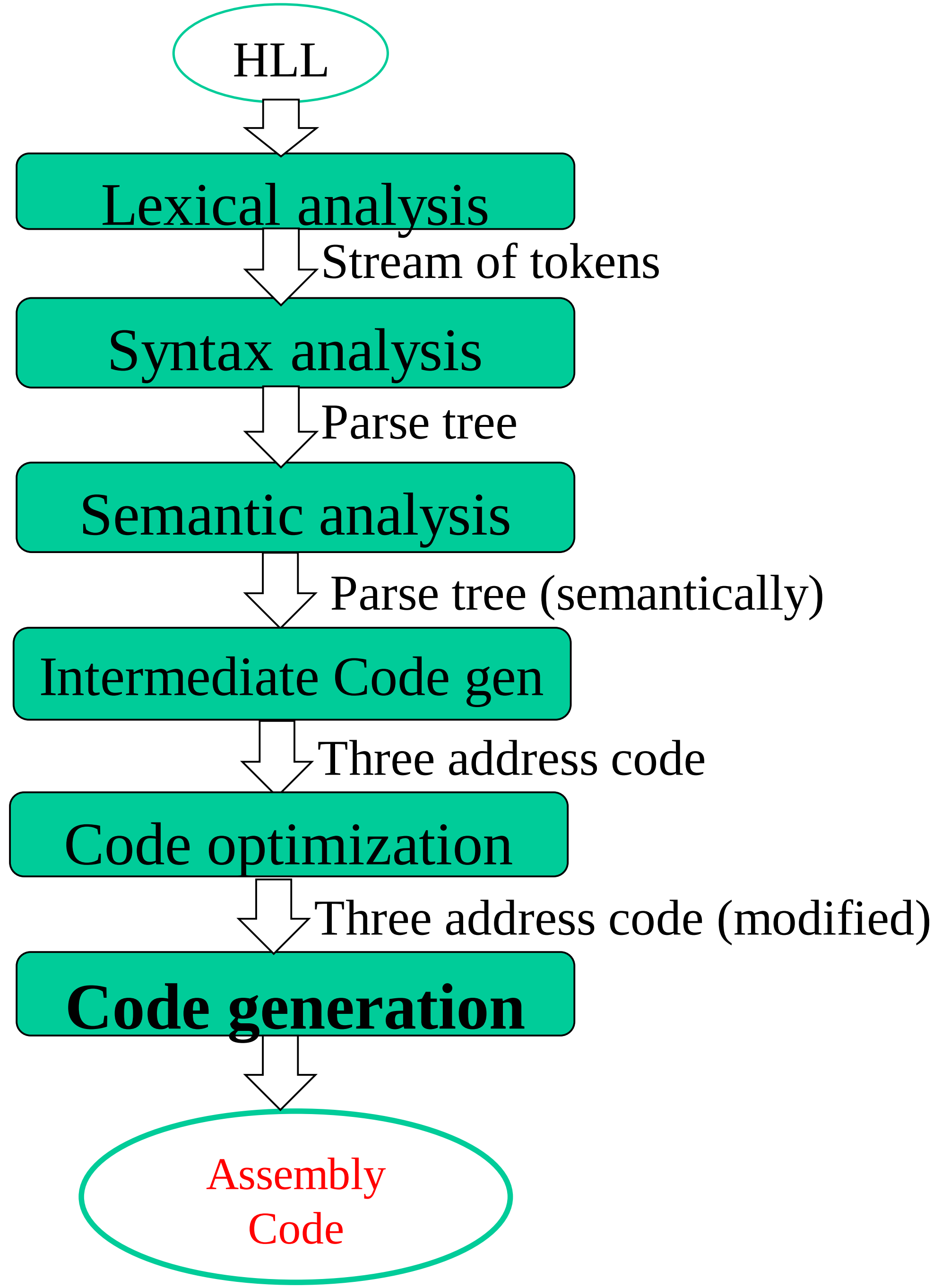

HLL
Lexical analysis
Stream of tokens
Syntax analysis
Semantic analysis
Intermediate Code gen
Code optimization
Code generation
Assembly Code

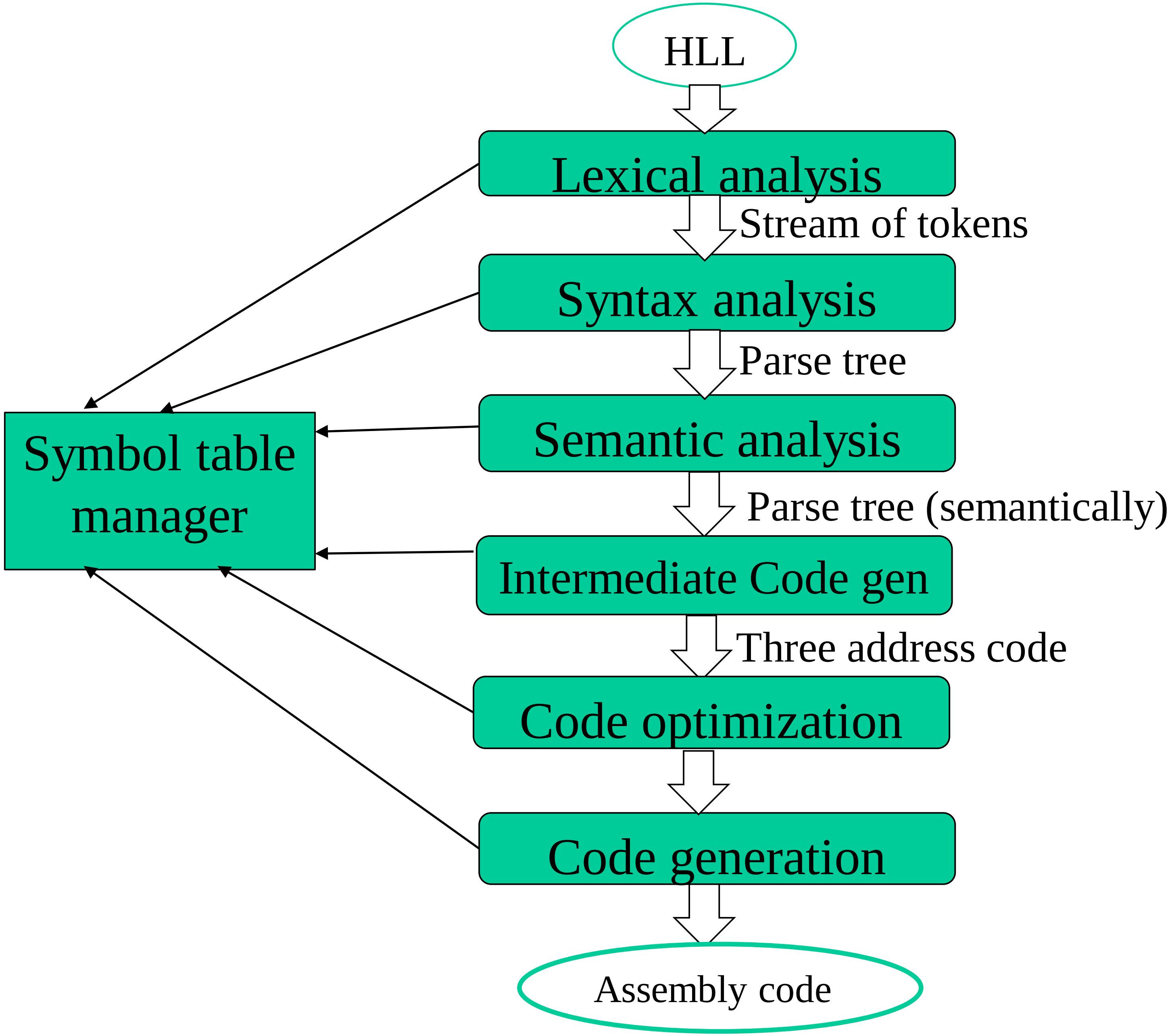

HLL
Lexical analysis
Stream of tokens
Syntax analysis
Parse tree
Semantic analysis
Intermediate Code gen
Code optimization
Code generation
Assembly code

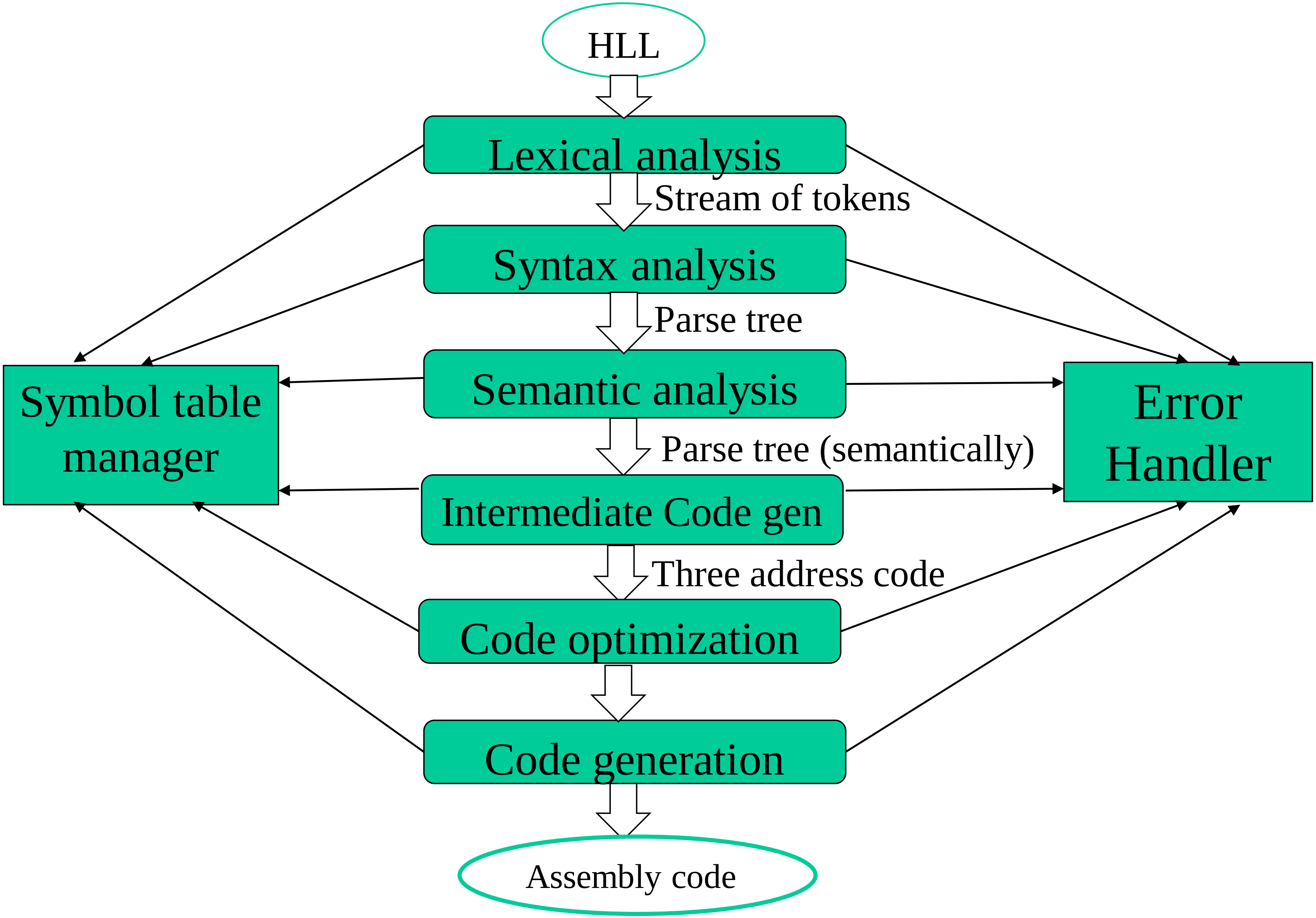

HLL
Lexical analysis
Stream of tokens
Syntax analysis
Parse tree
Semantic analysis
Parse tree (semantically)
Intermediate Code gen
Code optimization
Code generation
Assembly
code


HLL
Lexical analysis
Stream of tokens
Syntax analysis
Parse tree
Semantic analysis
Parse tree (semantically)
Intermediate Code gen
Three address code
Code optimization
Code generation
Assembly code


HLL
Lexical analysis
Stream of tokens
Syntax analysis
Parse tree
Semantic analysis
Parse tree (semantically)
Intermediate Code gen
Three address code
Code optimization
Three address code (modified)
Code generation
Assembly code


HLL
Lexical analysis
Stream of tokens
Syntax analysis
Parse tree
Semantic analysis
Parse tree (semantically)
Intermediate Code gen
Three address code
Code optimization
Three address code (modified)
Code generation
Assembly
Code


HLL
Lexical analysis
Stream of tokens
Syntax analysis
Parse tree
Semantic analysis
Parse tree (semantically)
Intermediate Code gen
Three address code
Code optimization
Code generation
Assembly code
Symbol table manager


HLL
Lexical analysis
Stream of tokens
Syntax analysis
Parse tree
Semantic analysis
Parse tree (semantically)
Intermediate Code gen
Three address code
Code optimization
Code generation
Assembly code
Symbol table manager
Error Handler

Lexical Analysis (Scanning)

- Reads characters in the source program and groups them into words (basic unit of syntax)
- Produces words and recognises what sort they are.
- The output is called token and is a pair of the form <*type*, *lexeme*> or <*token_class*, *attribute*>
- E.g.: **`a=b+c`** becomes <id,**`a`**> <=,> <id,**`b`**> <+,> <id,**`c`**>
- Needs to record each id attribute: keep a **symbol table**.
- Lexical analysis eliminates white space, etc…
- Speed is important - use a specialised tool: e.g., flex - a tool for generating <u>**scanners**</u>: programs which recognise lexical patterns in text; for more info: **`% man flex`**

Syntax (or syntactic) Analysis (Parsing)

- Imposes a hierarchical structure on the token stream.
- This hierarchical structure is usually expressed by recursive rules.
- Context-free grammars formalise these recursive rules and guide syntax analysis.
- Example:

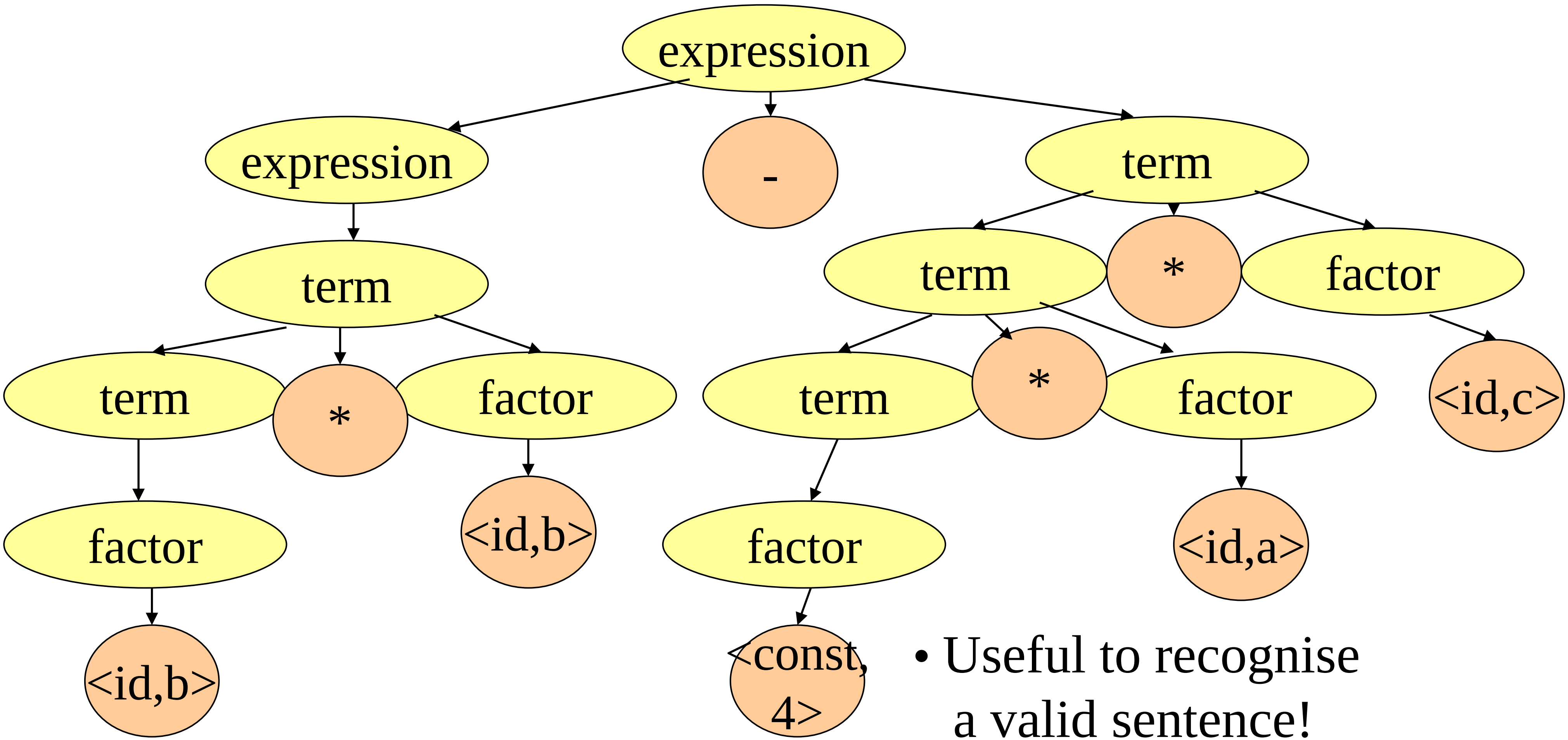
expression → expression '+' term | expression '-' term | term

term → term '*' factor | term '/' factor | factor

factor → identifier | constant | '(' expression ')'

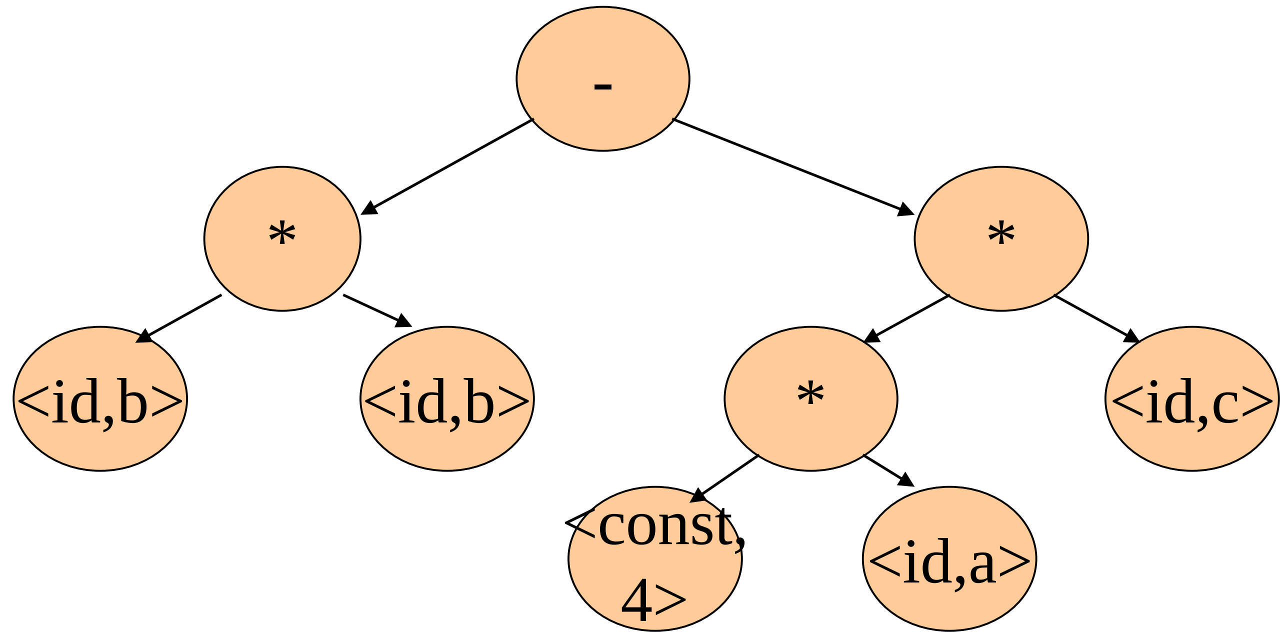
(this grammar defines simple algebraic expressions)

Parsing: parse tree for *b*b-4*a*c*



- Useful to recognise a valid sentence!
- Contains a lot of unneeded information!

AST for *b*b-4*a*c*



- An Abstract Syntax Tree (AST) is a more useful data structure for internal representation. It is a compressed version of the parse tree (summary of grammatical structure without details about its derivation)
- ASTs are one form of IR

Semantic Analysis (context handling)

- Collects context (semantic) information, checks for semantic errors, and annotates nodes of the tree with the results.
- Examples:
  - type checking: report error if an operator is applied to an incompatible operand.
  - check flow-of-controls.
  - uniqueness or name-related checks.

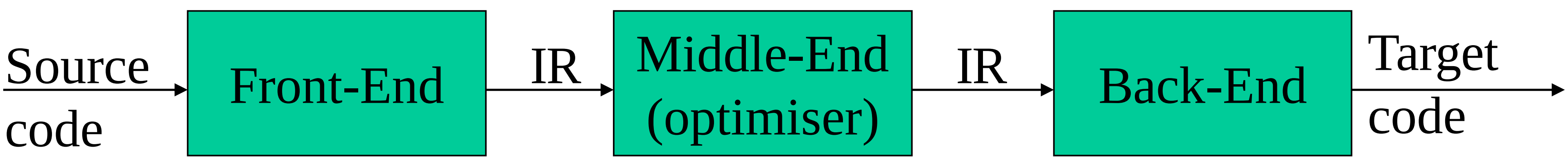
Intermediate code generation

- Translate language-specific constructs in the AST into more general constructs.
- A criterion for the level of "generality": it should be straightforward to generate the target code from the intermediate representation chosen.
- Example of a form of IR (3-address code):

```
tmp1=4
tmp2=tmp1*a
tmp3=tmp2*c
tmp4=b*b
tmp5=tmp4-tmp3
```

Code Optimisation

- The goal is to improve the intermediate code and, thus, the effectiveness of code generation and the performance of the target code.
- Optimisations can range from trivial (e.g. constant folding) to highly sophisticated (e.g, in-lining).
- For example: replace the first two statements in the example of the previous slide with: **`tmp2=4*a`**
- Modern compilers perform such a range of optimisations, that one could argue for:



Code Generation Phase

- Map the AST onto a linear list of target machine instructions in a symbolic form:
  - Instruction selection: a pattern matching problem.
  - Register allocation: each value should be in a register when it is used (but there is only a limited number): NP-Complete problem.
  - Instruction scheduling: take advantage of multiple functional units: NP-Complete problem.
- Target, machine-specific properties may be used to optimise the code.
- Finally, machine code and associated information required by the Operating System are generated.

Phases with example

$x = a + b * c;$ /*statement*/

⇓

Lexical analyser $\quad l(l+d)^*$

⇓

$id = id + id * id;$

⇓

Syntax analyser ⇐

$S \rightarrow id = E;$
$E \rightarrow E + T / T$
$T \rightarrow T * F / F$
$F \rightarrow id$

⇓

(parse tree)

⇒ **Semantic analyser**

⇓

(Parse tree Semantic...)

⇓

ICG

⇓

$t_1 = b * c;$
$t_2 = a + t_1;$
$x = t_2;$

⇓

ICO

⇓

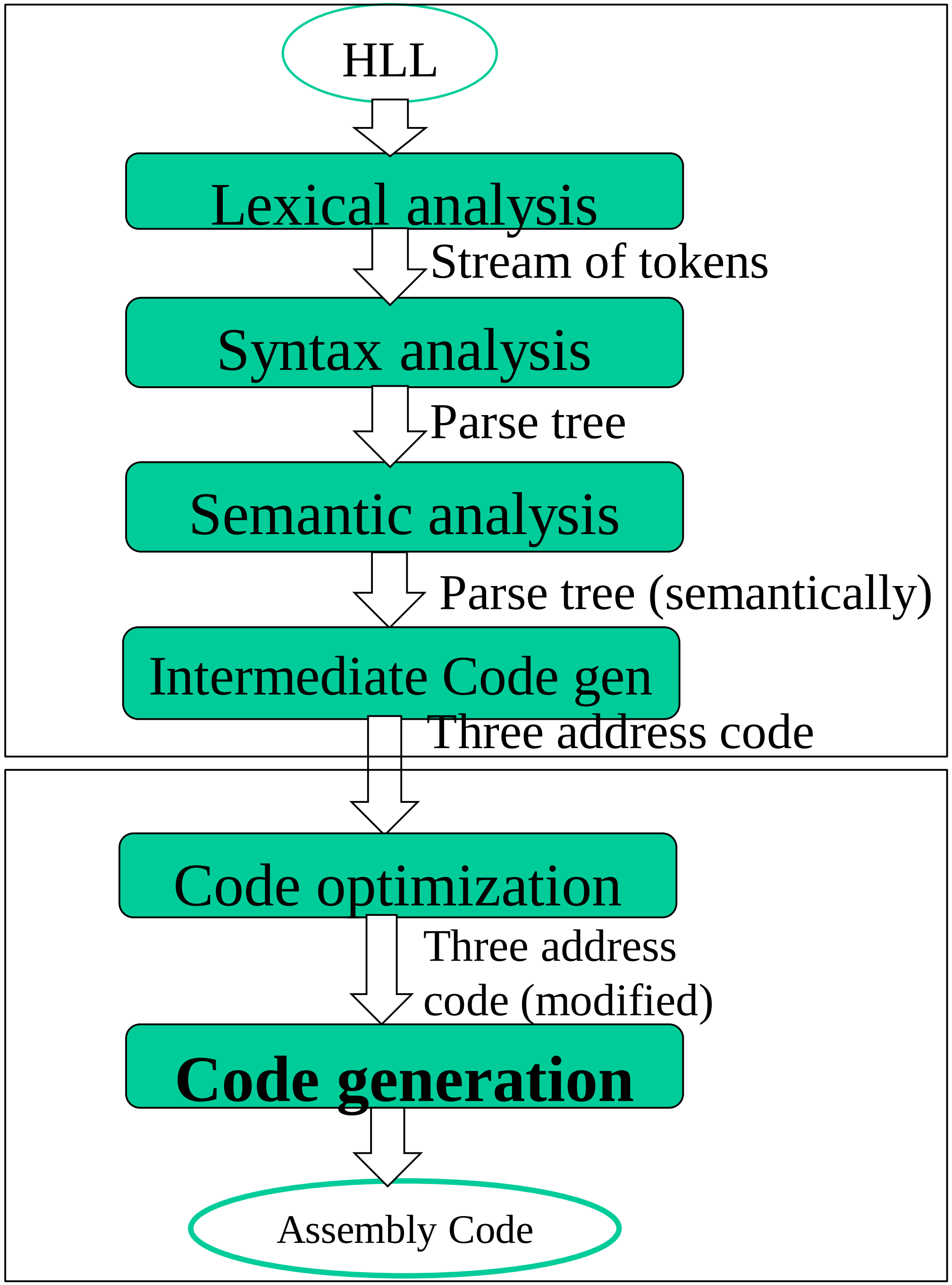
$t_1 = b * c;$
$t_2 = a + t_1;$

⇓

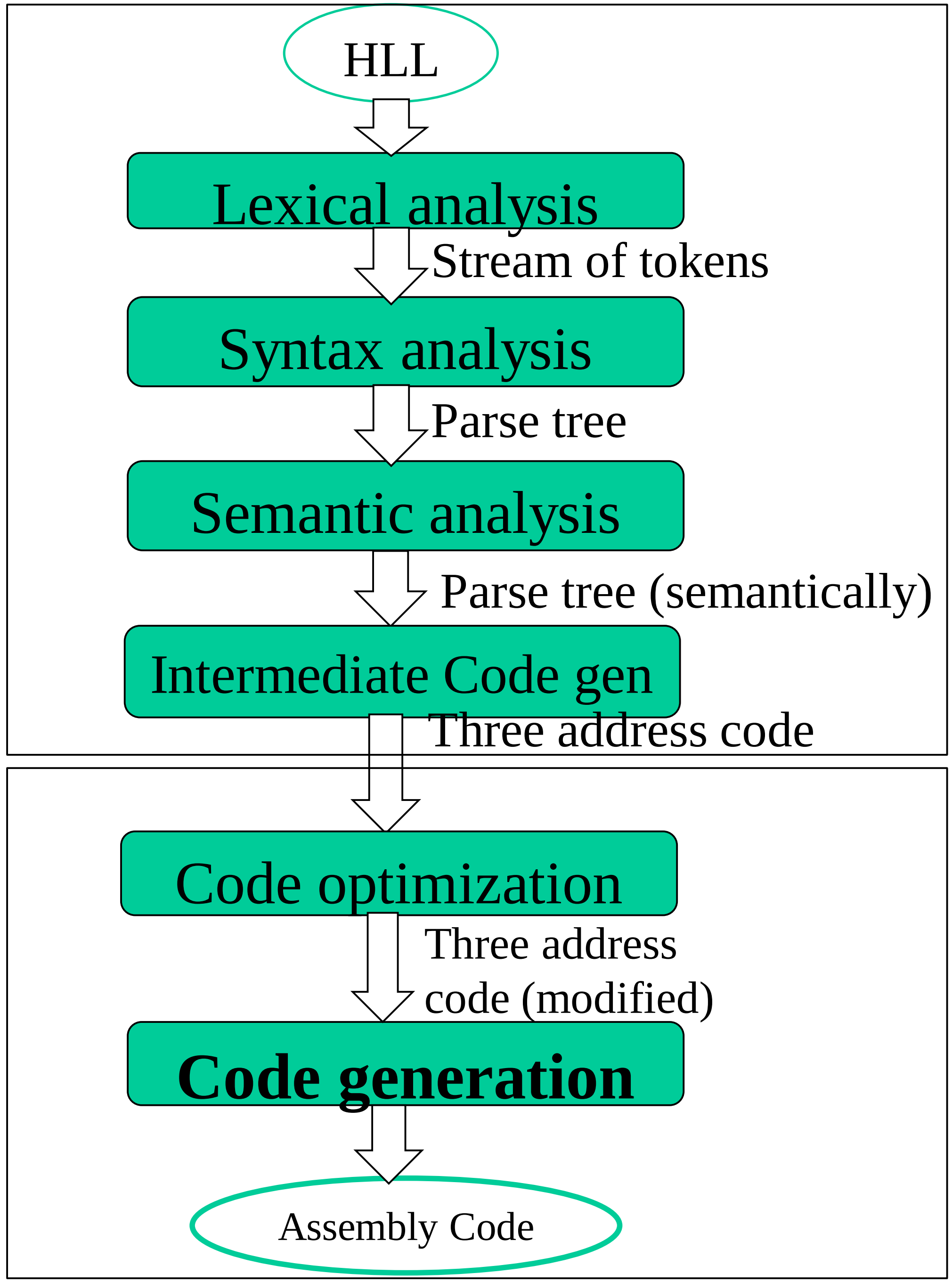
TCG

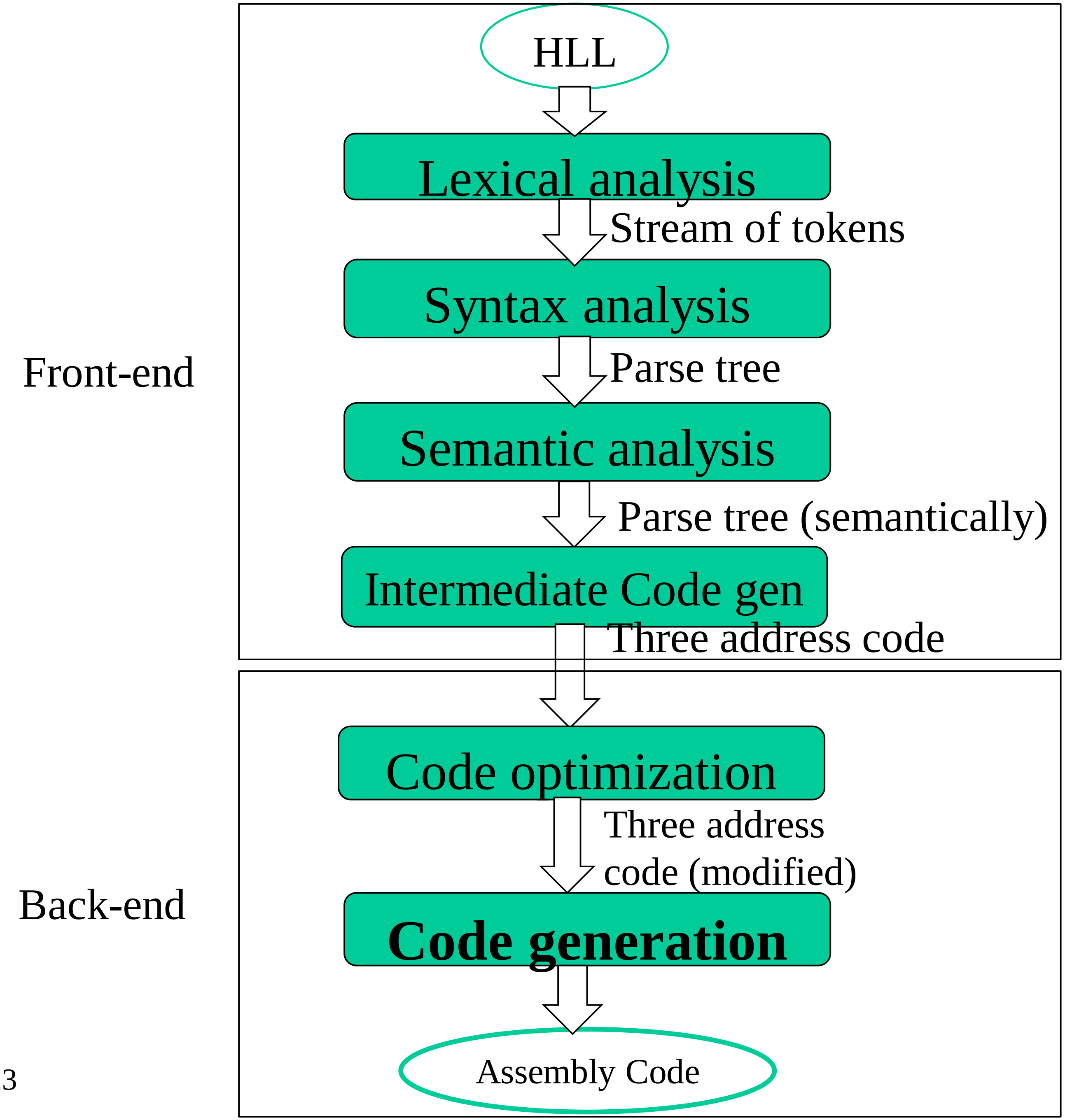
⇓

mul	R_1, R_2	$a \rightarrow R_0$
add	R_0, R_2	$b \rightarrow R_1$
mov	R_2, x	$c \rightarrow R_2$

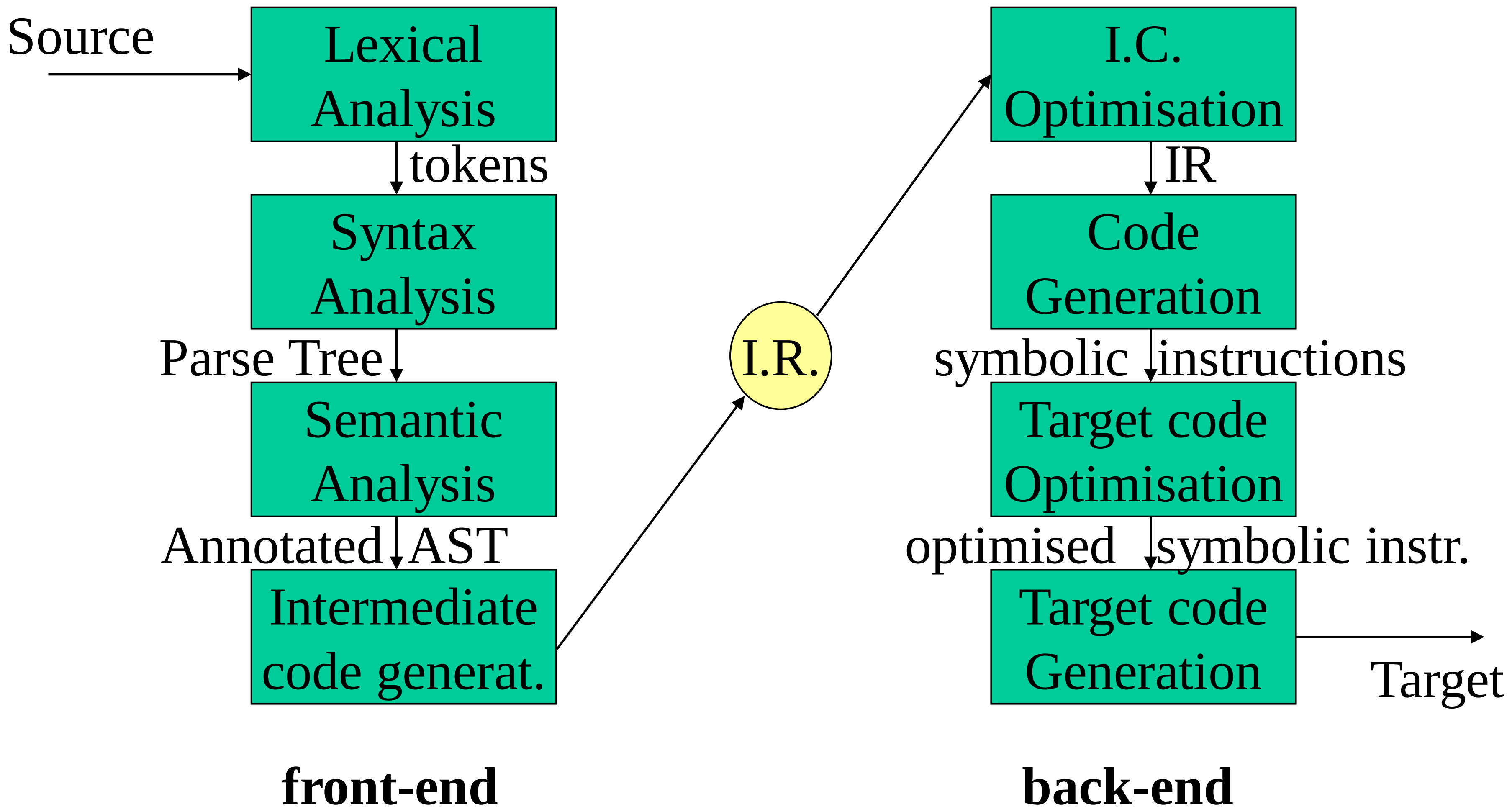

HLL
Lexical analysis
Stream of tokens
Syntax analysis
Parse tree
Semantic analysis
Parse tree (semantically)
Intermediate Code gen
Three address code
Code optimization
Three address
code (modified)
Code generation
Assembly Code

Front-end


HLL
Lexical analysis
Stream of tokens
Syntax analysis
Parse tree
Semantic analysis
Parse tree (semantically)
Intermediate Code gen
Three address code
Code optimization
Three address
code (modified)
Code generation
Assembly Code


HLL
Lexical analysis
Stream of tokens
Syntax analysis
Parse tree
Semantic analysis
Parse tree (semantically)
Intermediate Code gen
Three address code
Front-end
Code optimization
Three address
code (modified)
Code generation
Back-end
Assembly Code

General Structure of a compiler

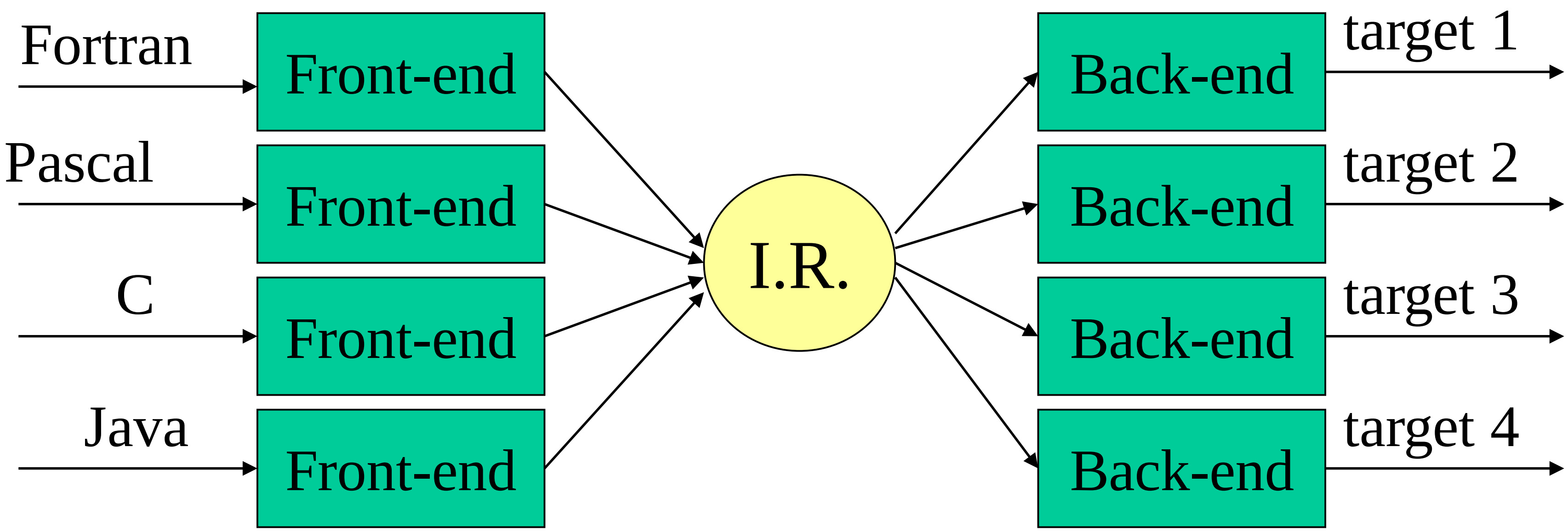


Conceptual Structure:two major phases

- **Front-end** performs the **analysis** of the source language:
  - Recognises legal and illegal programs and reports errors.
  - “understands” the input program and collects its semantics in an IR.
  - Produces IR and shapes the code for the back-end.
  - Much can be automated.

- **Back-end** does the target language **synthesis**:
  - Chooses instructions to implement each IR operation.
  - Translates IR into target code.
  - Needs to conform with system interfaces.
  - Automation has been less successful.

m×n compilers with m+n components!



- All language specific knowledge must be encoded in the front-end
- All target specific knowledge must be encoded in the back-end

But: in practice, this strict separation is not free of charge.

Qualities of a Good Compiler

What qualities would you want in a compiler?

- generates correct code (first and foremost!)
- generates fast code
- conforms to the specifications of the input language
- copes with essentially arbitrary input size, variables, etc.
- compilation time (linearly)proportional to size of source
- good diagnostics
- consistent optimisations
- works well with the debugger

Principles of Compilation

The compiler must:

- *preserve the meaning of the program being compiled*.
- *"improve" the source code in some way*.

Other issues (depending on the setting):

- Speed (of compiled code)
- Space (size of compiled code)
- Feedback (information provided to the user)
- Debugging (transformations obscure the relationship source code vs target)
- Compilation time efficiency (fast or slow compiler?)

Historical Notes:

The Move to Higher-Level Programming Languages

- Machine Languages (1st generation)
- Assembly Languages (2nd generation) – early 1950s
- High-Level Languages (3rd generation) – later 1950s
- 4th generation higher level languages (SQL, Postscript)
- 5th generation languages (logic based, eg, Prolog)
- Other classifications:
  - Imperative (how); declarative (what)
  - Object-oriented languages
  - Scripting languages

Finally...

Parts of a compiler can be generated automatically using generators based on formalisms. E.g.:

- Scanner generators: flex
- Parser generators: bison