Compiler Design
(CST309)

Dinesh Gopalani
dgopalani.cse@mnit.ac.in

Protocol for Online Mode

- NO late joining will be allowed
- Attendance will be as per your joining
- You shall attend and not your siblings or parents
- You shall always use a Notebook and a pen
- There will be a quiz at the end of each lecture
- You shall keep your microphone mute and camera off
- For asking any doubt you shall unmute your microphone and camera on

Course Scheme

- Credits

  3

- Teaching

  3-0-0

- Assessment

  Mid Term (30%)

  End Term (50%)

  Course Project and Quizzes (20%)

AIM

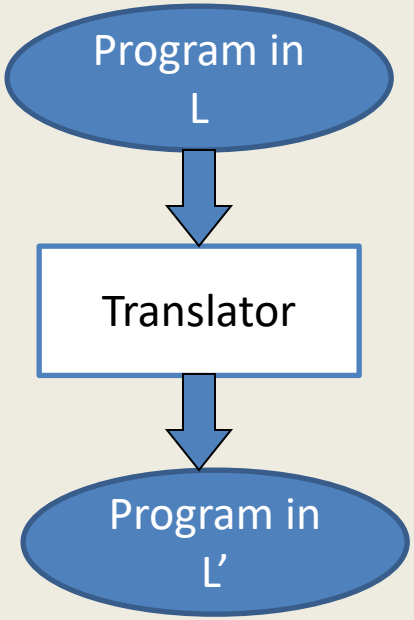
The aim of this course is to study and discuss various methods or techniques used in designing as well as implementing a language translator.

List of Books

- Principles of Compiler Design – Aho, Lam, Sethi, Ullman (Pearson Education)
- The Theory and Practice of Compiler Writing – Tremblay, Sorenson (BS Pub.)
- Engineering a Compiler – Cooper, Torczon (Morgan-Kaufmann)
- Lex and Yacc – Levine, Mason, Brown (O'Reilly)

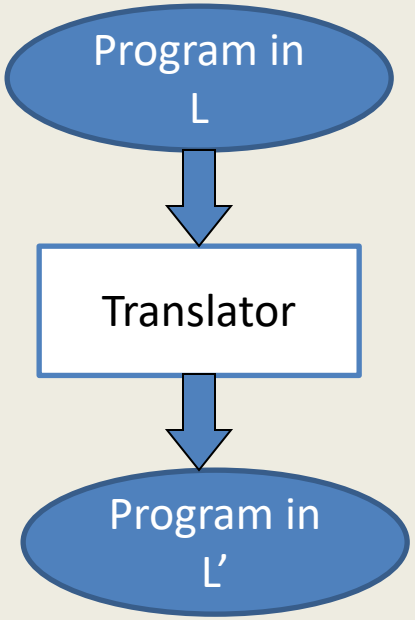
What is a Compiler?

- Translator



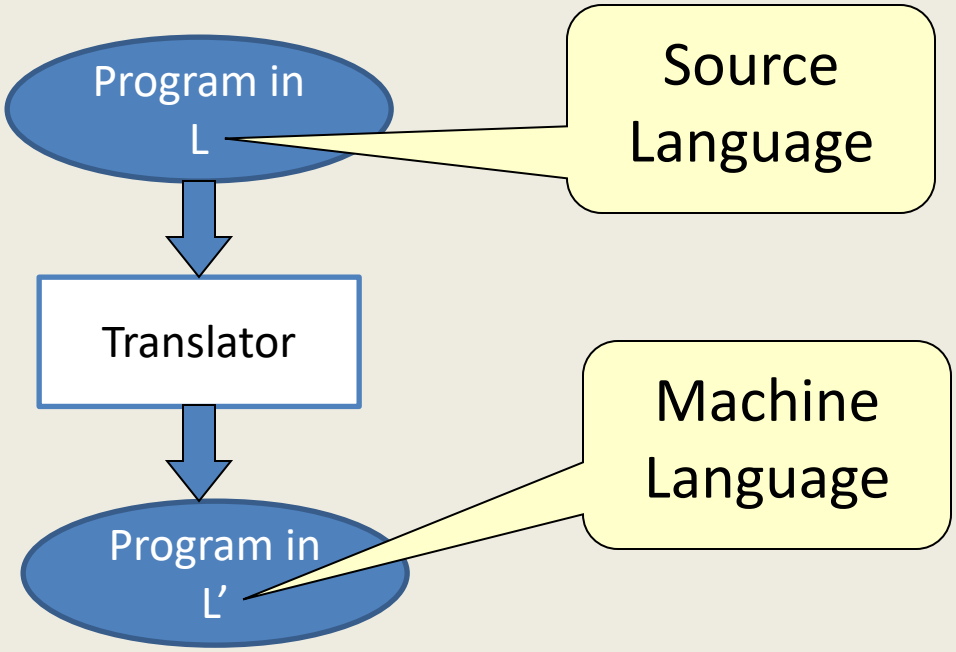
What is a Compiler?

- Translator



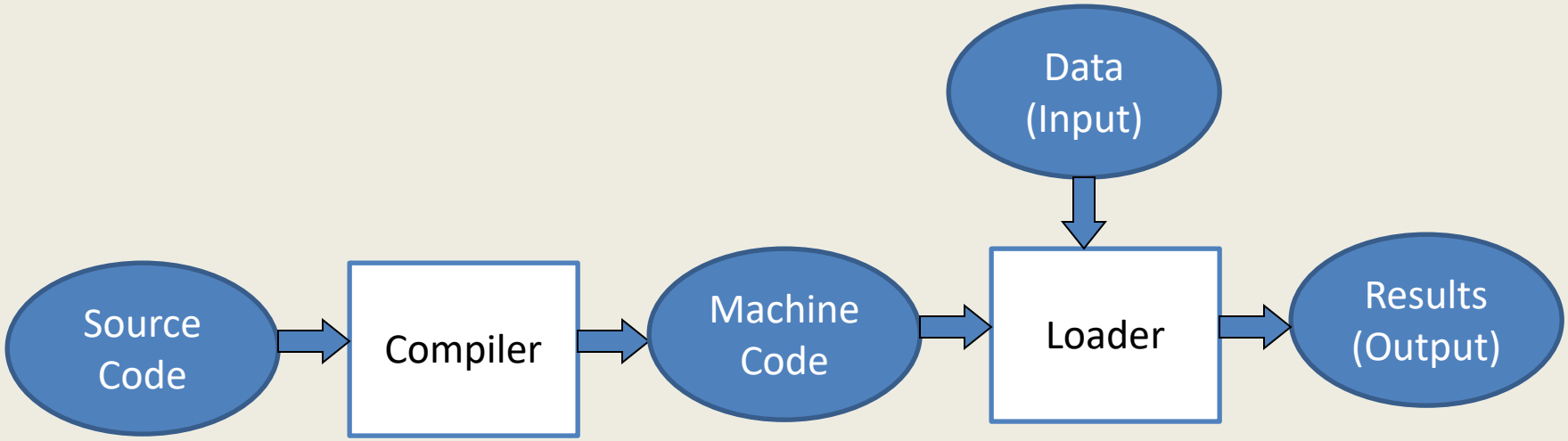
- When L is a High-level language (C, Java, …) and L' is a Low-level language (Assembly or Machine) then such translator is called a Compiler or an Interpreter.
- A translator may be implemented in two ways –
  - Compiler
  - Interpreter

What is a Compiler?

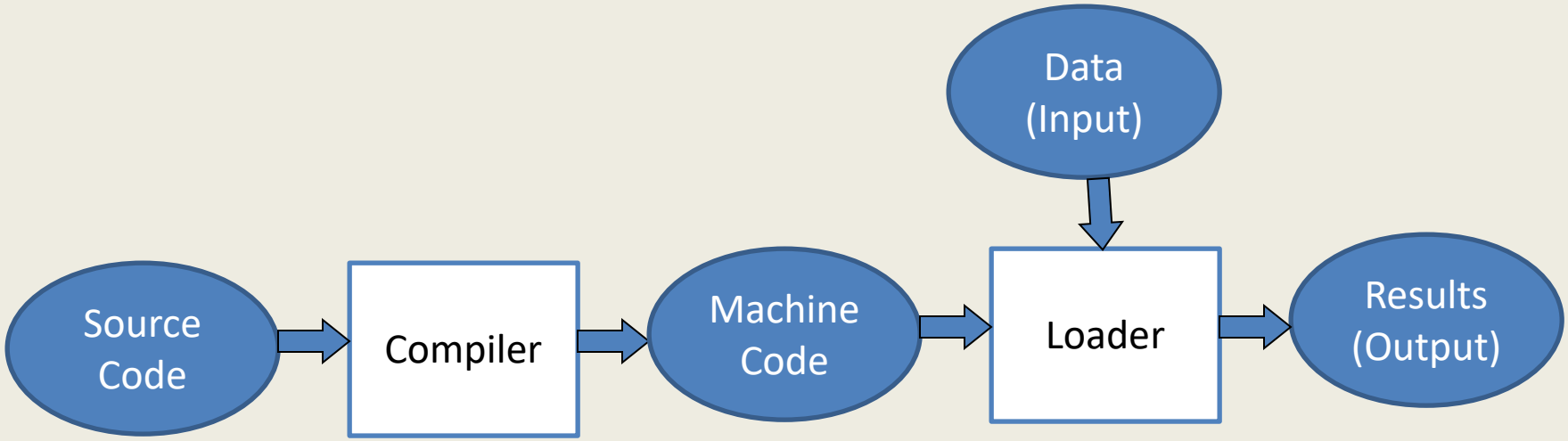


- When L is a High-level language (C, C++, Fortran, …) and L' is a Low-level language (Assembly or Machine) then such translator is called a Compiler or an Interpreter.
- A translator may be implemented in two ways –
  - Compiler
  - Interpreter

How a Compiler works?



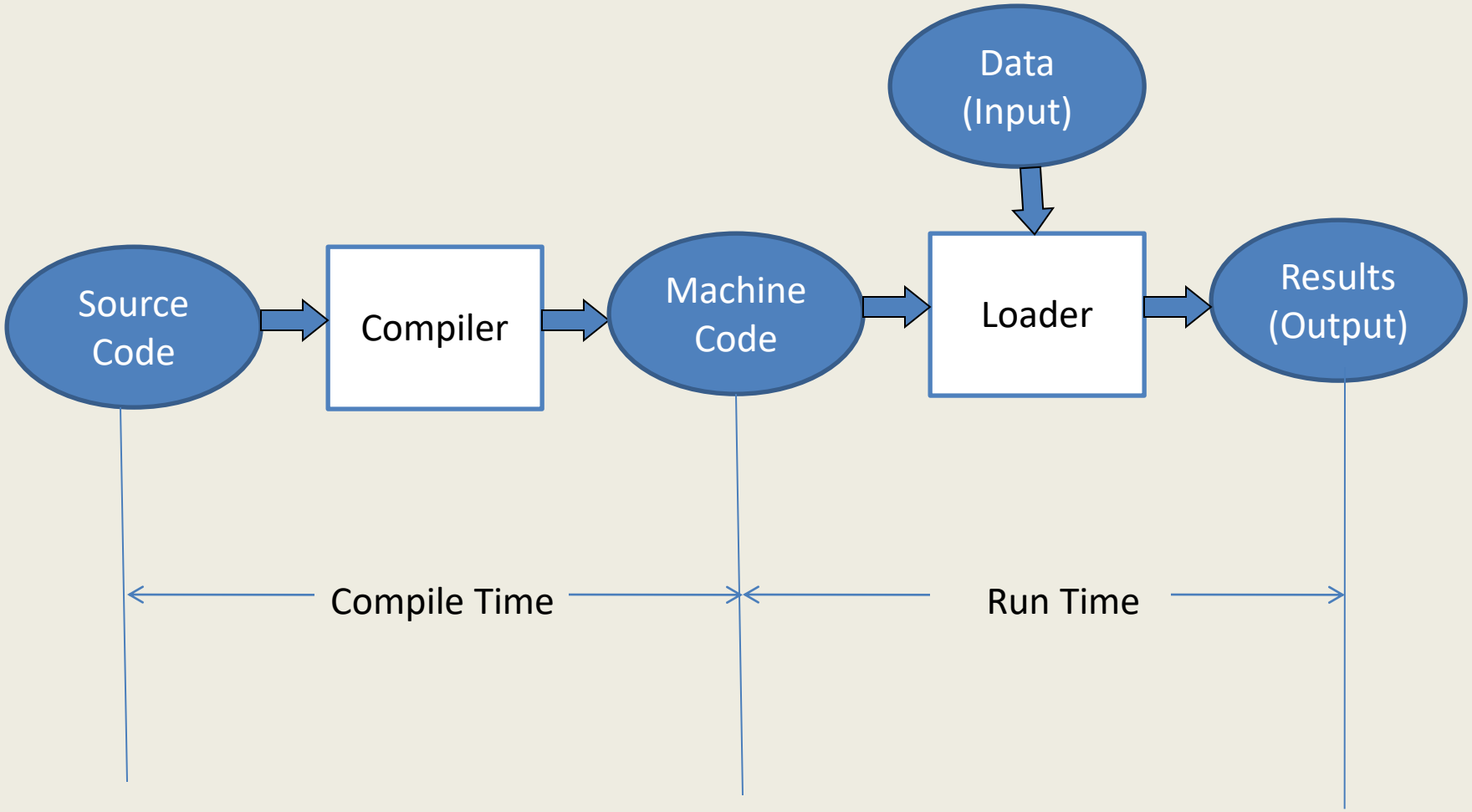
How a Compiler works?



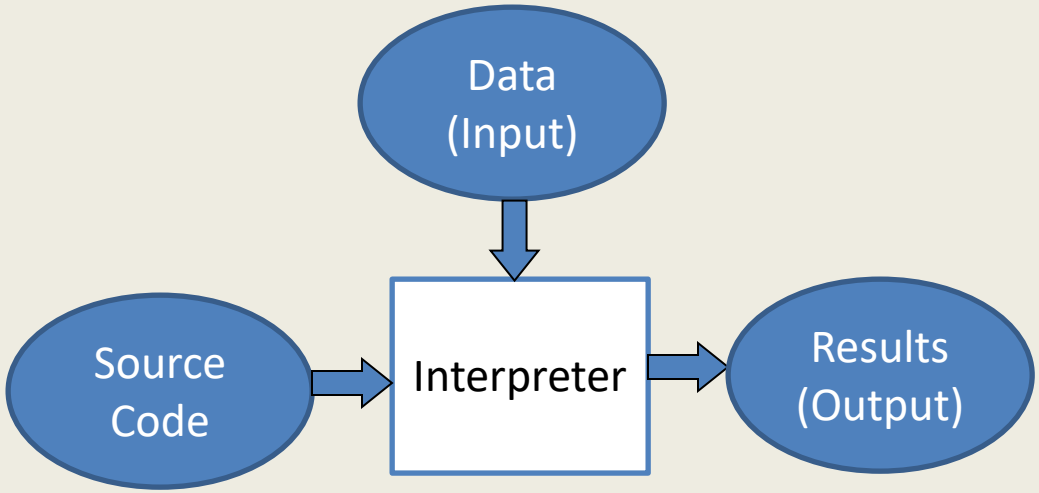
It is a two stage process –

Compilation followed by Execution

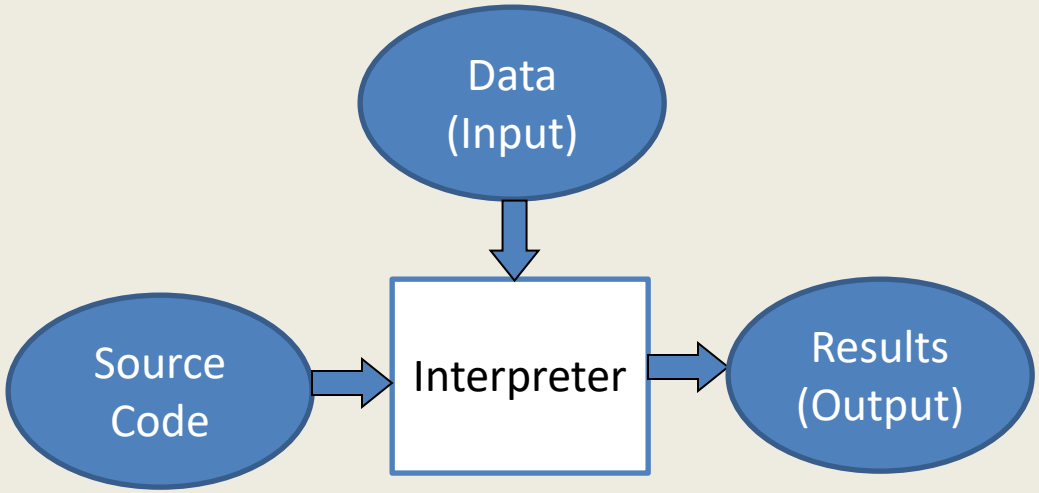
How a Compiler works?



How an Interpreter works?



How an Interpreter works?



Here code is translated on the fly, during Execution itself.

Compiler vs. Interpreter

Compiler

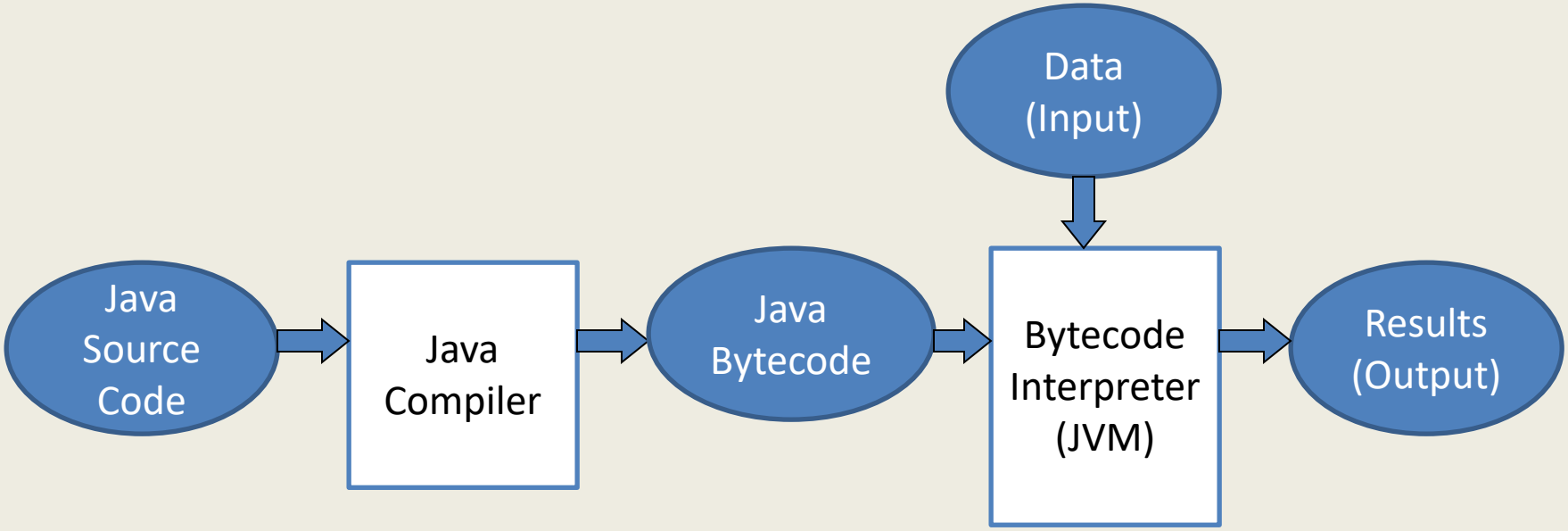
- Translation is done before execution starts
- Translates the entire code at a time
- Faster Execution
- Extensive code optimization possible
- No support of Dynamic Typing, Dynamic Scoping …
- Ex – C, C++, Ada, Algol, Fortran, Pascal, ….

Interpreter

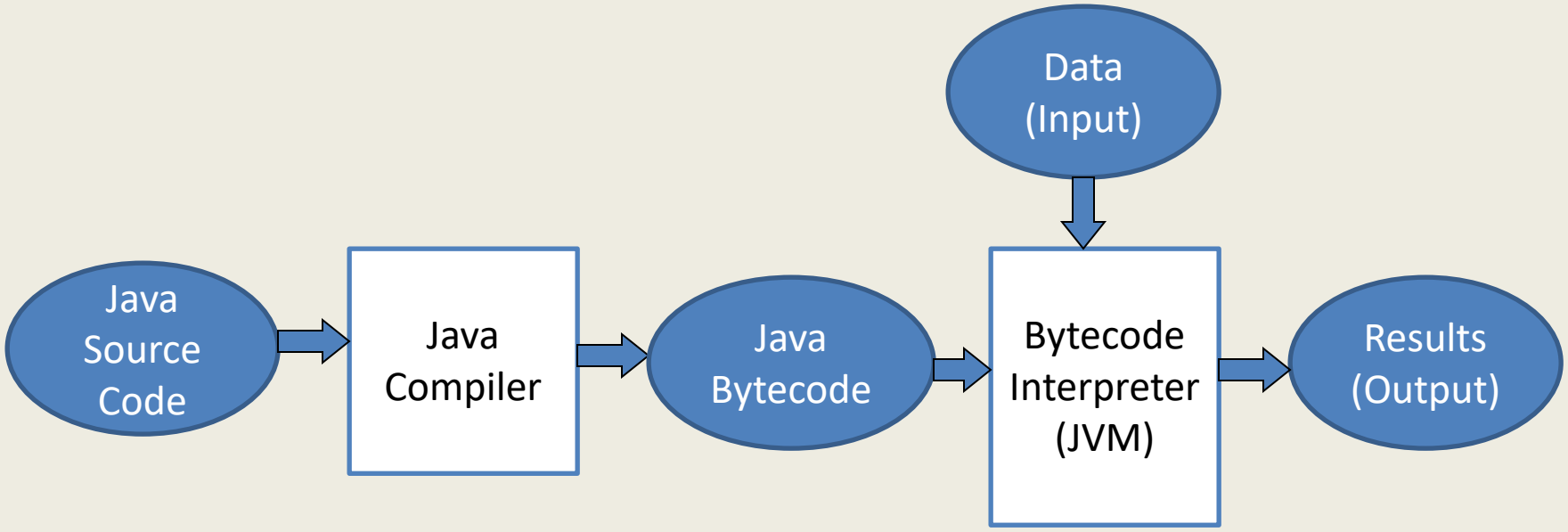
- Translation and Execution going side by side
- Translates the code line by line
- Slower Execution
- No or Least code optimization possible
- Supports Dynamic Typing, Dynamic Scoping …
- Ex – Basic, Lisp, APL, PHP, …

Hybrid Approach

Modern Languages – Java, Python, Ruby, … use both Compiler and Interpreter approaches.



Hybrid Approach



Compiler is same for different platforms, but Bytecode interpreters are different for different platforms – Platform Free

How about Run-Time Performance?

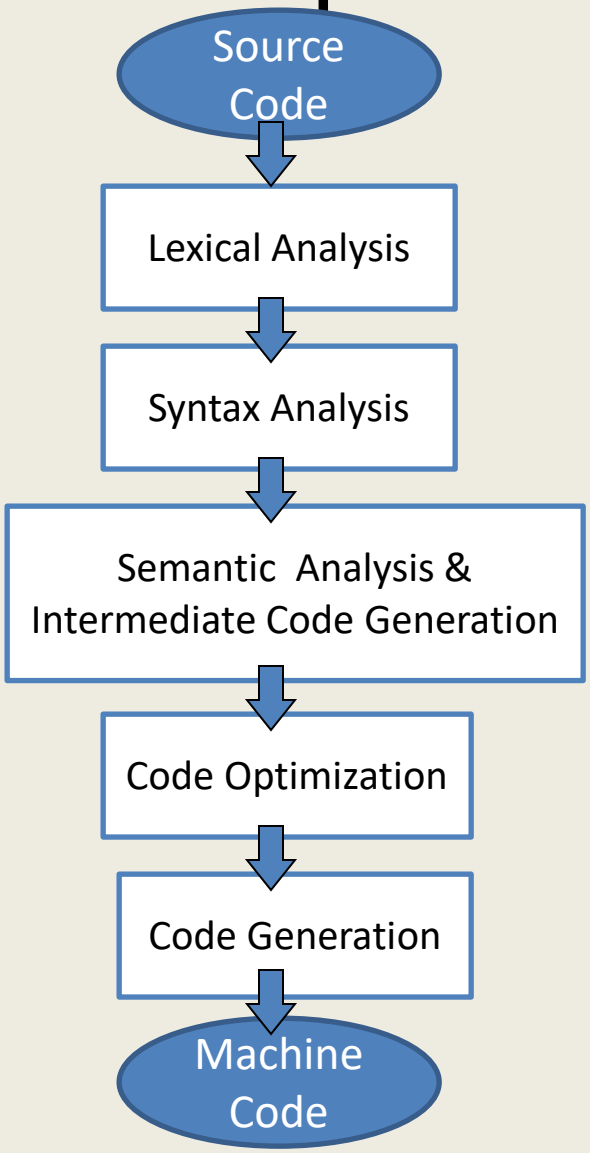
- The bytecode need to be compiled into machine language for execution and will increase Run-time.
- JIT (Just-In-Time) Translation:

  The bytecode is compiled when it is about to be executed, and then cached and may reused later without need of recompiling.

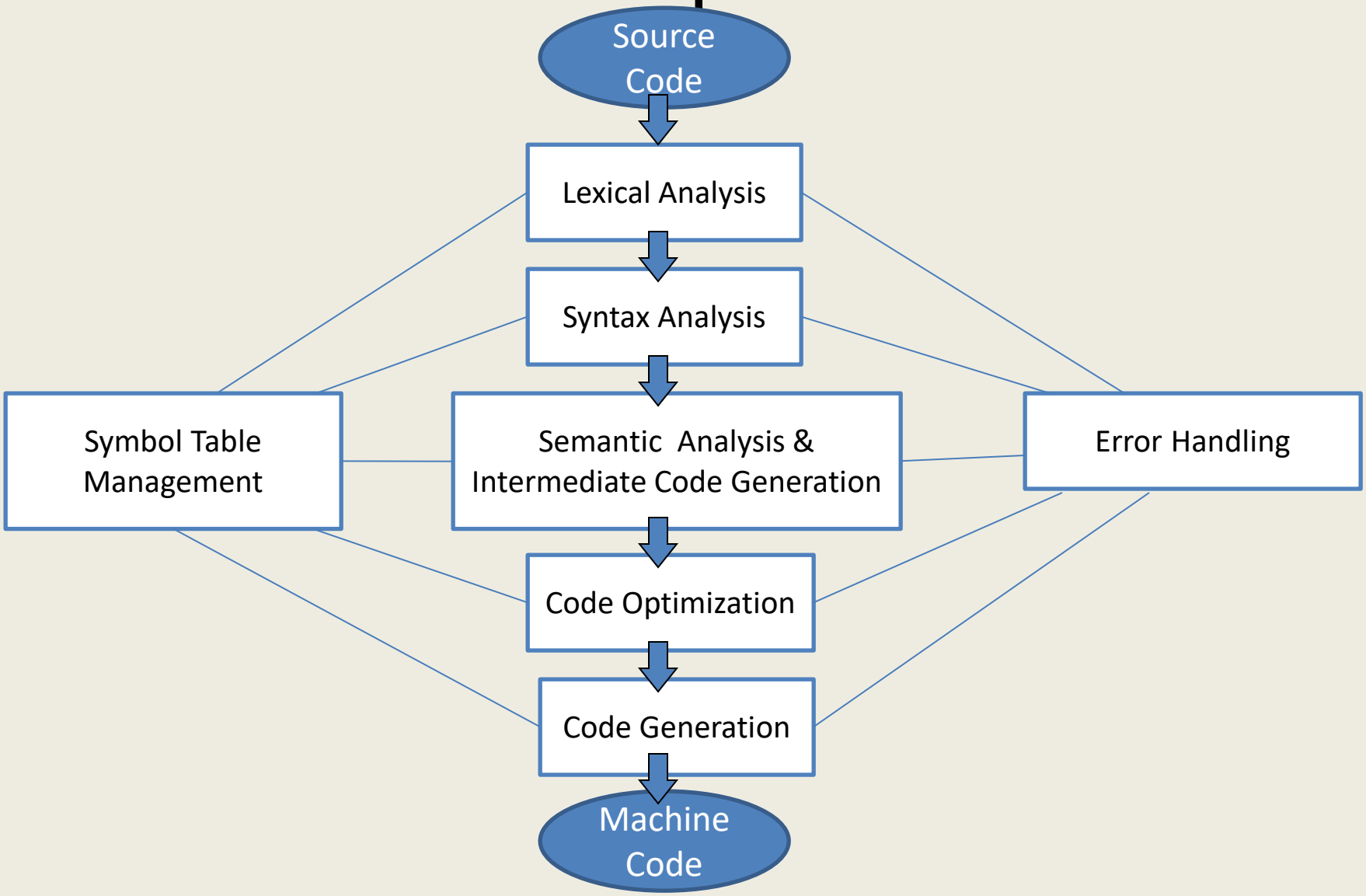
How a Compiler works?

- As compilation process is so complex that it is not convenient to perform in single step.
- The process is divided into a series of sub-processes – Phases.
- A Phase is a logical operation that takes as input one representation of the source code and produces as output another representation.

How a Compiler works?

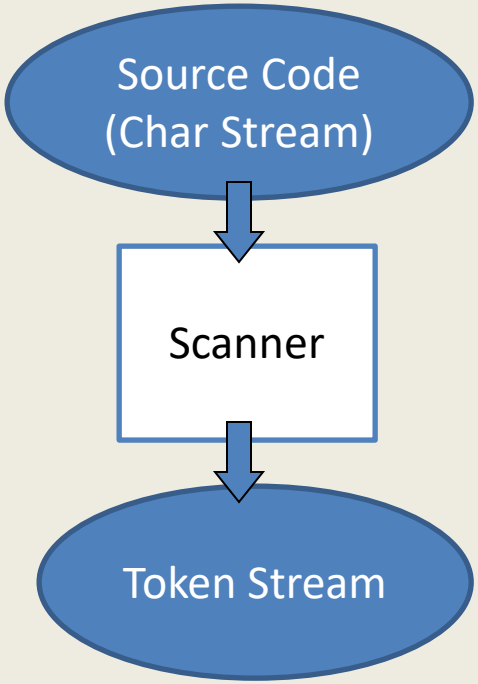


How a Compiler works?



Lexical Analyzer (Scanner)

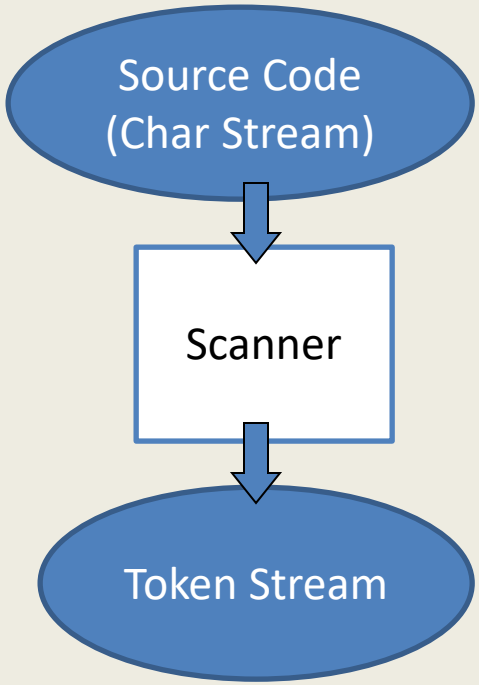
Forms the group of characters that logically belong together - Tokens



Lexical Analyzer (Scanner)

Example :

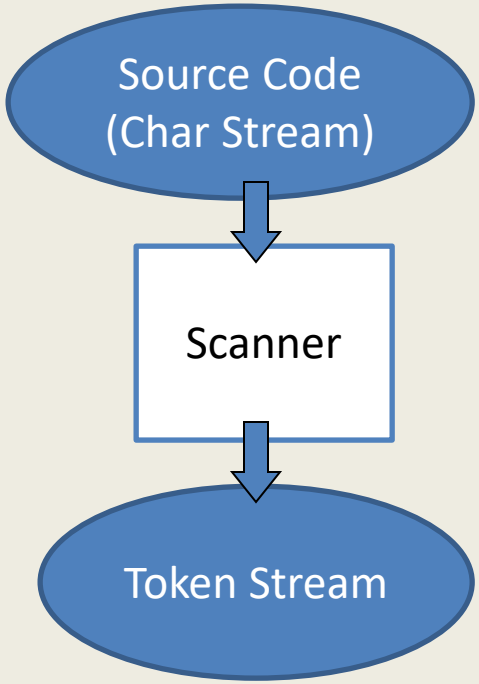
sum = sum + 100 ;



Lexical Analyzer (Scanner)

Example :

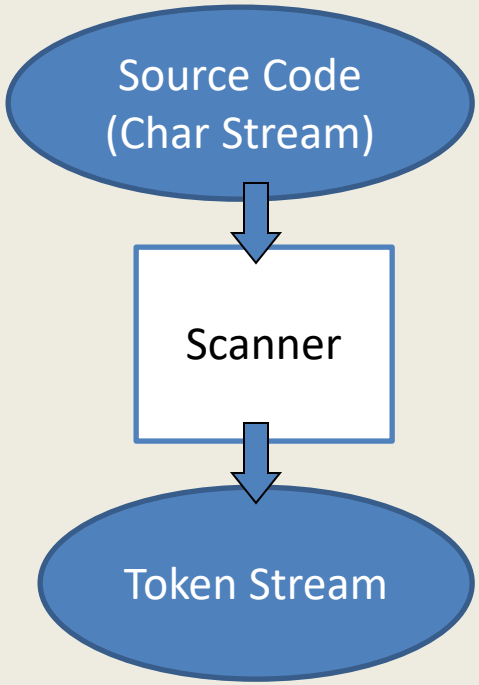
sum	=	sum	+	100	;



Lexical Analyzer (Scanner)

Example :

sum	=	sum	+	100	;



Categories of Tokens :

Keywords – while, if, …

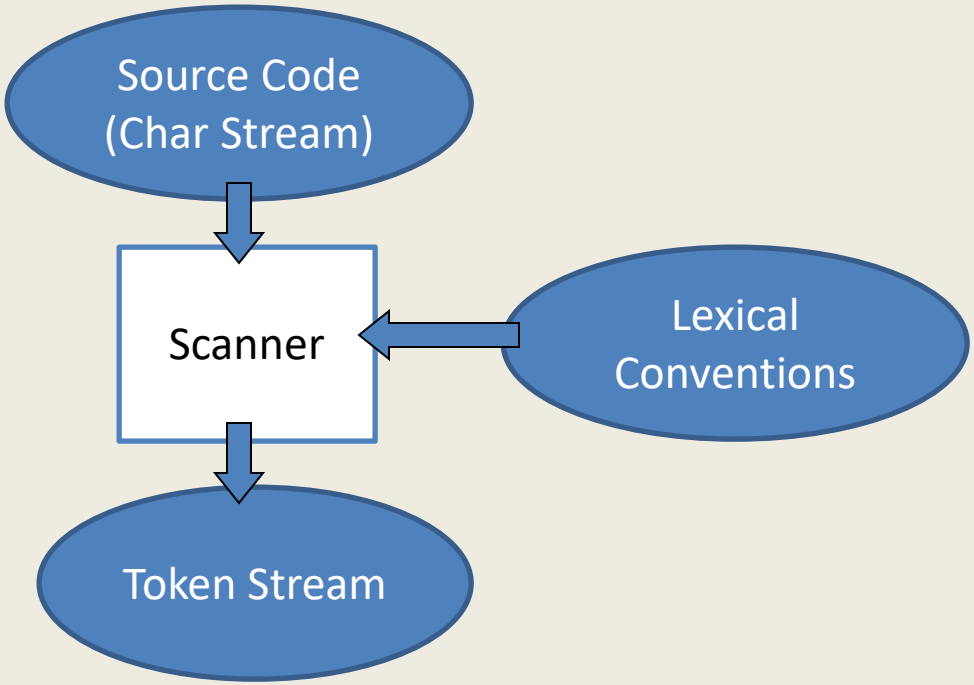
Identifiers – i, j, sum, …

Constants – 100, 12.5, “India”, …

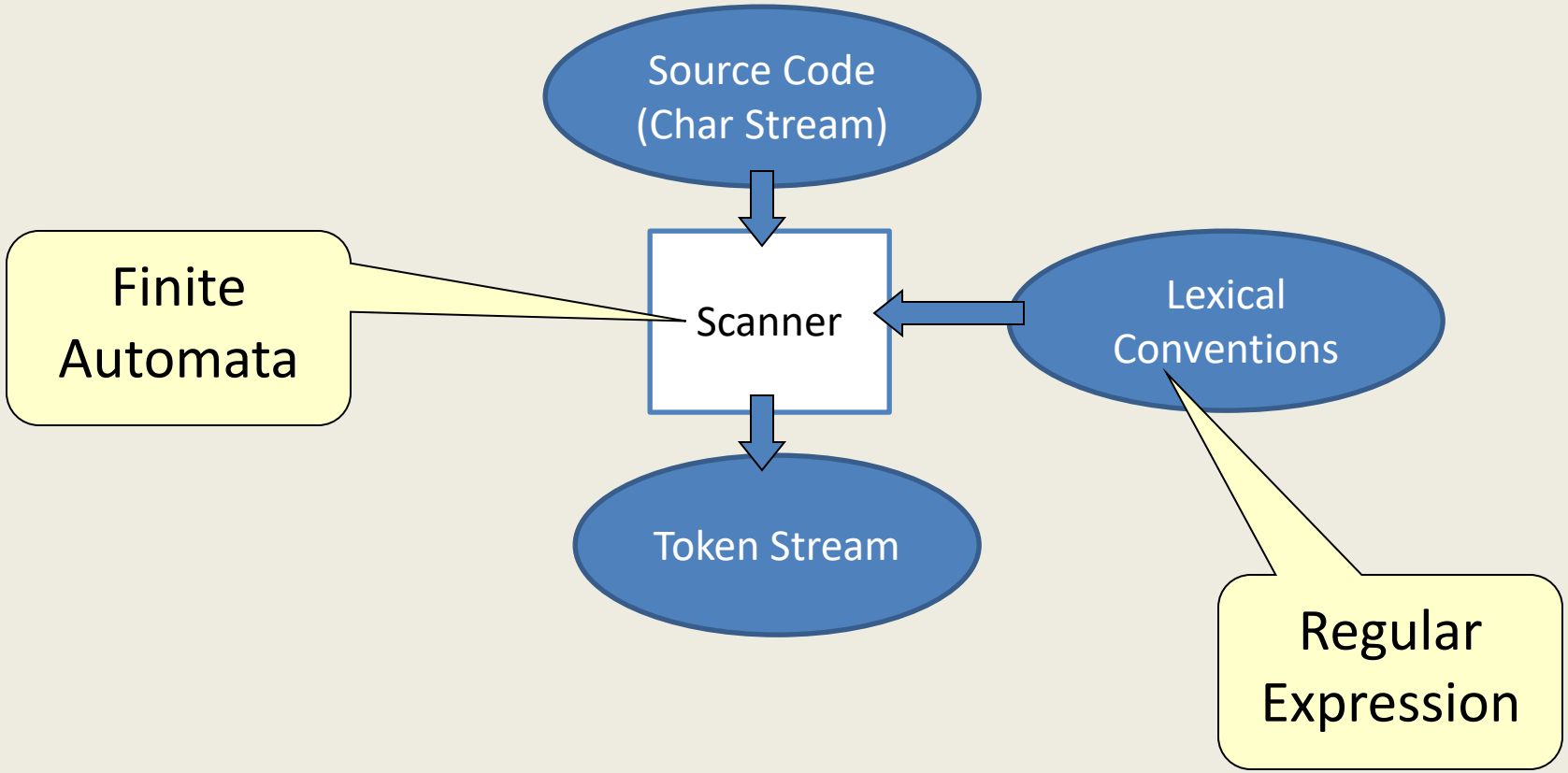
Operators - +, <, <=, …

Punctuation Symbols – (,), ;, …

How a Scanner works?

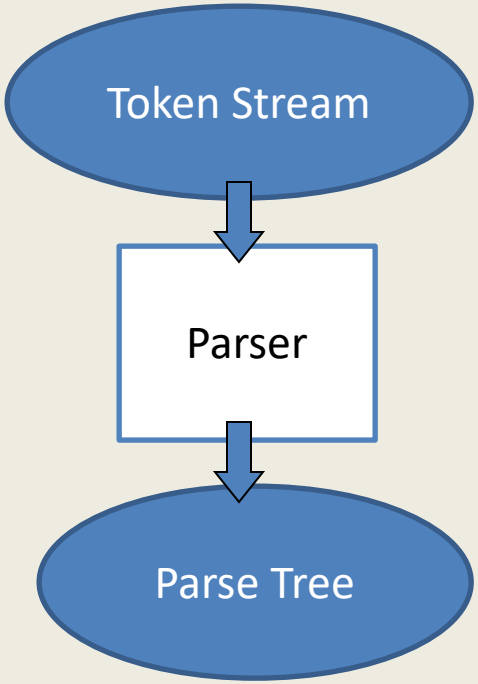


How a Scanner works?



Syntax Analyzer (Parser)

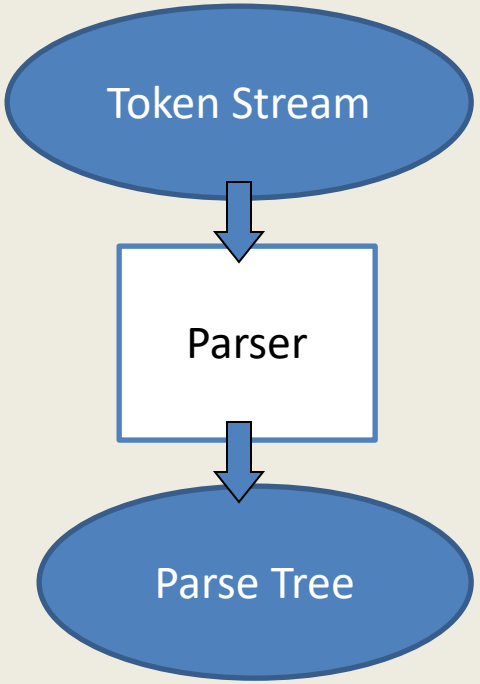
Forms the group of tokens that logically belong together – Syntactic Structures



Syntax Analyzer (Parser)

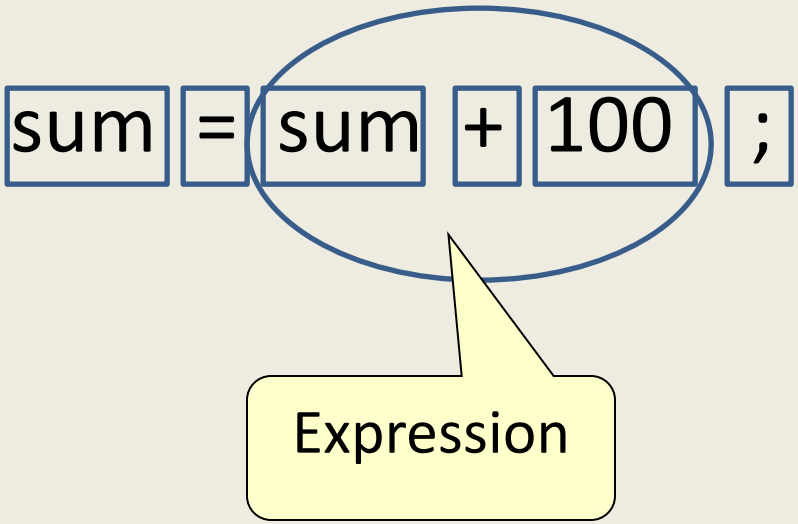
Example :

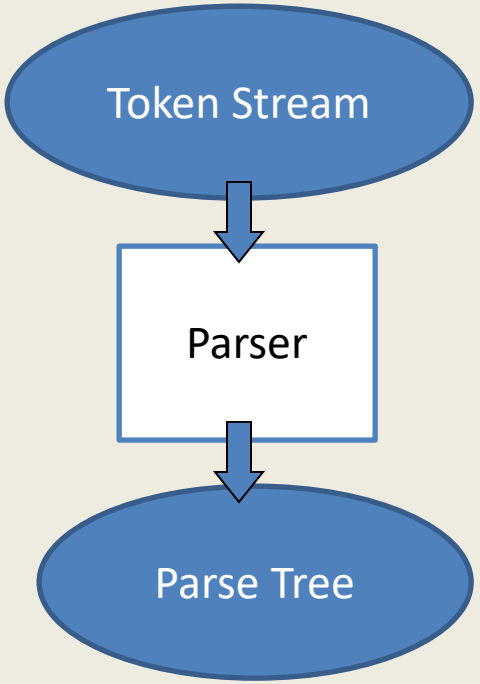
sum	=	sum	+	100	;



Syntax Analyzer (Parser)

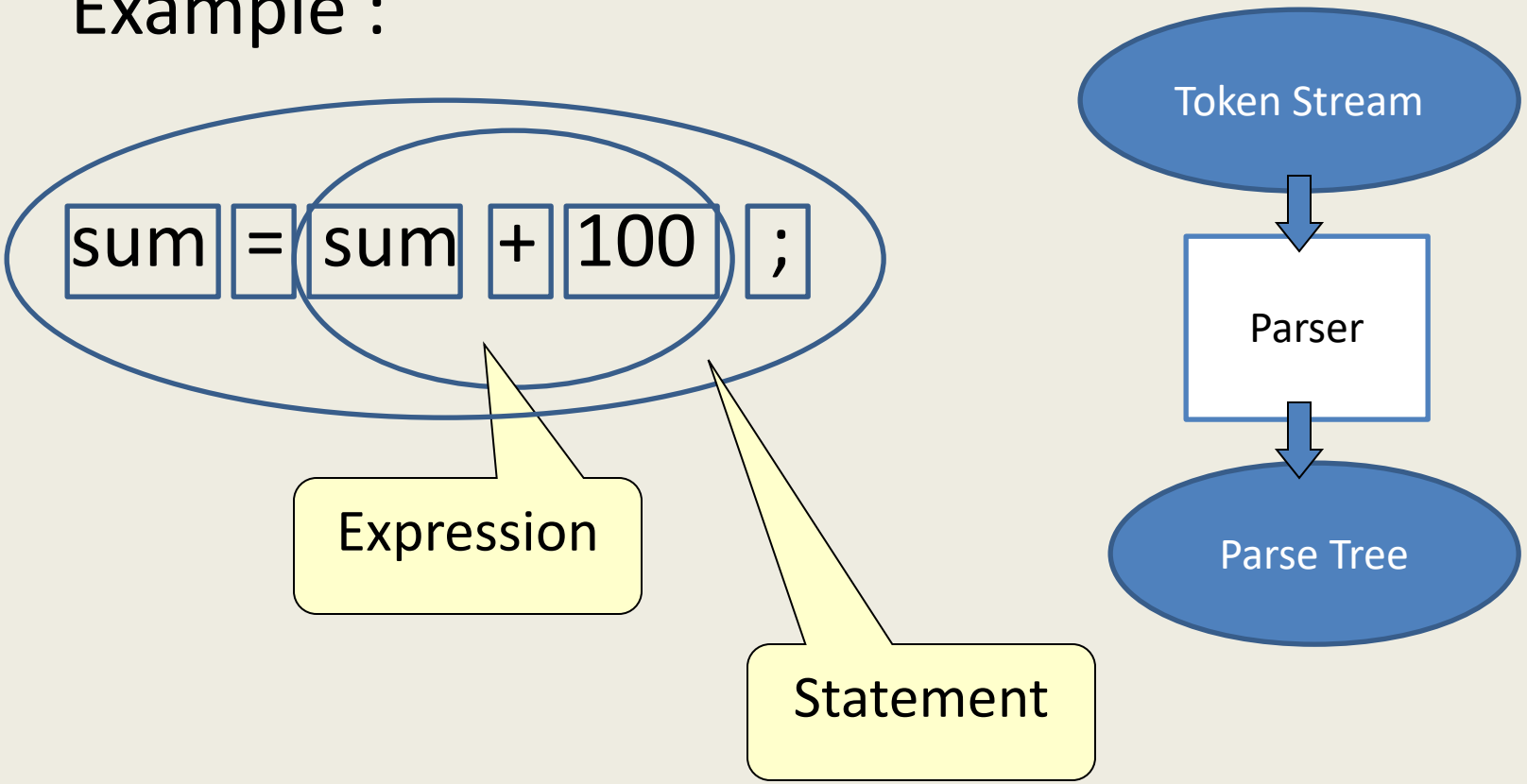
Example :



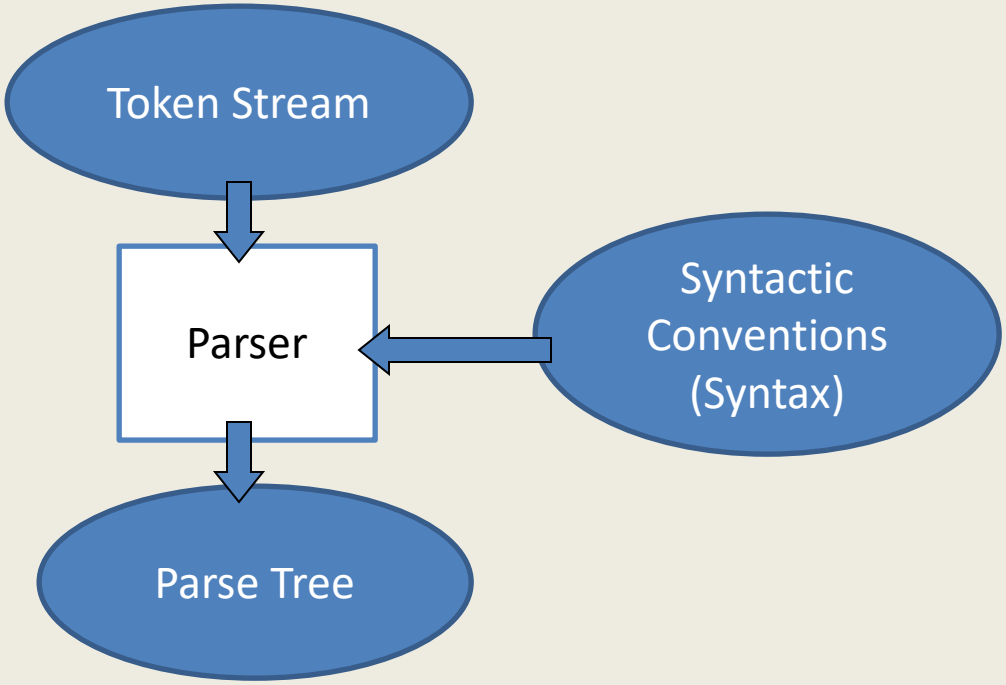


Syntax Analyzer (Parser)

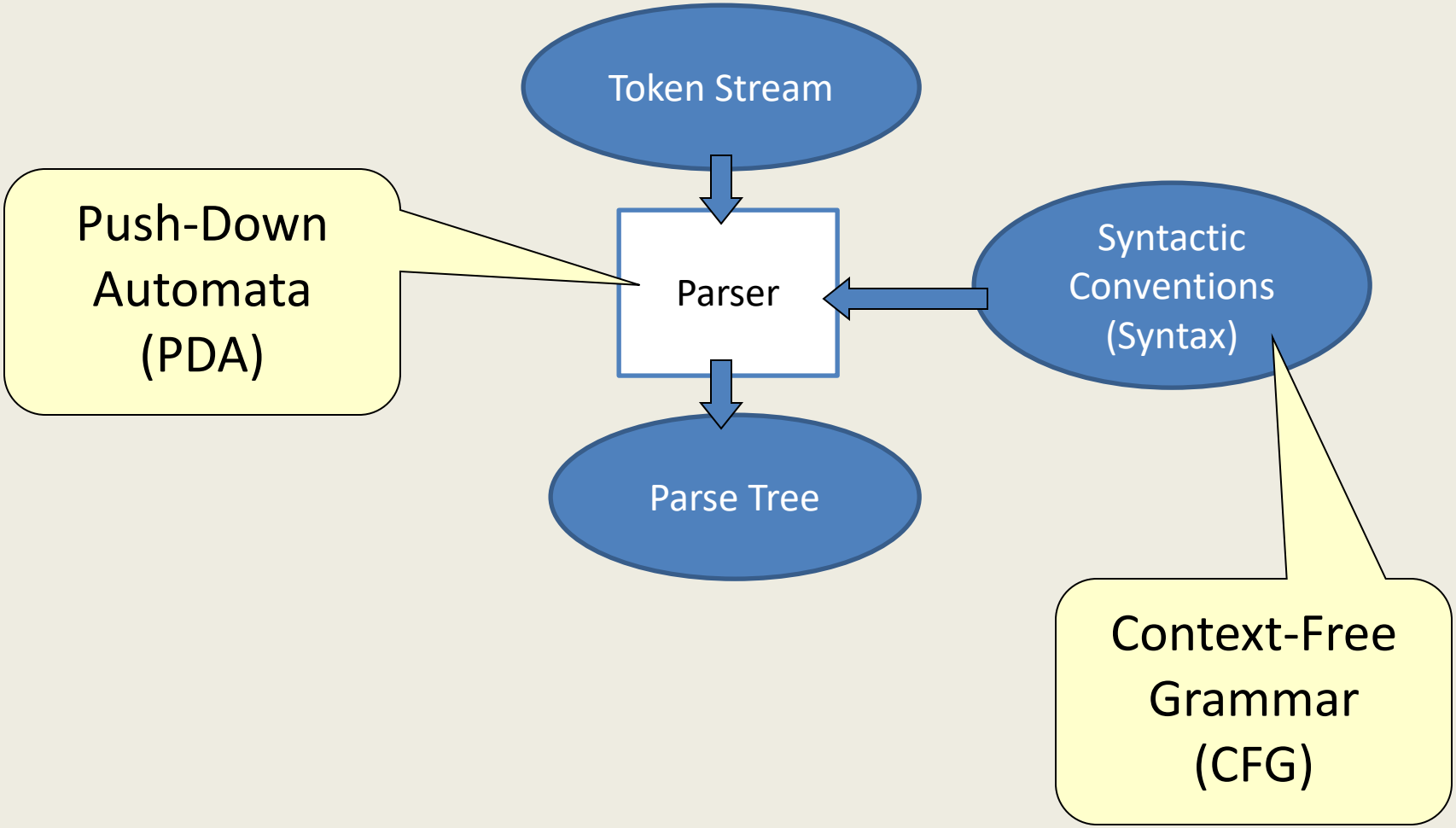
Example :



How a Parser works?

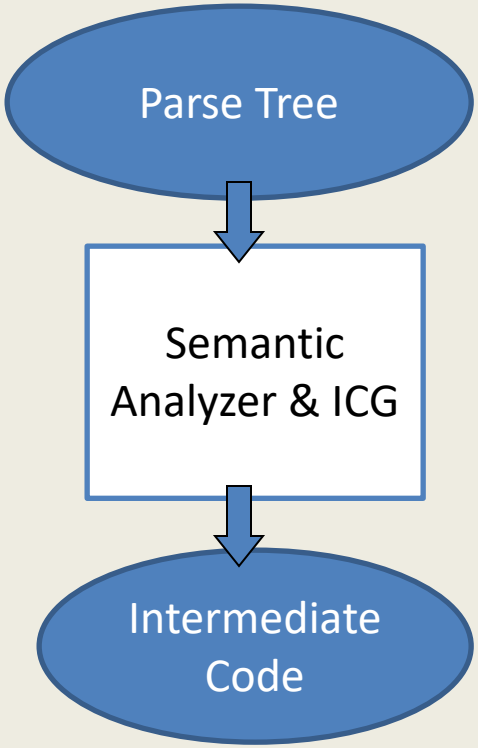


How a Parser works?



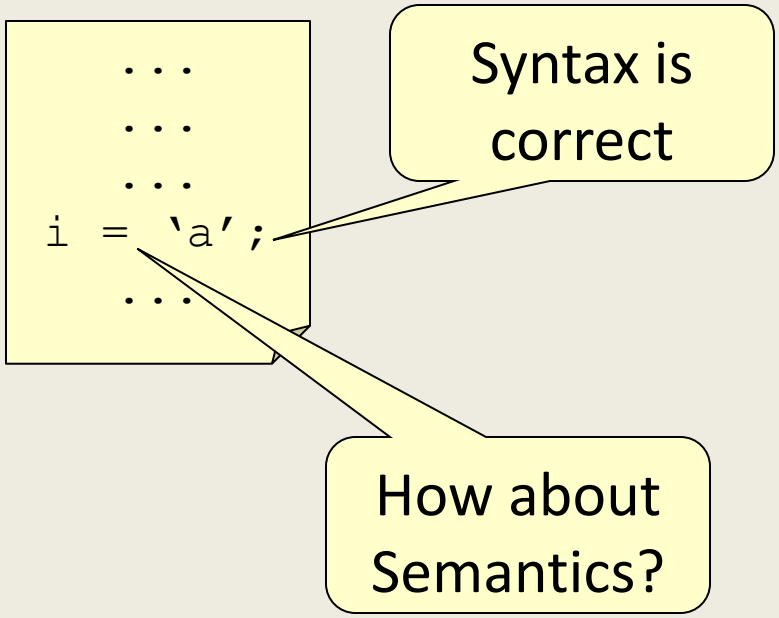
Semantic Analyzer & Intermediate Code Generation

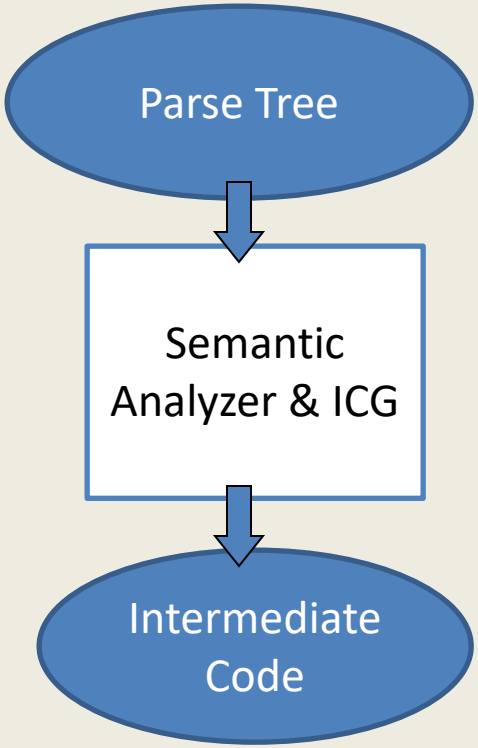
Checks whether the code is semantically correct and then transforms into Intermediate Code



Semantic Analyzer & Intermediate Code Generation

Example :



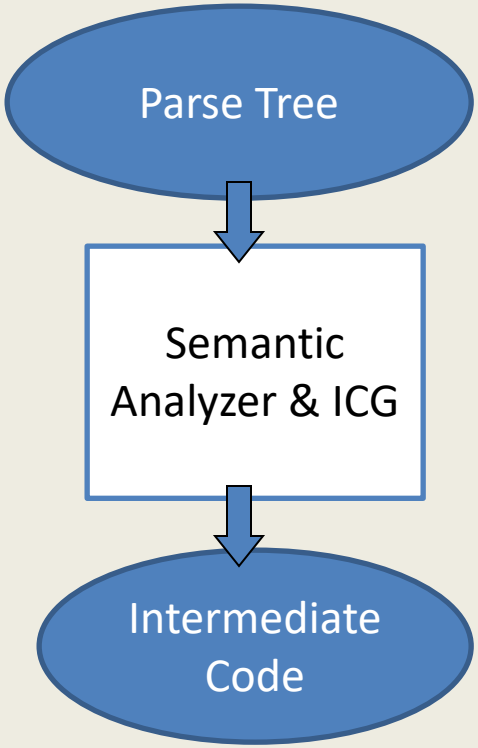


Semantic Analyzer & Intermediate Code Generation

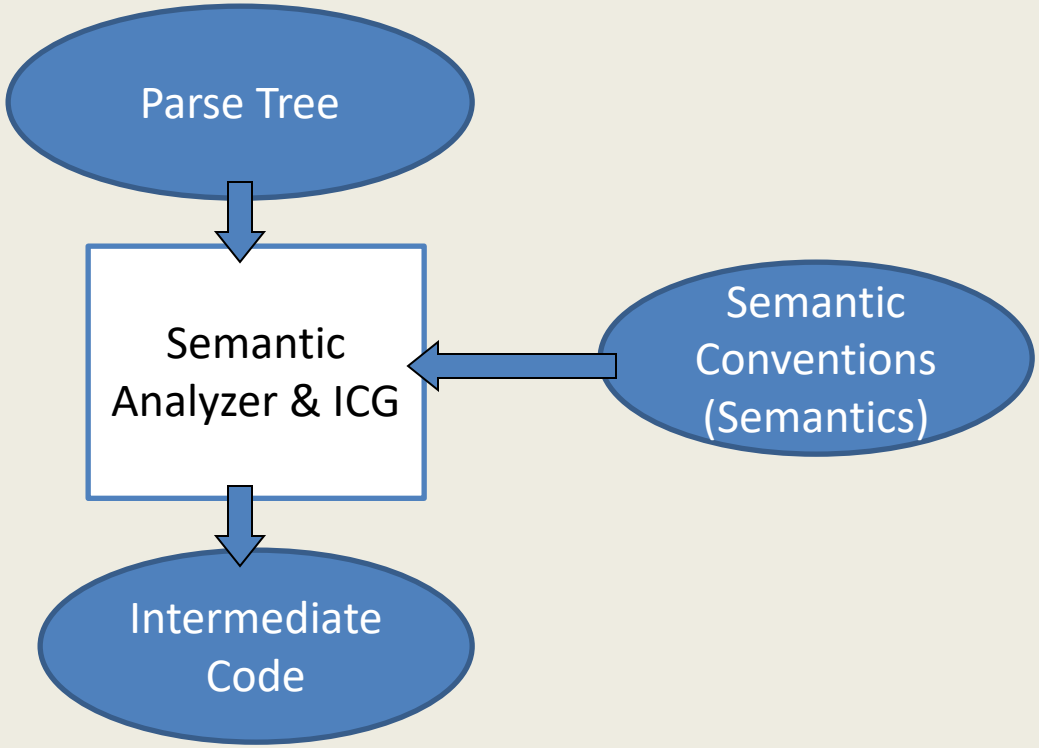
Example :

```
int i;
...
...
i = 'a';
...
```

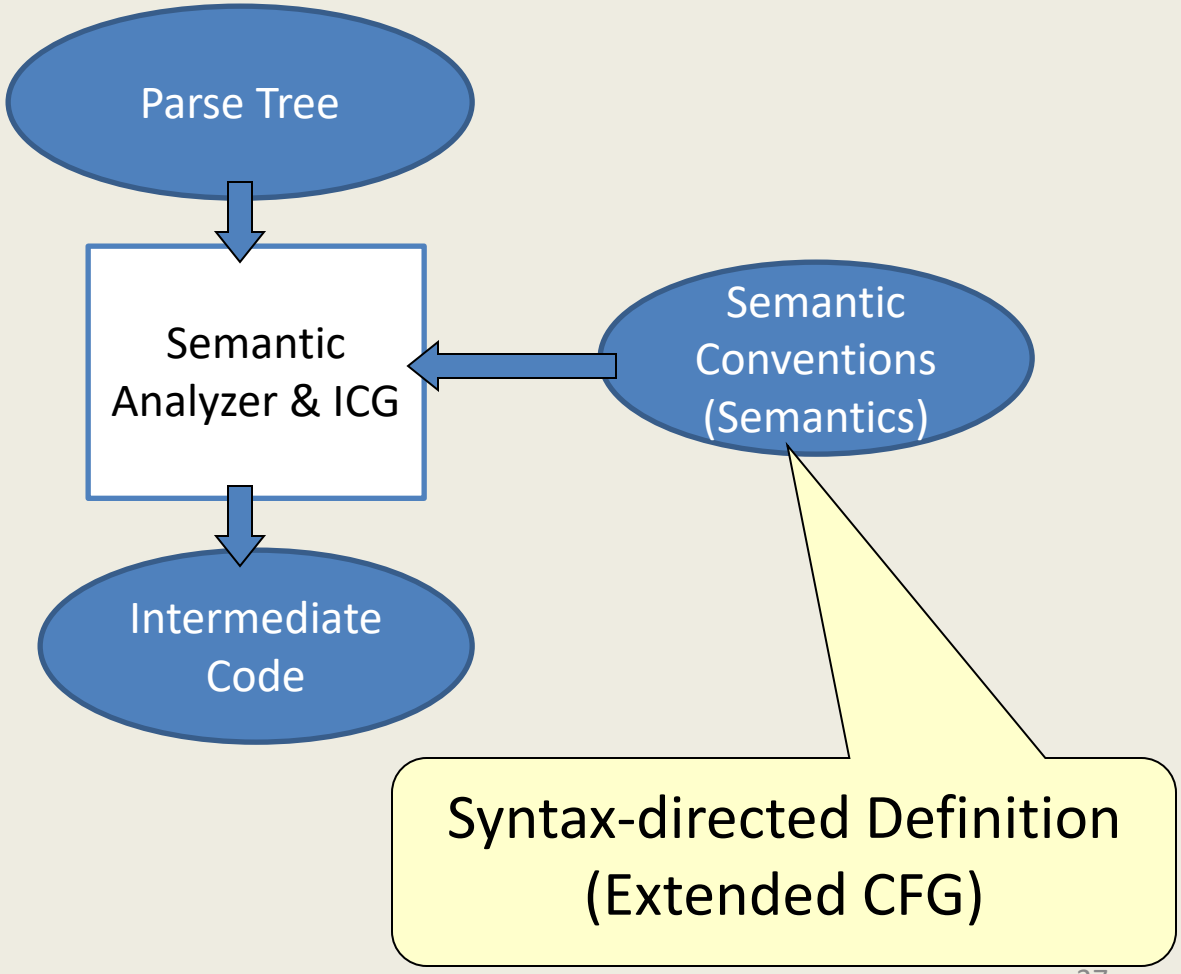
Context will decide



How Semantic Analyzer & ICG works?

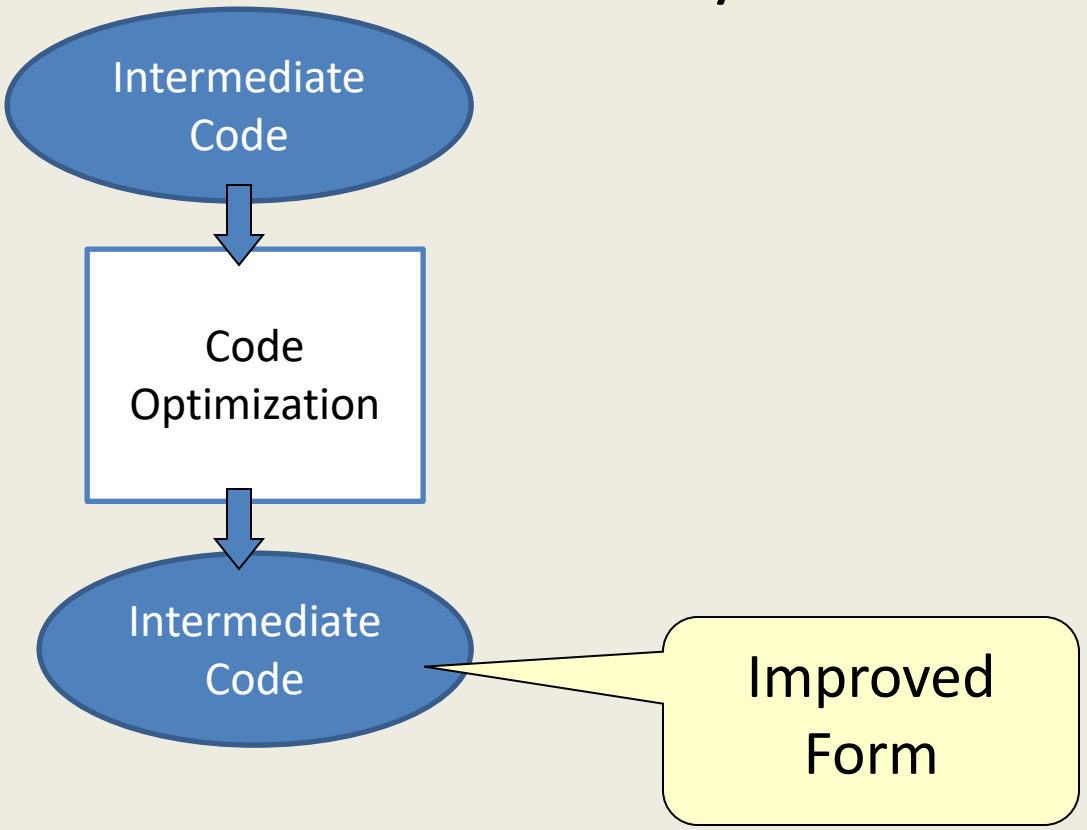


How Semantic Analyzer & ICG works?



Code Optimization

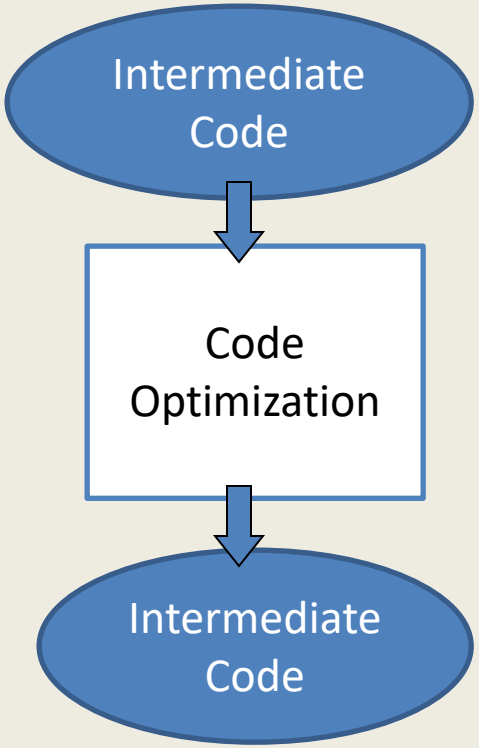
To improve the Intermediate Code so that the final Machine Code runs faster and/or takes less space



Code Optimization

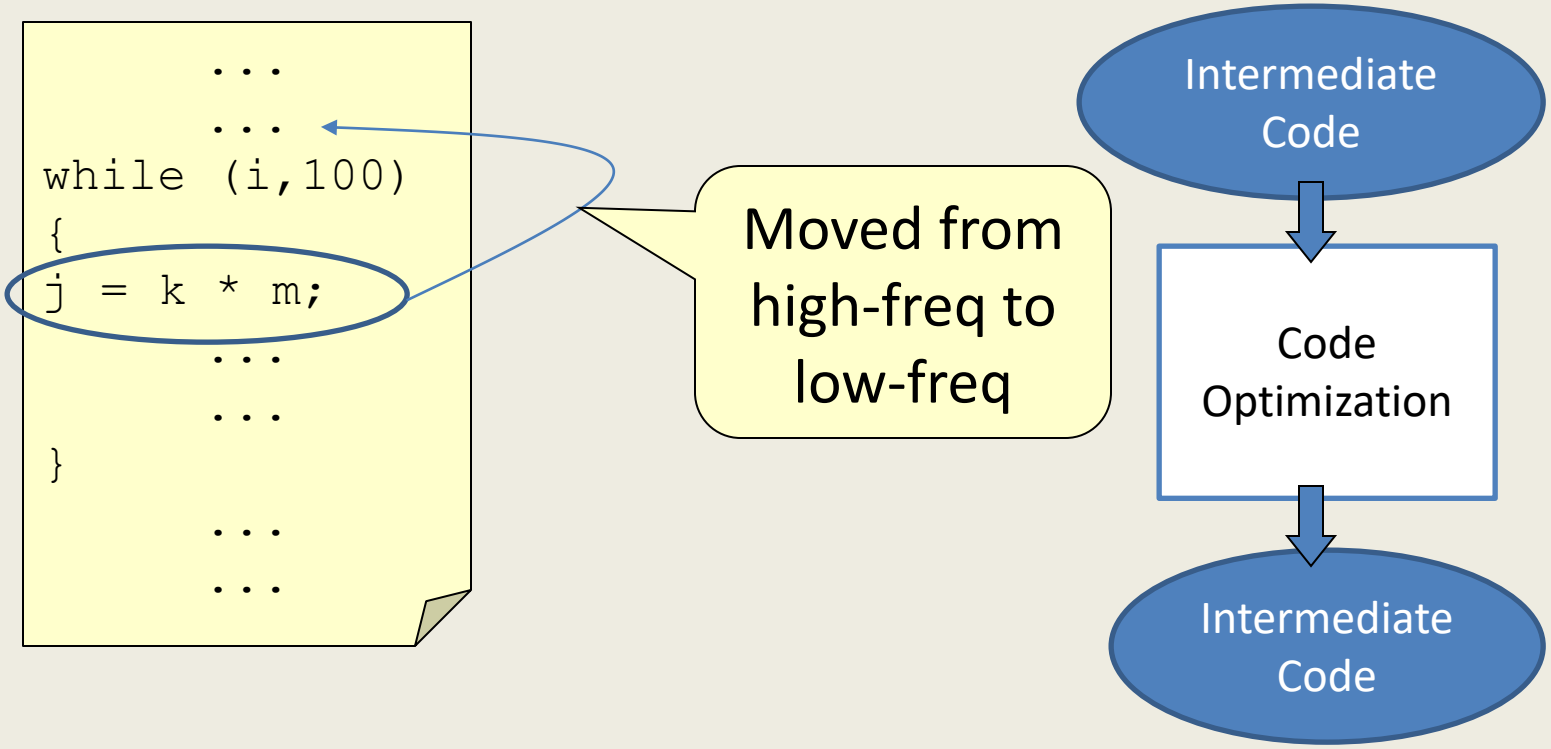
Example :

```
        ...
        ...
while (i<100)
{
j = k * m;
        ...
        ...
}
        ...
        ...
```

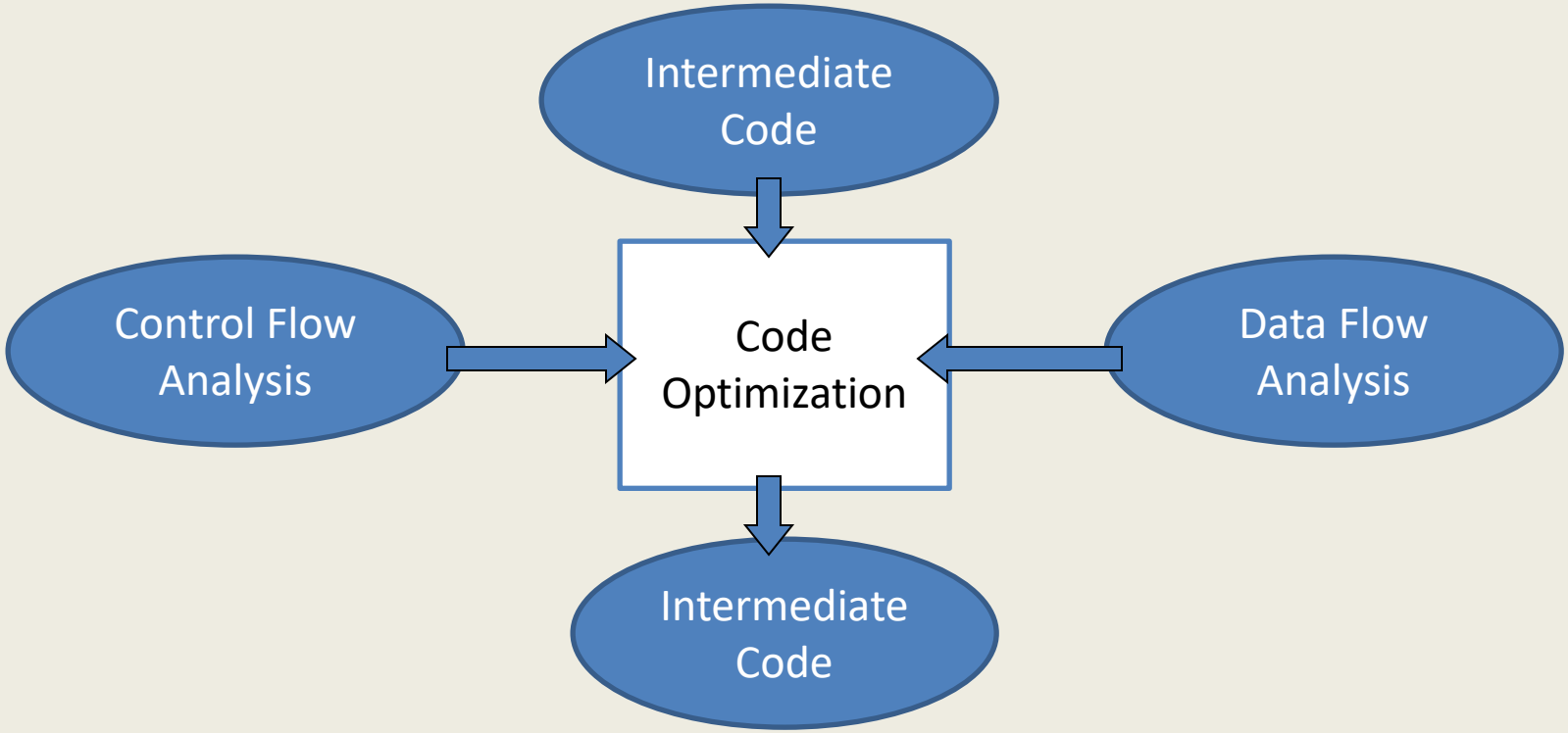


Code Optimization

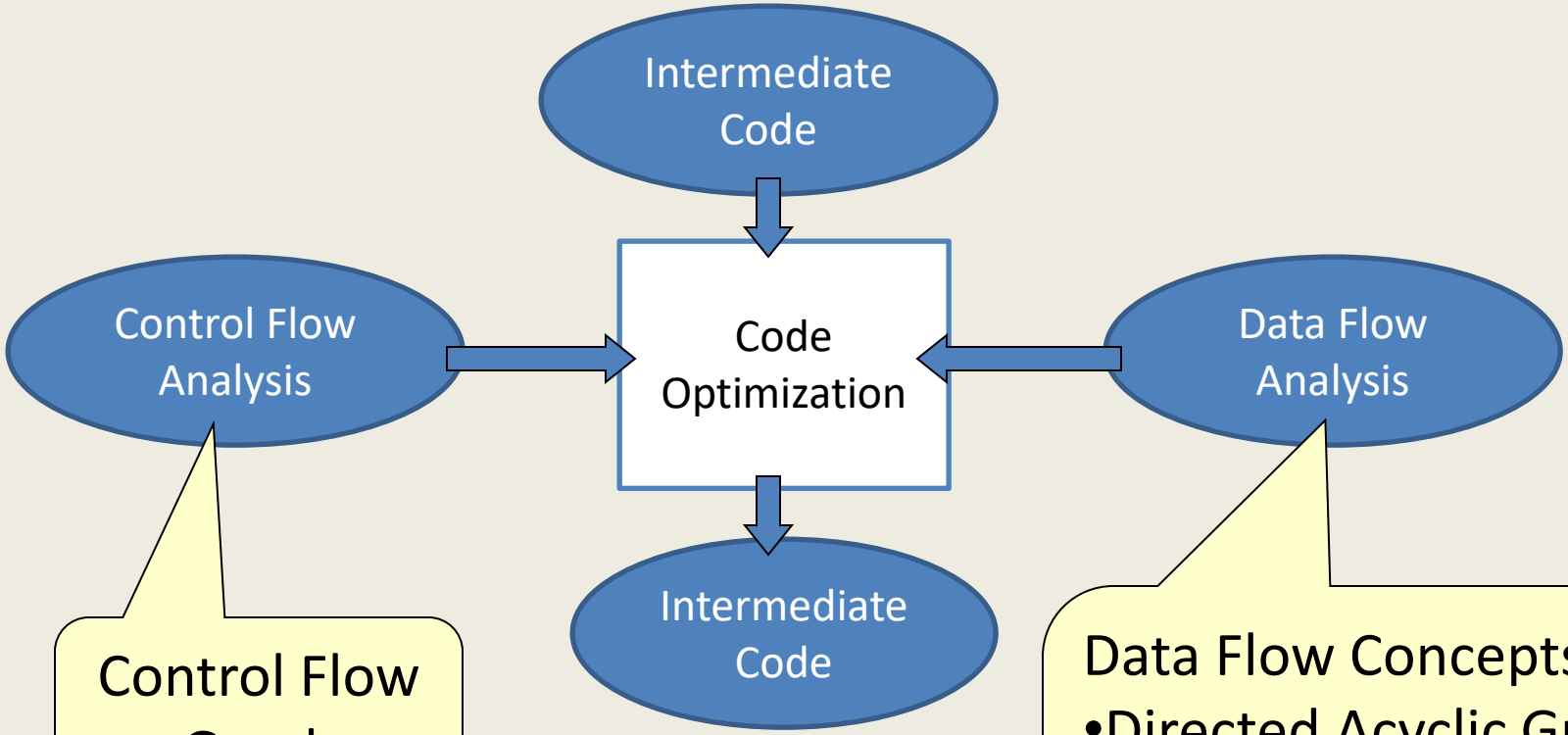
Example :



How Code Optimization works?

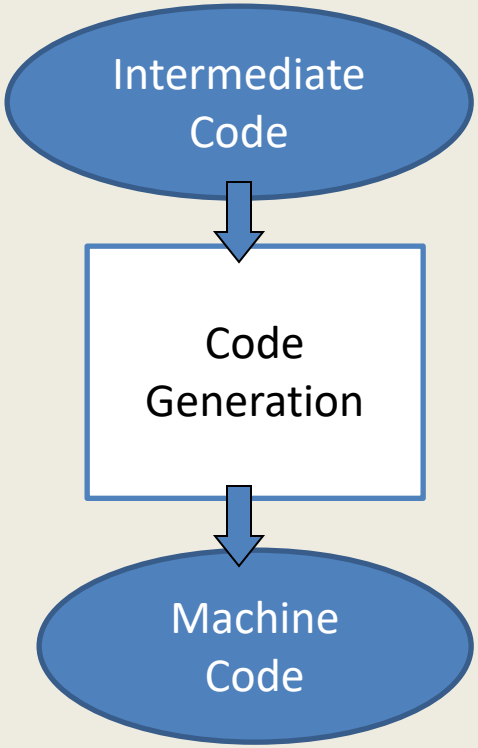


How Code Optimization works?



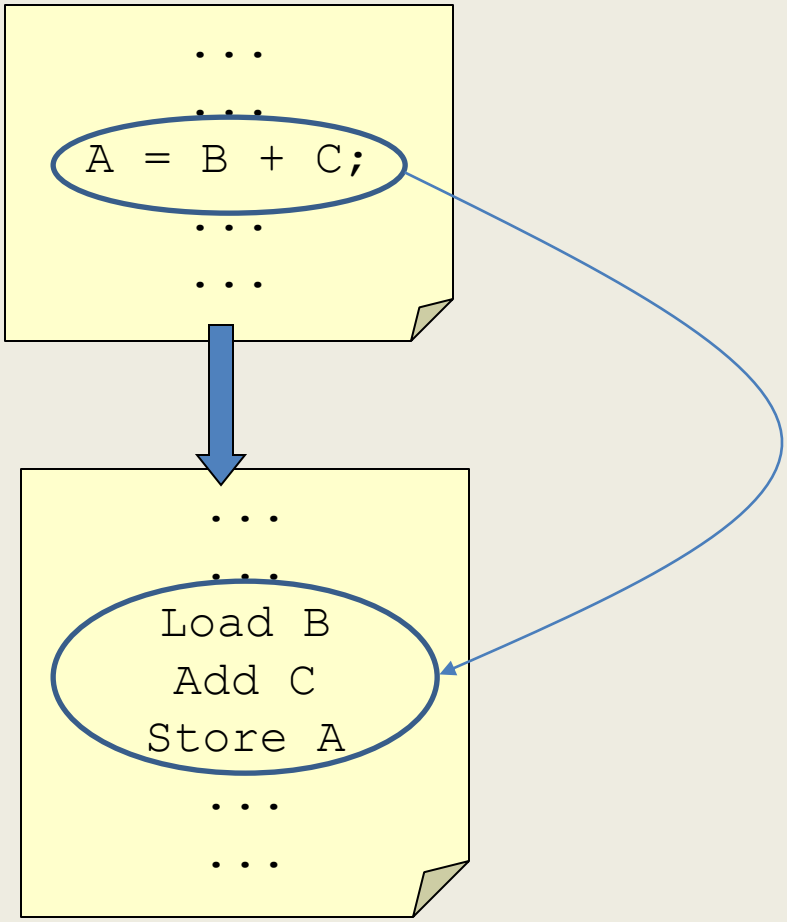
Code Generation

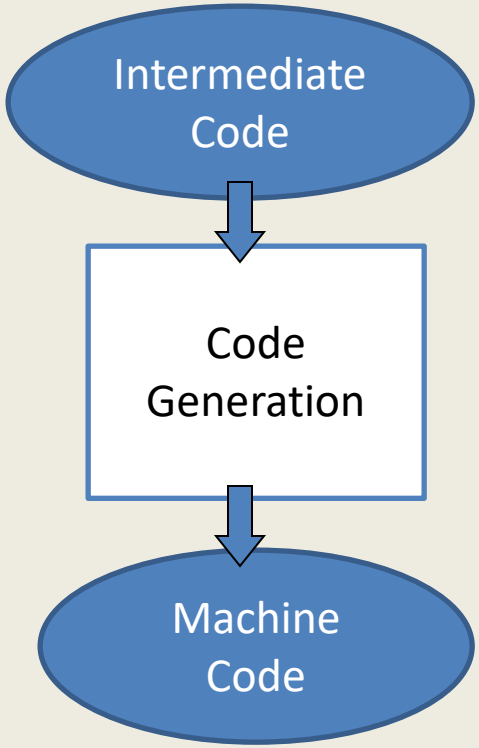
Converts the Intermediate Code into a sequence of Machine Instructions



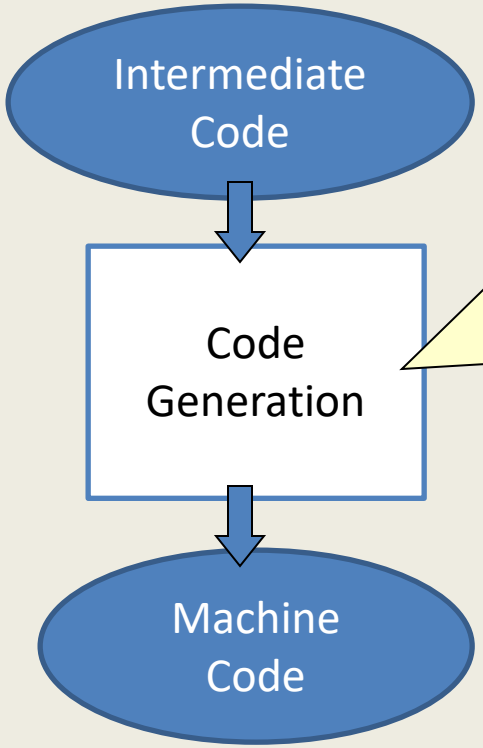
Code Generation

Example :





How Code Generation works?



Machine specific Decisions:

- Selection of Registers
- Memory Locations
- Address Resolution

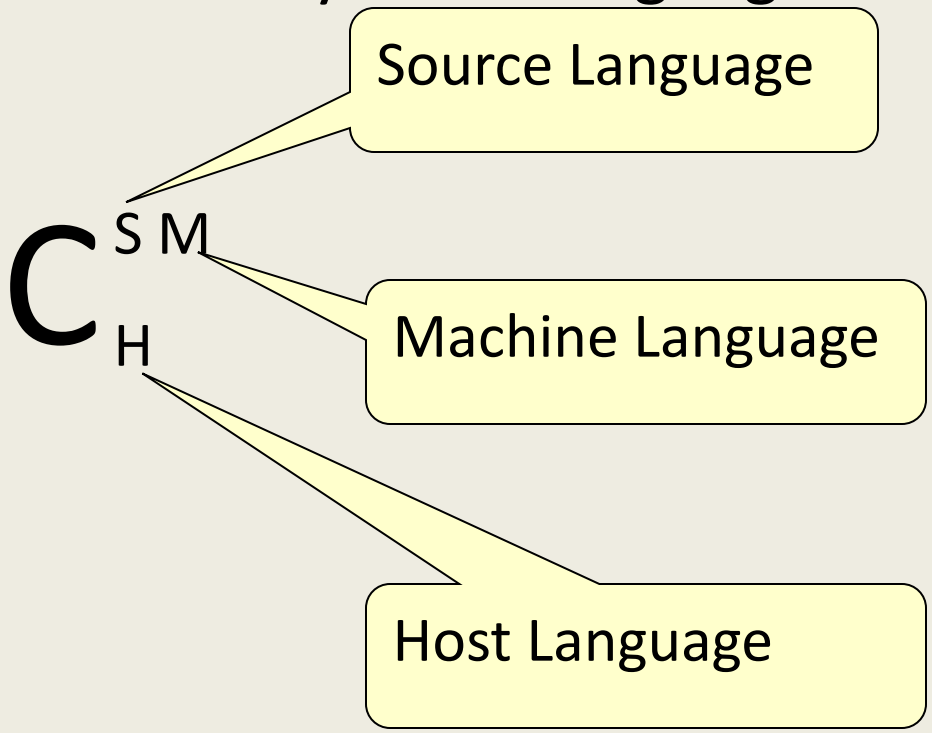
...

Self Compiler - Bootstrapping

A typical Compiler is characterized by three languages

Self Compiler - Bootstrapping

A typical Compiler is characterized by three languages



Self Compiler - Bootstrapping

Let there is a new language L which is to be made available for a machine A

$$C^{L\,A}_{A}$$

Self Compiler - Bootstrapping

Let there is a new language L which is to be made available for a machine A

$$C_{A}^{L\,A}$$

We may start with a smaller and simpler language S ($S \subseteq L$)

$$C_{A}^{S\,A}$$

Self Compiler - Bootstrapping

Let there is a new language L which is to be made available for a machine A

$$C^{L\,A}_{A}$$

Now we write another compiler for language L in S

$$C^{L\,A}_{S}$$

$$C^{S\,A}_{A}$$

Self Compiler - Bootstrapping

Let there is a new language L which is to be made available for a machine A

Compiler for full language L written in S

$C^{L\,A}_{S}$

$C^{L\,A}_{A}$

$C^{S\,A}_{A}$

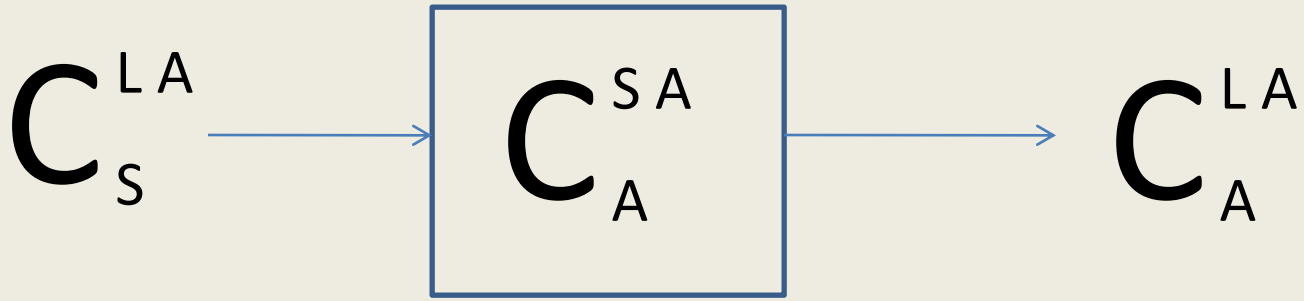
Compiler for smaller language S written in A

Self Compiler - Bootstrapping

Let there is a new language L which is to be made available for a machine A

$$C^{L\,A}_{A}$$

We may start with a smaller and simpler language S ($S \subseteq L$)



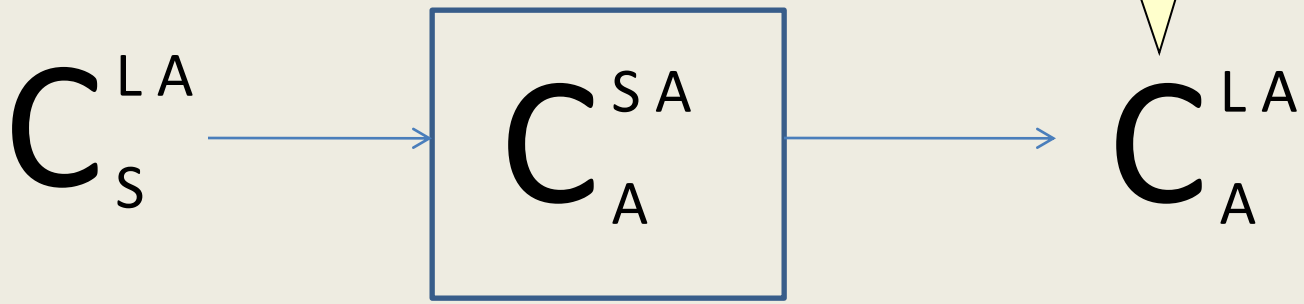
Self Compiler - Bootstrapping

Let there is a new language L which is to be made available for a machine A

$$C^{L\,A}_{A}$$

Compiler for full language L and can run on A

We may start with a smaller and simpler language S ($S \subseteq L$)



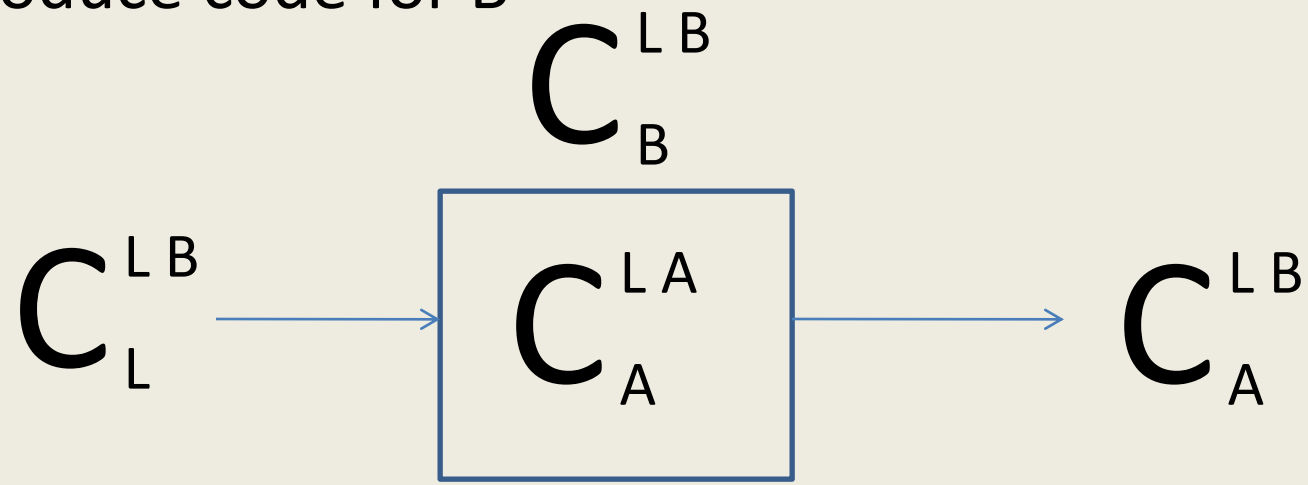
Cross Compiler

Now we want another compiler for L to run on machine B and produce code for B

$$C^{L\,B}_{B}$$

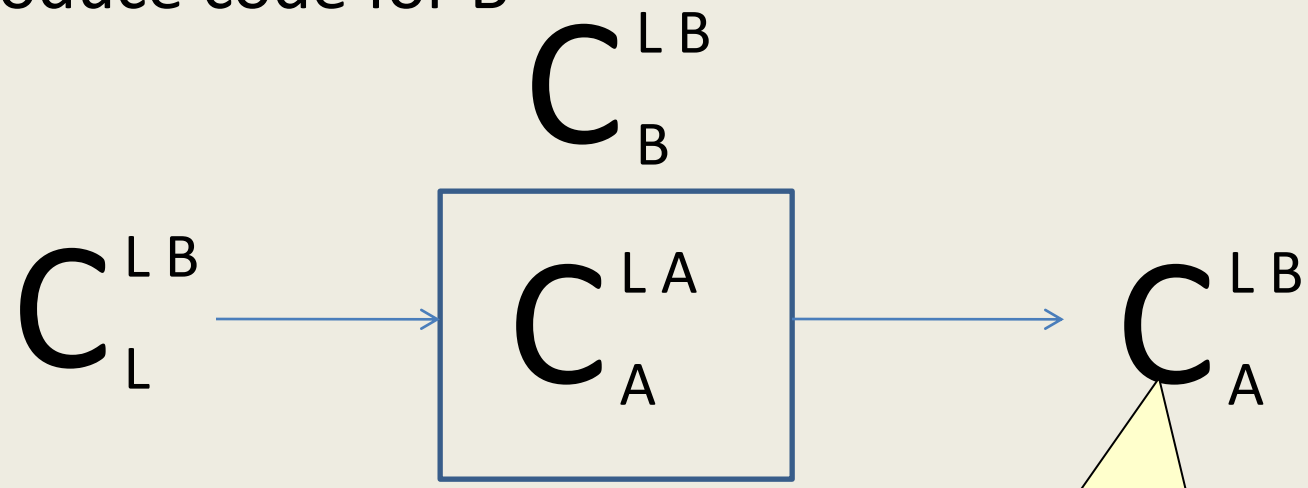
Cross Compiler

Now we want another compiler for L to run on machine B and produce code for B



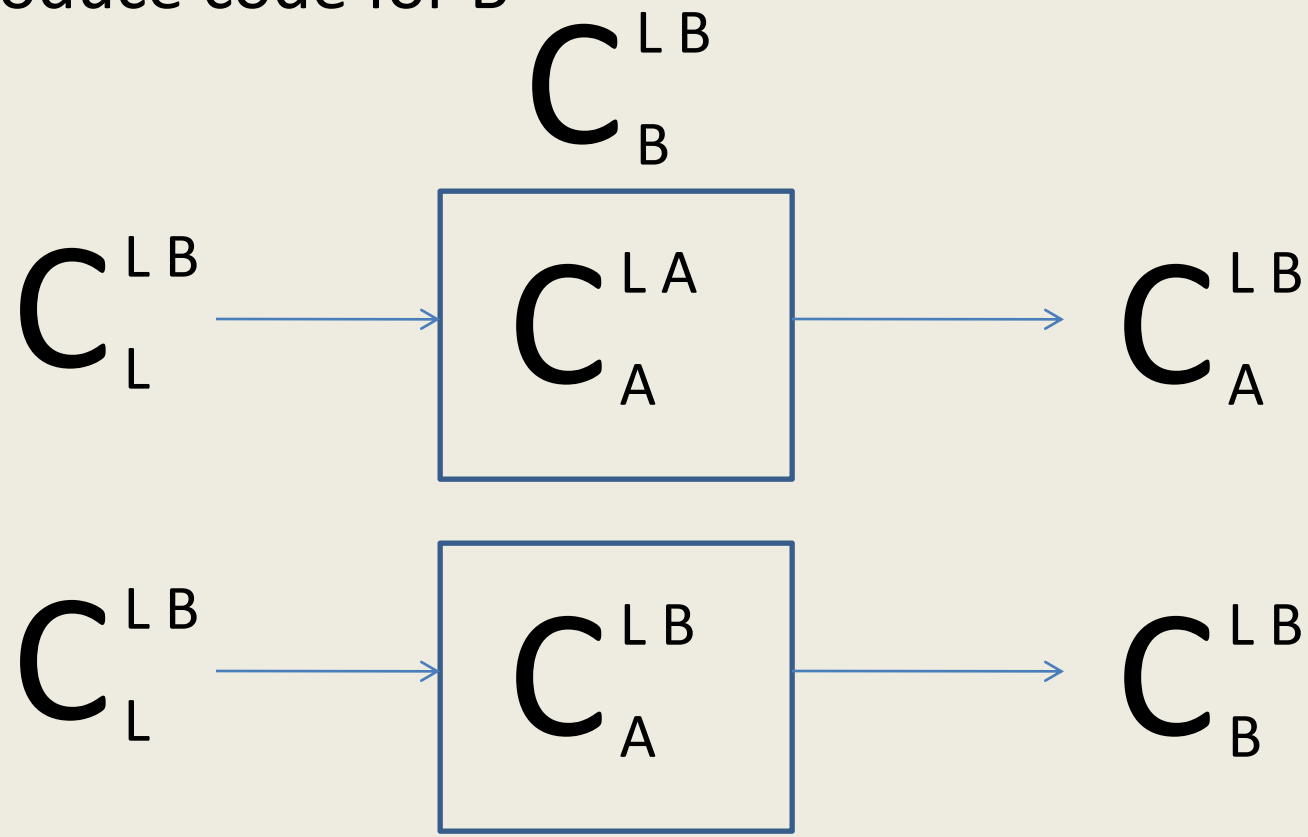
Cross Compiler

Now we want another compiler for L to run on machine B and produce code for B



Cross Compiler

Now we want another compiler for L to run on machine B and produce code for B



Cross Compiler

Now we want another compiler for L to run on machine B and produce code for B