Introduction to Compilers

Lecture 1
Chapter 1

Rokan Uddin Faruqui

Associate Professor
Dept of Computer Science and Engineering
University of Chittaong, Bangladesh
Email: *rokan@cu.ac.bd*

Outline

Outline

Compiler

high level —> low level/ object code

What is a compiler, what does it do?

Compiler

source program $\rightarrow$ **COMPILER** $\rightarrow$ **target program**

- semantically equivalent programs,
- **source programs**: typically high level,
- **target programs**: typically assembler or object/machine code.

object code $\neq$ **object – oriented**
byte code

Roles of a Compiler

- allow **programming at an understandable abstraction level** but **execution based on low-level code**;
- allow programs to be written in **machine-independent languages** but **execution based on machine-specific code**;
- help in **verifying software**
- early **discovery of programming errors**
- provide **automatic code optimization**.

Some Historical Highlights

- **Grace Hopper** coins the concept and writes the first compiler in 1952.
- **John W. Backus** presents the first formally articulated (**FORTRAN**) compiler in 1957.
- **Frances E. Allen, John Cocke** introduce most of the abstract concepts used in compiler optimization and parallel compilers today up through the early 1960s.
- **Alfred V. Aho and Jeffrey D. Ullman (and others)** formalize parsing in 1960s and 70s (finite automata, context-free grammars).
- **Donald Knuth** publishes attribute grammars in 1968, which defines modern compiler construction methodology

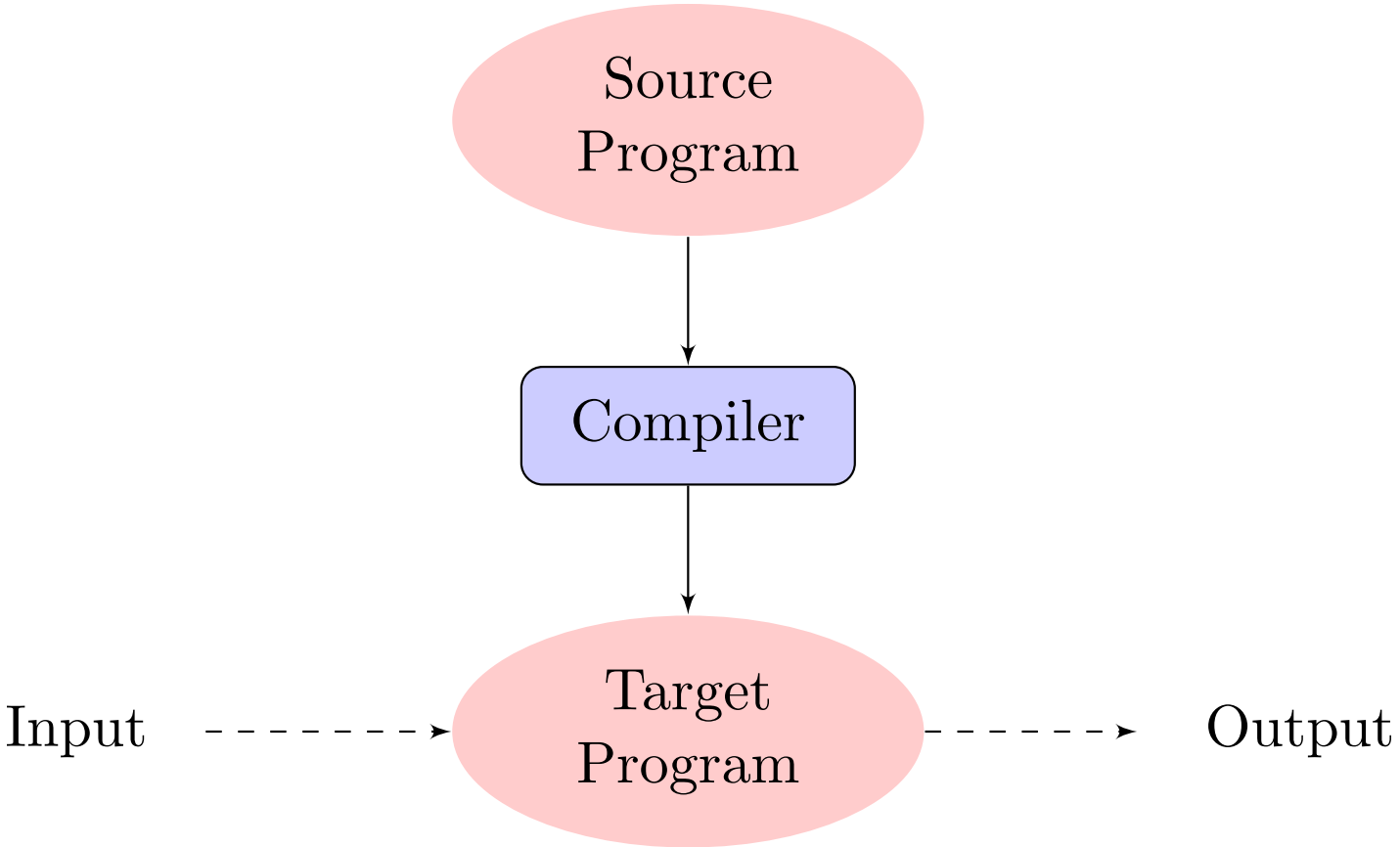
Compilation and Interpretation

- Pure compilation
- Pure interpretation
- Mixing compilation and interpretation
- Preprocessing
- Library routines and linking
- Post-compilation assembly
- The C preprocessor
- Source-to-source translation
- Dynamic and just-in-time compilation

Compilation and Interpretation

Pure Compilation

translates - a source program to a target program



Compilation and Interpretation

Pure Compilation

translates - a source program to a target program

- compiler is the locus of control during compilation
- target program is the locus of control during its own execution
- compiler is itself a machine language program
- machine language is commonly known as object code

Compilation and Interpretation

Pure Interpretation

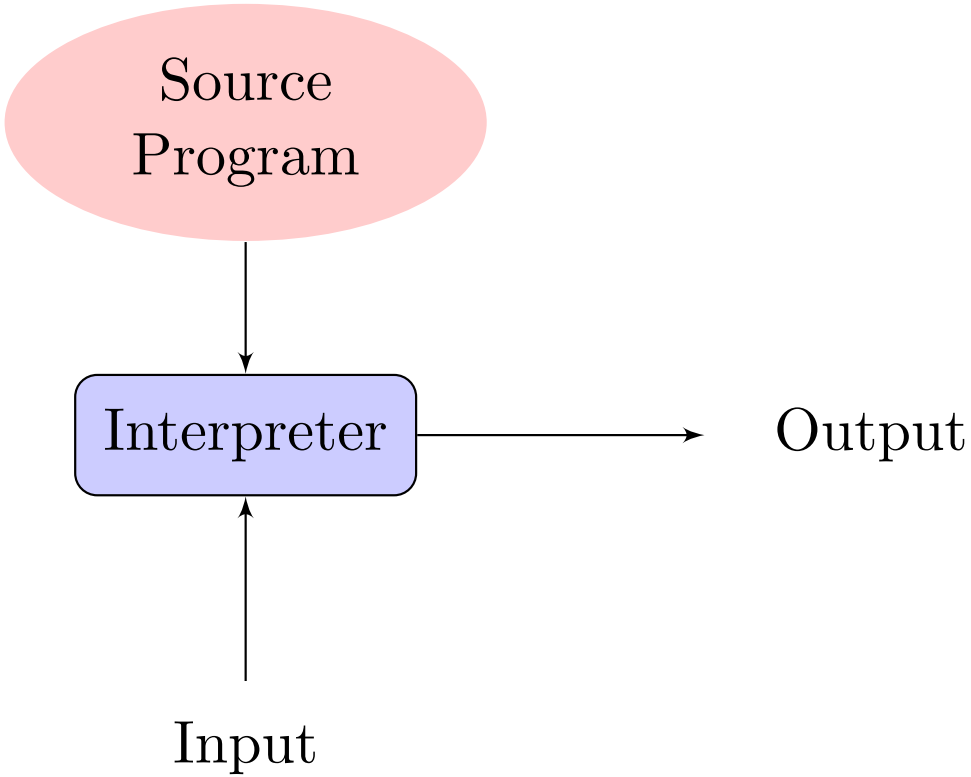


Figure: Pure Interpretation

Compilation and Interpretation

Pure Interpretation

- *late binding* - delaying decisions about program implementation until run time
- source-level debugger

Compilation and Interpretation

Mixing compilation and interpretation

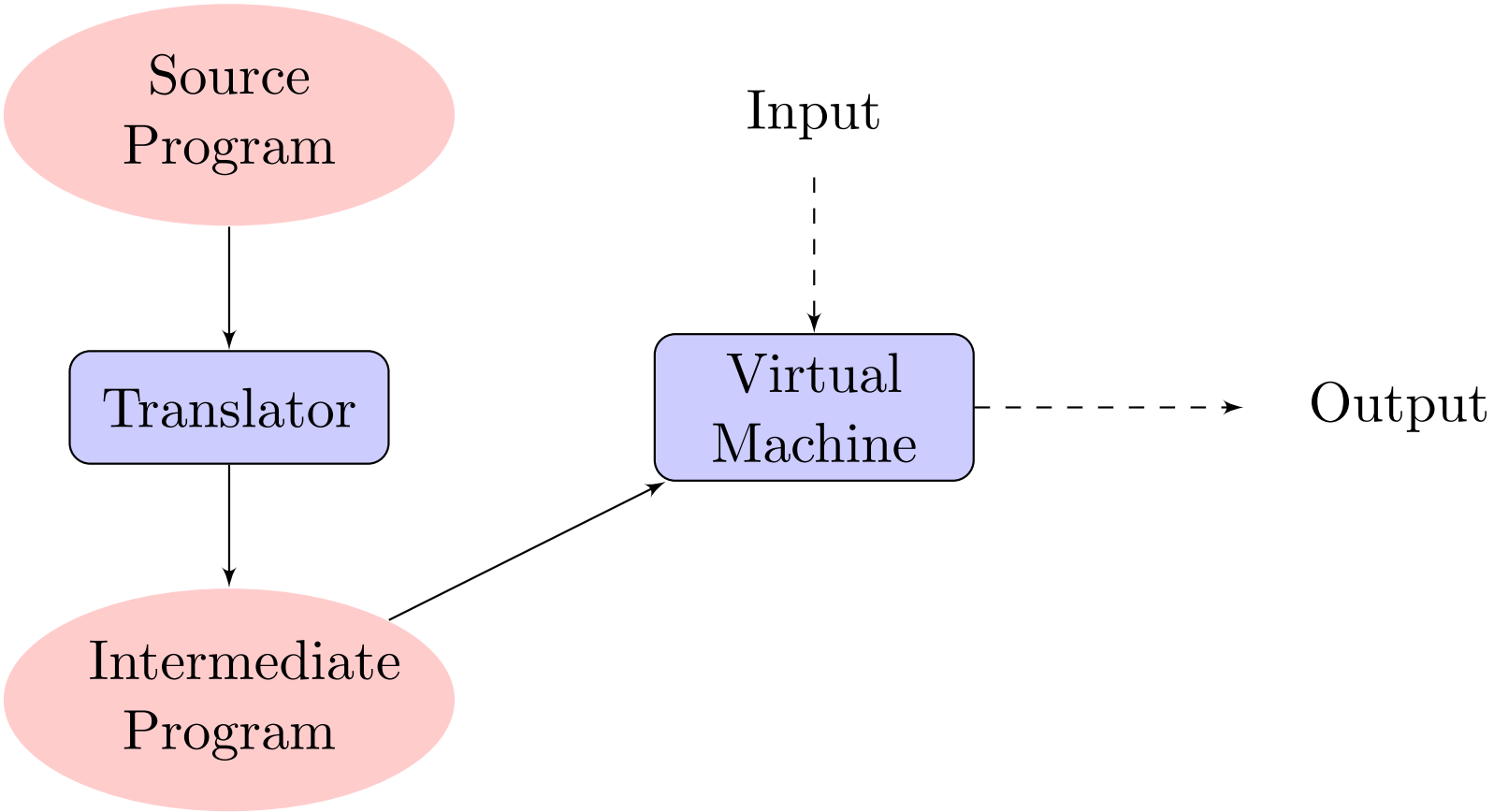


Figure: Mixing compilation and interpretation

Compilation and Interpretation

Preprocessing

- removes comments and white space
- groups characters together into tokens such as keywords, identifiers, numbers, and symbols
- produce an intermediate form that can be interpreted more efficiently

Compilation and Interpretation

Library routines and linking

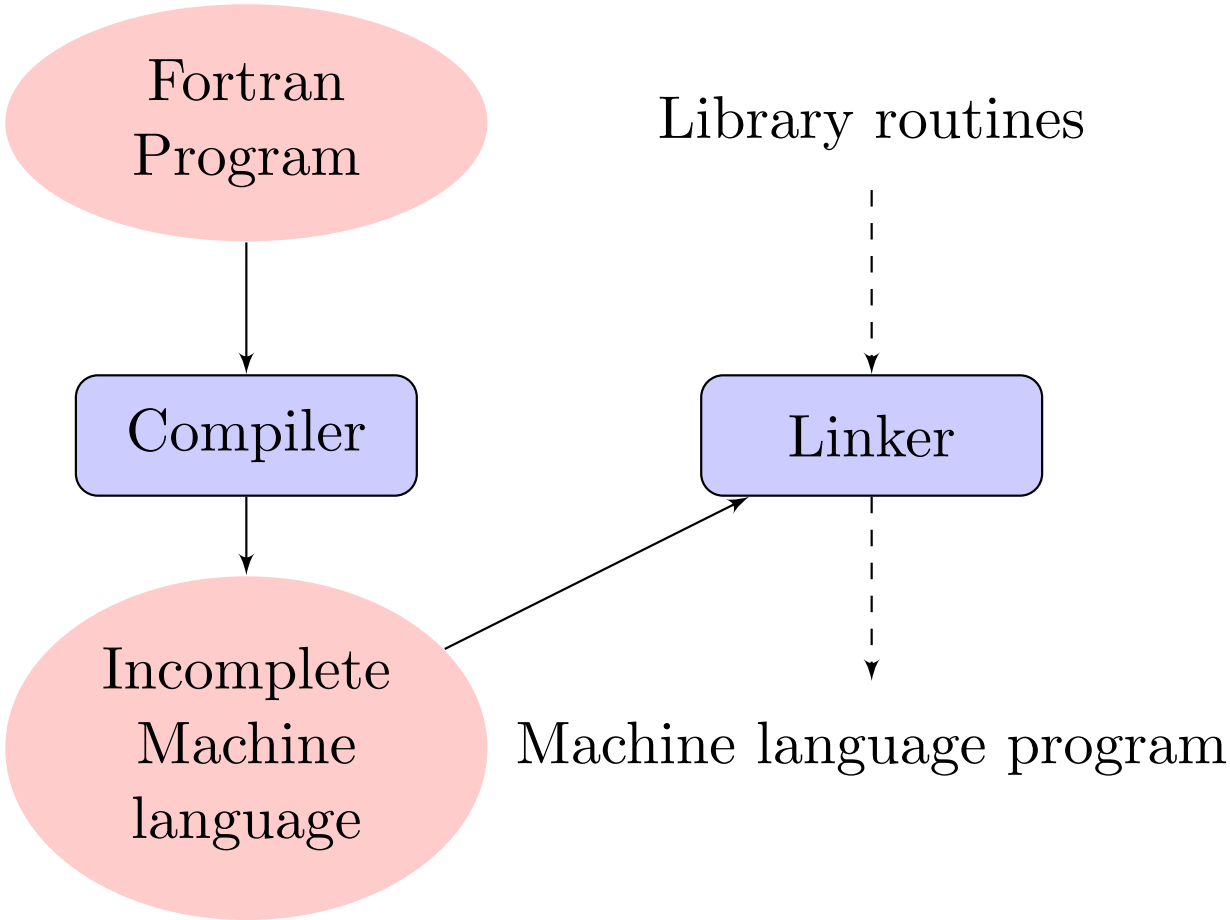
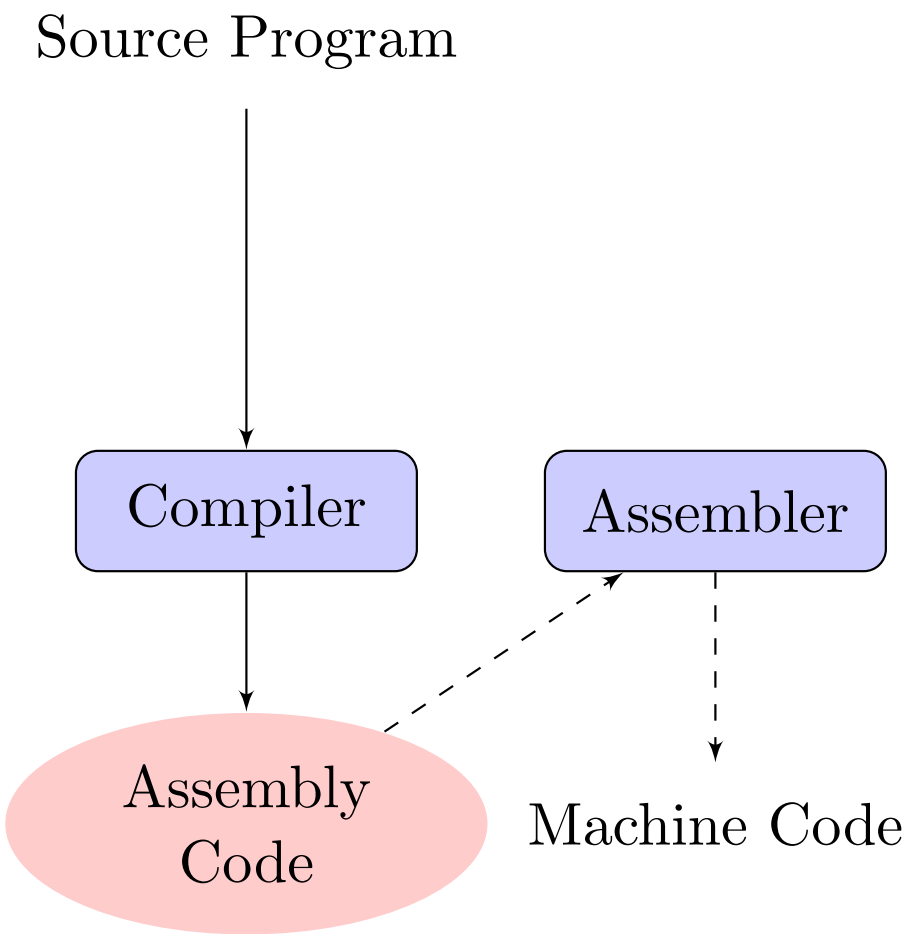


Figure: Fortran program Compilation

Compilation and Interpretation

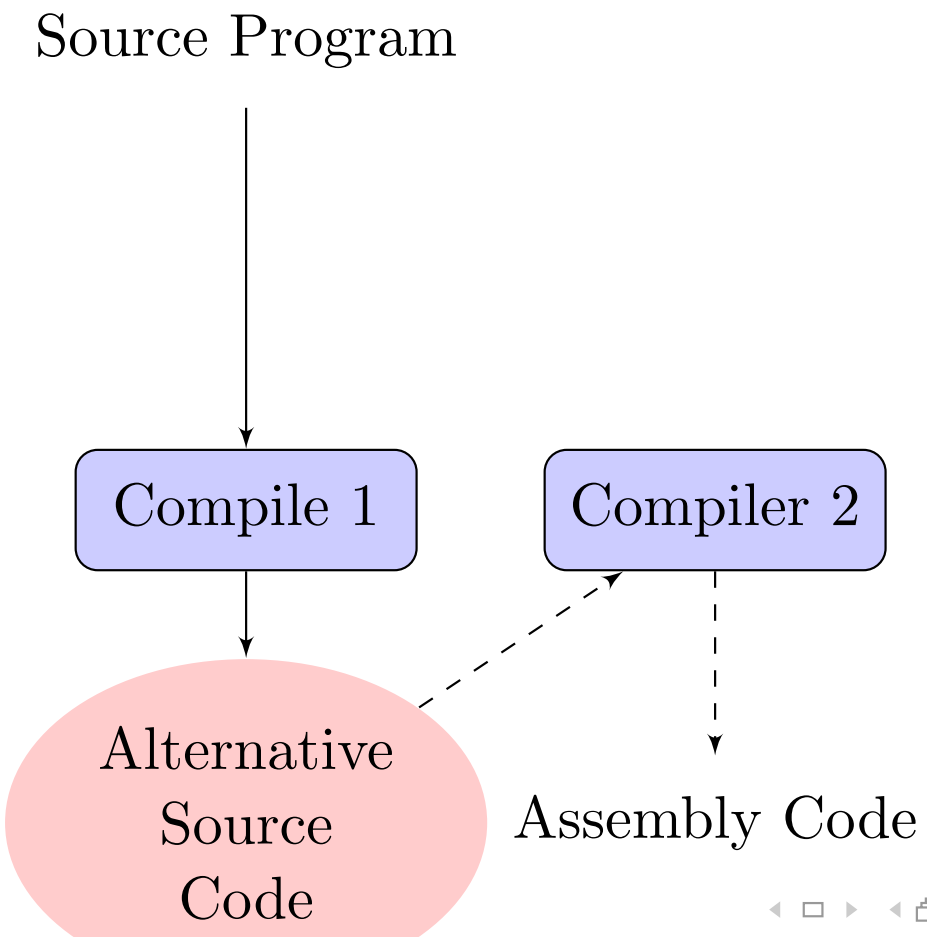
Post-compilation assembly



Compilation and Interpretation

Source-to-source translation

Example - AT&T's original compiler for **C++**



Compilation and Interpretation

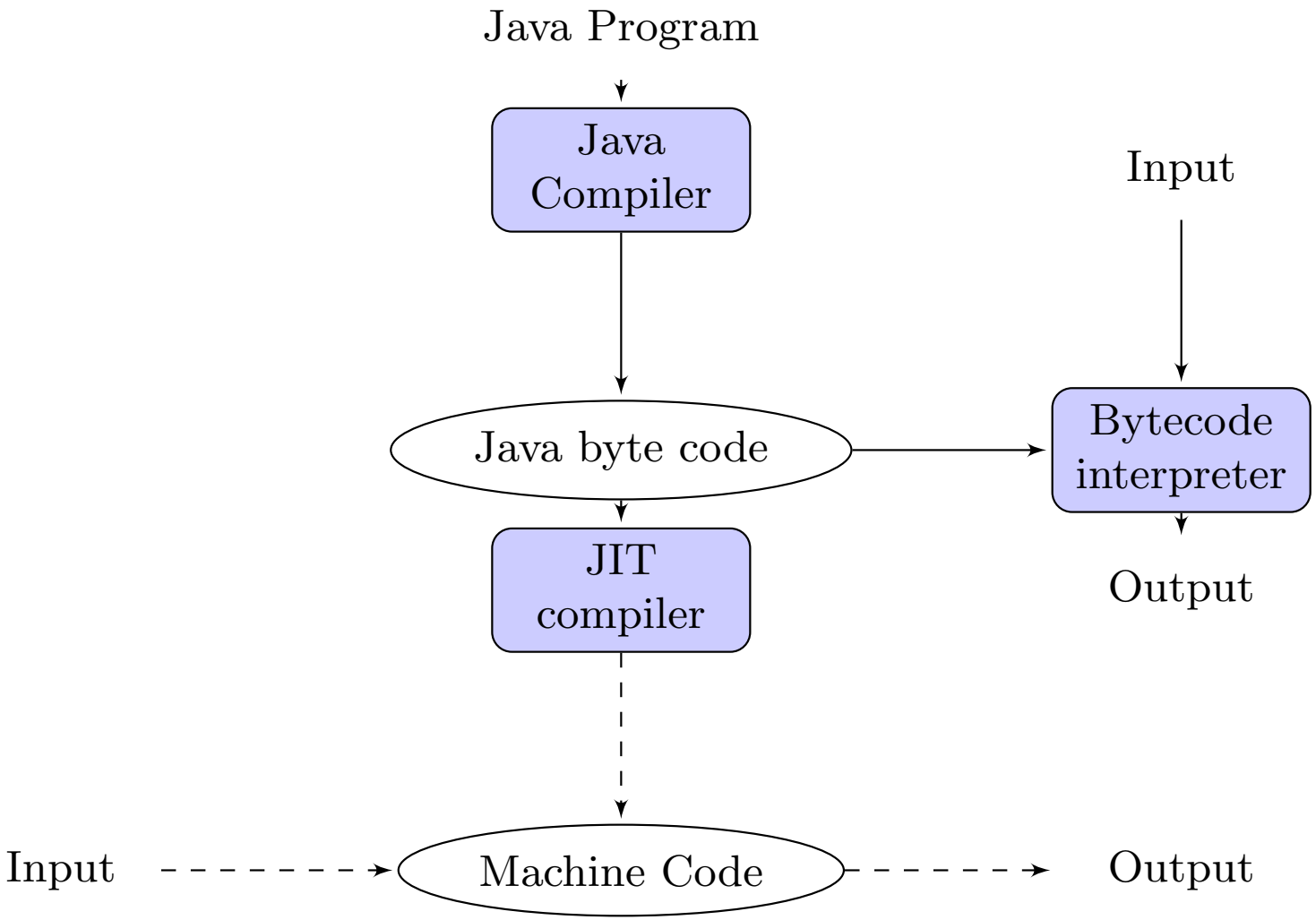


Figure: Just-In Time compilation

Language-Processing System

#define MAX 100

Preprocessor: expands "macros" and combines source program modules.

Compiler: translates source language to symbolic (assembler) machine code.

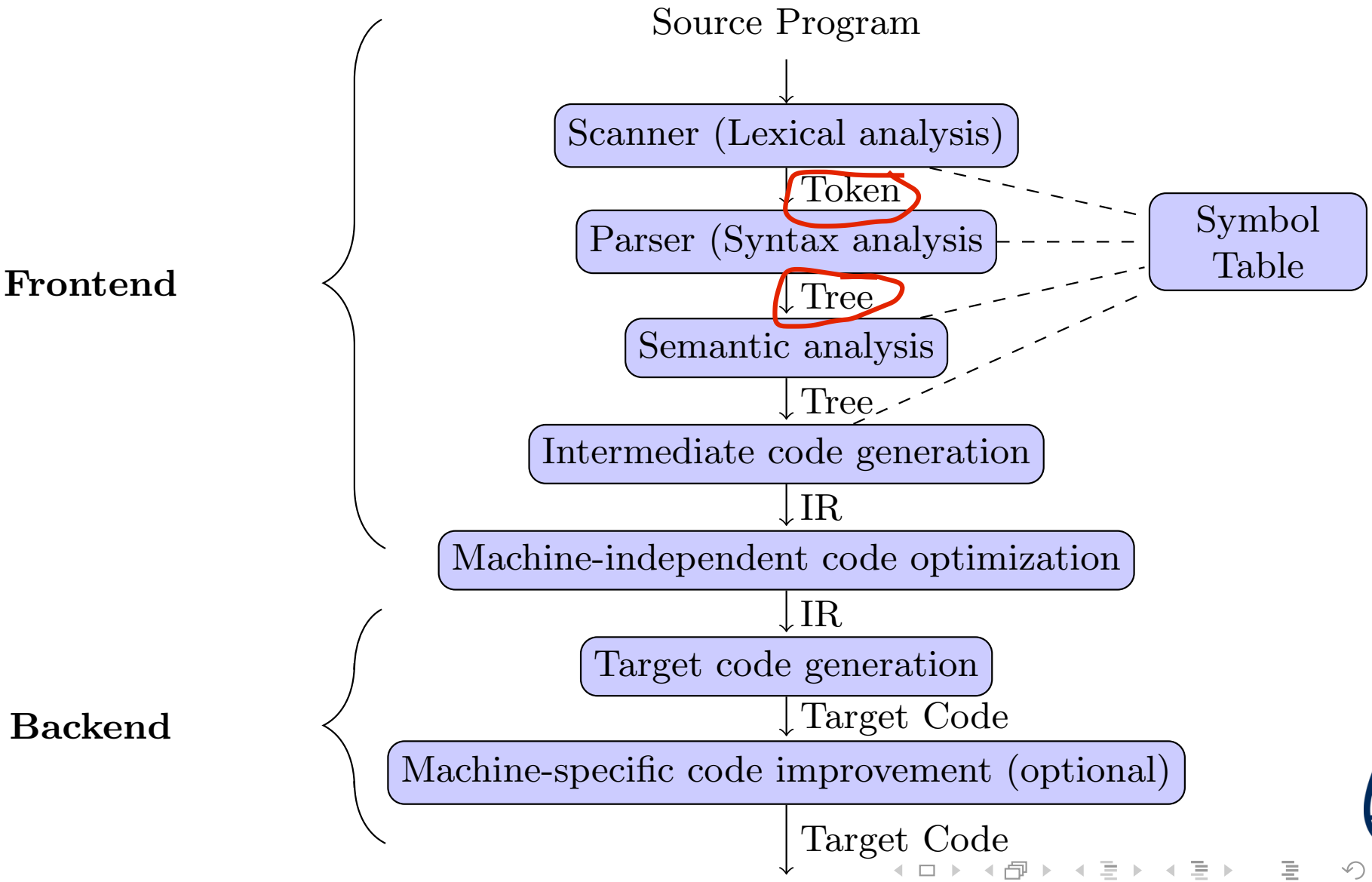
Assembler: translates symbolic machine code to relocatable binary code.

Linker: resolves links to library files and other relocatable object files.

Loader: combines executable object files into memory

Outline

The Phases of Compilation



The Phases of Compilation

- The Phases of Compilation
  - Lexical analysis
  - Syntactic analysis.
  - Semantic analysis.
  - Intermediate code generation.
  - Optimization.
  - Machine code generation.

Outline

Lexical Analysis

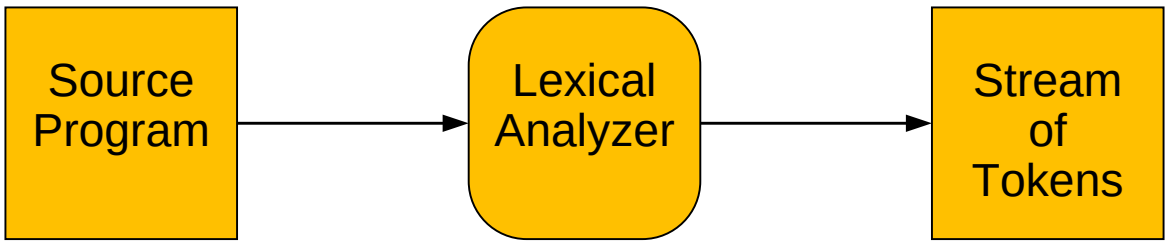
```
l -->[a-zA-Z]
d—> [0-9]
id —>l (l|d)*
```

Definition (Token)

A token is a smallest meaningful group symbols.

Definition (Lexical analyzer)

A lexical analyzer, also called a lexer or a scanner, receives a stream of characters from the source program and groups them into tokens.



Example

Example (Lexical Analysis)

- What are the tokens in the following program?

```
int main()
{
float a = 123.4;
return 0;
}
```

Tokens

- Each token has a type and a value.
- For example,
  - The variable `count` has type **id** and value "`count`".
  - The number `123` has type **num** and value "123".
  - The keyword `int` has type **int** and value "int".
  - The symbol { has the type **lbrace** and value "lbrace".

Example

a = “hello” * 3

Example (Lexical Analysis)

- The statement

  ```
  position = initial + rate * 60;
  ```

  would be viewed as

  $$\mathbf{id}_1 \ = \ \mathbf{id}_2 + \mathbf{id}_3 \ * \ \mathbf{num} \ ;$$

  or

  $$\mathbf{id}_1 \ \mathbf{assign} \ \mathbf{id}_2 \ \mathbf{plus} \ \mathbf{id}_3 \ \mathbf{times} \ \mathbf{num} \ \mathbf{semi}$$

  by the lexer.

Lexical Analysis Tools

- There are tools available to assist in the writing of lexical analyzers.
  - `lex` - produces C source code (UNIX).
  - `flex` - produces C source code (gnu).
  - `JLex` - produces Java source code.
  - `JFlex` - produces Java source code.

Outline

Syntactic Analysis

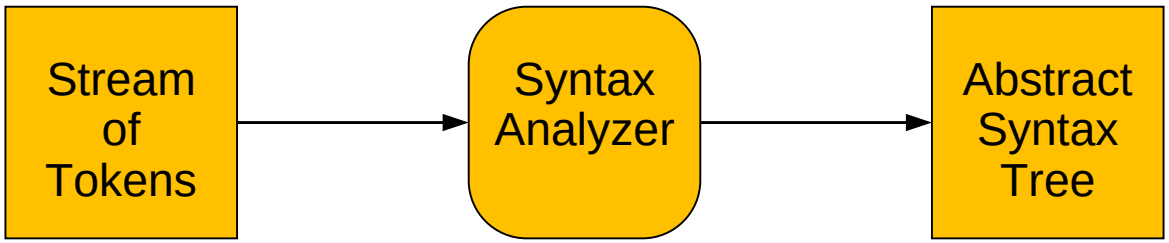
Definition (Syntax analyzer)

A syntax analyzer, also called a parser, receives a stream of tokens from the lexer and groups them into phrases that match specified grammatical patterns.

Syntactic Analysis

Definition (Abstract syntax tree)

The output of the parser is an abstract syntax tree representing the syntactical structure of the tokens.



Grammatical Patterns

- Grammatical patterns are described by a context-free grammar.
- For example, an assignment statement may be defined as

p = i + r * 60;
p = i ++ r * 60;

G

$stmt \rightarrow \mathbf{id} \ = \ expr \ ;$

$expr \rightarrow expr \ + \ expr \mid expr \ * \ expr \mid \mathbf{id} \mid \mathbf{num}$

id = id ++ id * num

stmt ==> id + id * id
stmt ==> id * * num

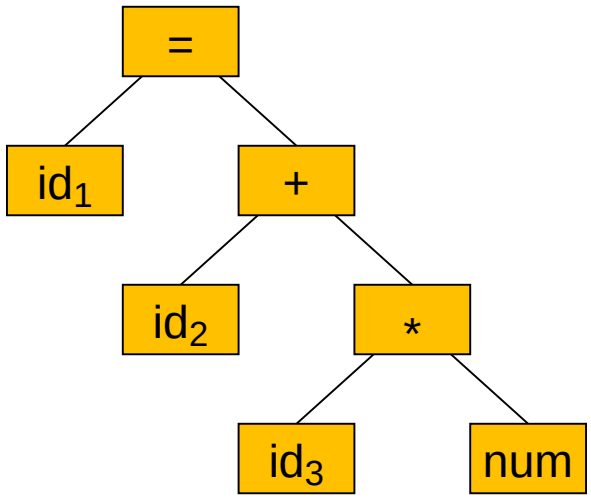
Example

Example (Syntactic Analysis)

- The form

$$\mathbf{id}_1 = \mathbf{id}_2 \ + \ \mathbf{id}_3 \ * \ \mathbf{num} \ ;$$

may be represented by the following syntax tree.



Syntax Analysis Tools

- There are tools available to assist in the writing of parsers.
  - `yacc` - produces C source code (UNIX).
  - `bison` - produces C source code (gnu).
  - `CUP` - produces Java source code.

Outline

a = “hello” * 3

Semantic Analysis

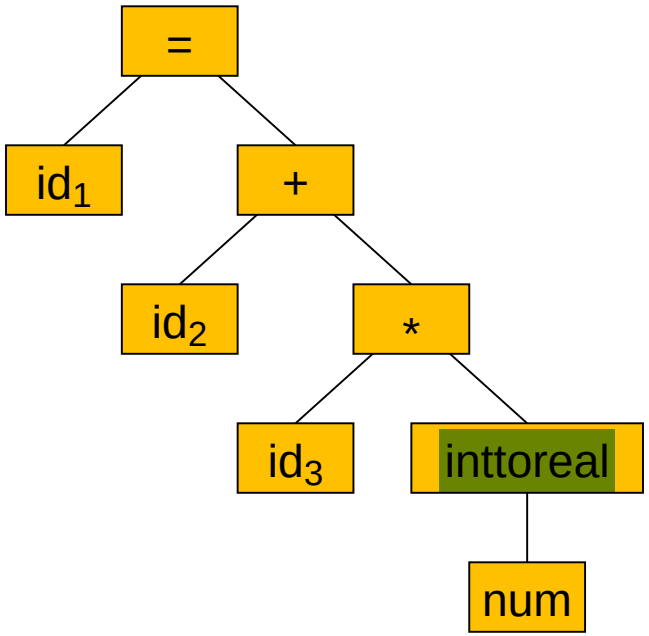
Definition (Semantic analyzer)

A semantic analyzer traverses the abstract syntax tree, checking that each node is appropriate for its context, i.e., it checks for semantic errors. It outputs a refined abstract syntax tree.

Example: Semantic Analysis

Example (Semantic Analysis)

- The previous tree may be refined to




```
int a;
float b;
a = int(b) / 2;
int a[b];
div
fdiv
mul
fmul
imul
```

Outline

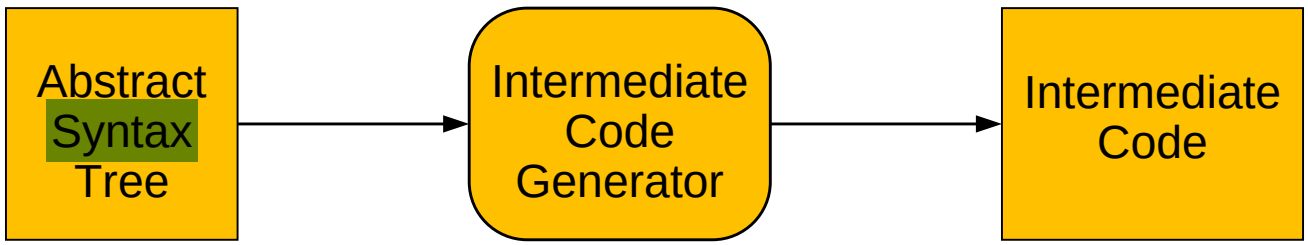
Intermediate Code Generation

Definition (Intermediate code)

Intermediate code is code that represents the semantics of a program, but is machine-independent.

Definition (Intermediate code generator)

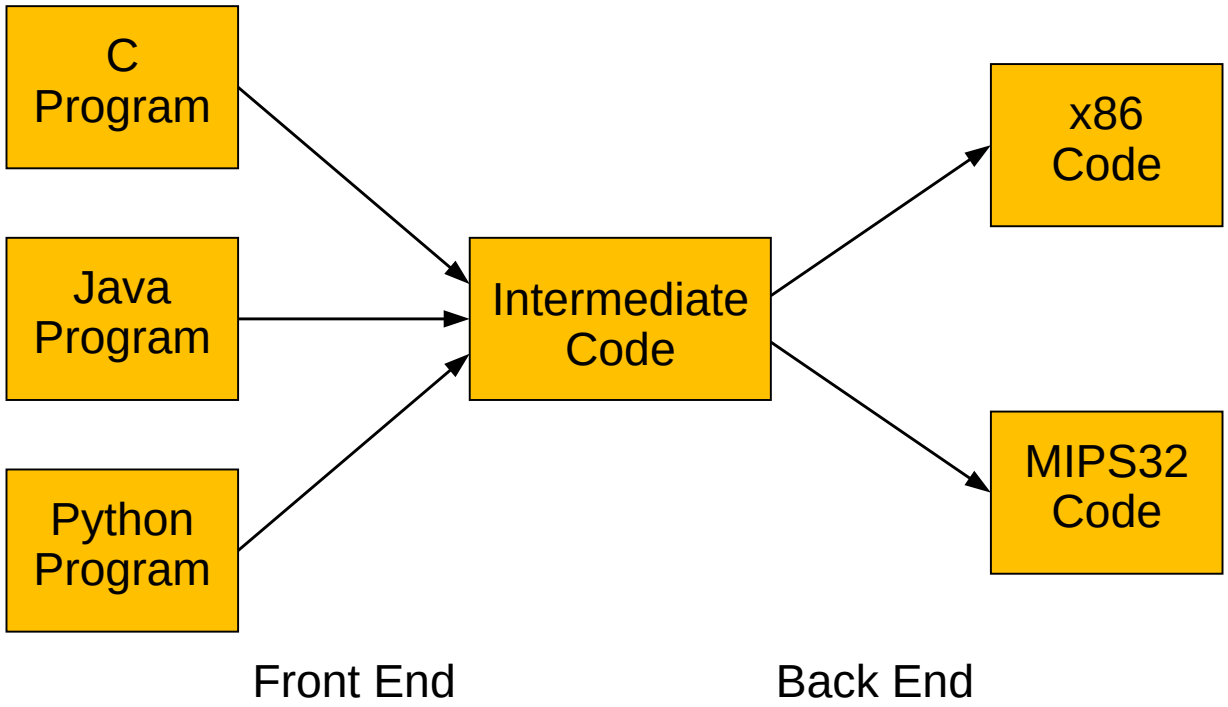
An intermediate code generator receives the abstract syntax tree and outputs intermediate code that semantically corresponds to the abstract syntax tree.



Intermediate Code

- This stage marks the boundary between the front end and the back end.
- The front end is language-specific and machine-independent.
- The back end is machine-specific and language-independent.

Intermediate Code



Example

position = initial + rate * 60;

id1 = id2 + id3*60;

id1 = id2 + id3* inttoreal(60)

Example (Intermediate Code Generation)

- The tree in our example may be expressed in intermediate code as

```
temp1 = inttoreal(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

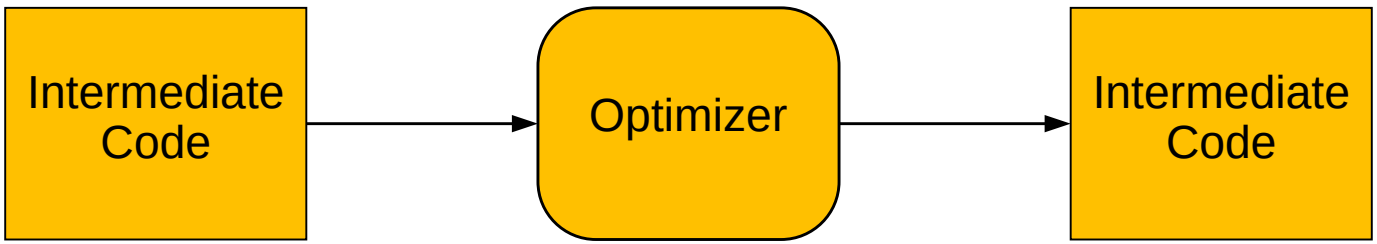
Outline

Code Optimizer

Definition (Optimizer)

An optimizer reviews the code, looking for ways to reduce the number of operations and the memory requirements.

- A program may be optimized for speed or for size.
- Typically there is a trade-off between speed and size.



Example

```
t2 = id3 * 60.0
id1 = id2 + t2
```

Example (Optimization)

- The intermediate code in this example may be optimized as

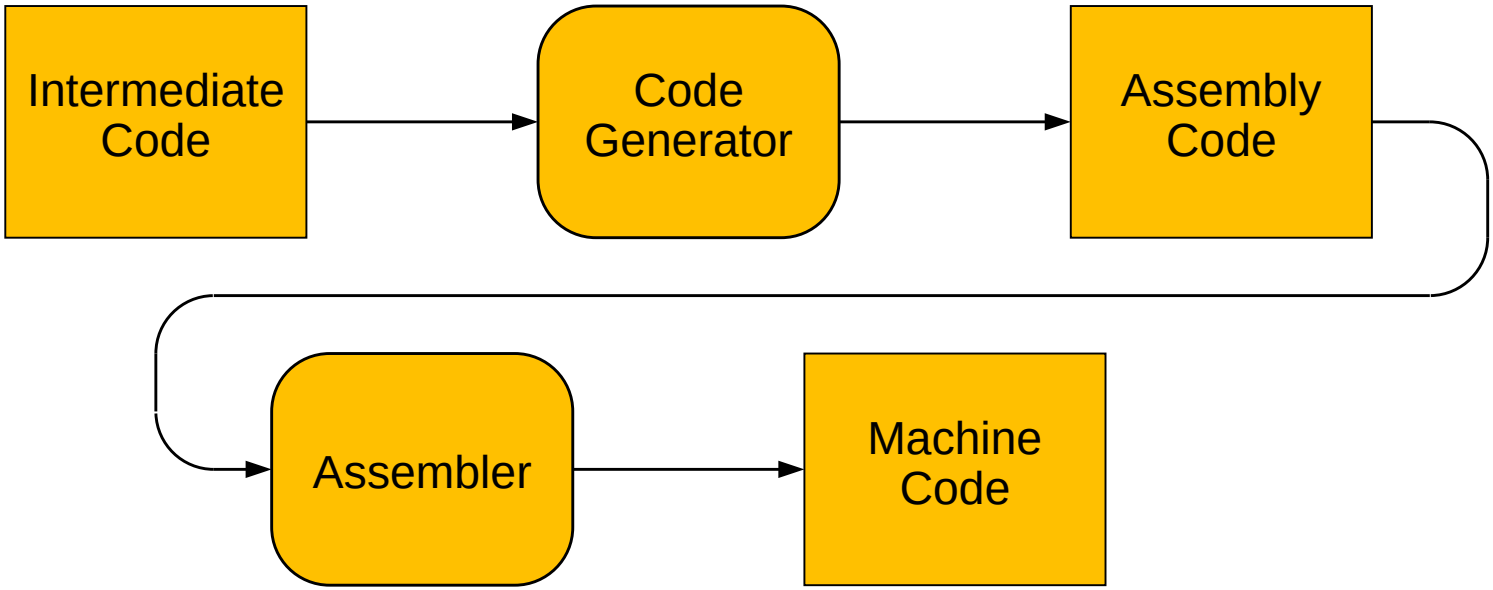
```
temp1 = id3 * 60.0
id1 = id2 + temp1
```

Outline

Machine Code Generation

- The code generator receives the (optimized) intermediate code.
- It produces either
  - Machine code for a specific machine, or
  - Assembly code for a specific machine and assembler.
- If it produces assembly code, then an assembler is used to produce the machine code.

Machine Code Generation



Example: Machine Code Generation

```
t2 = id3 * 60.0
id1 = id2 + t2
```

- The intermediate code may be translated into the assembly code

```
movf id3,R2
mulf #60.0,R2
movf id2,R1
addf R2,R1
movf R1,id1
```

Symbol Table

Records are < variable name, attributes>.

Attributes:

- storage information,
- type,
- scope (static/lexical,dynamic),
- procedure names: number and types of arguments,
- argument passing(by value or by reference)
- return type.

Design principle: **find**, **store**, **retrieve** records **quickly**.

Compiler-construction tools

Commonly used tools:

scanner generators: token description (RE) $\rightarrow$ lexical analyser,

parser generators: grammar $\rightarrow$ syntax analyser,

syntax-directed translation engines: syntax tree $\rightarrow$ IR,

code-generator generators: translation rules $\rightarrow$ code generator,

Data-flow engines: data-flow information analyzers (optimization)

Outline

Assignment

Assignment

- Read Chapter 1