Chapter 1

Introduction

Muhammad Kamal Hossen
Associate Professor
Dept. of CSE, CUET
Email: kamalcsecuet@gmail.com

Contents

- Introduction
- A Simple Syntax-Directed Translator
- Lexical Analysis
- Syntax Analysis
- Syntax-Directed Translation
- Intermediate-Code Generation
- Type Checking
- Run-time Environments
- Code Generation
- Code Optimization

Introduction

Programming languages are notations for describing computations to people & to machines

- World depends on *Programming Languages*
  - Compilers
- **Areas:** The study of compiler writing (principles and techniques) touches upon -
  - Programming languages
  - Machine architecture
  - Language theory
  - Algorithms
  - Software engineering

Language Processors

- **Compiler:** is a program that can read a program in one language (source language) and translate it into an equivalent program in another language (target language)
  - The essential interface between applications & architectures

Mapping inputs to outputs

Language Processors

Interpreter: instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs

Statement-by-statement

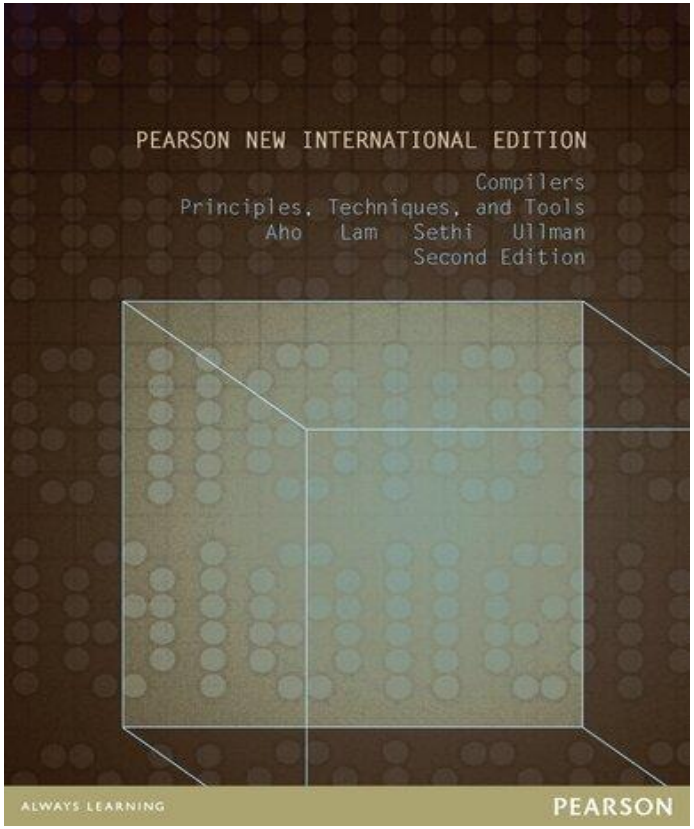
Recommended Book

- **Compilers: Principles, Techniques, and Tools**

-Alfred V. Aho

-Ravi Sethi

-Jeffrey D. Ullman

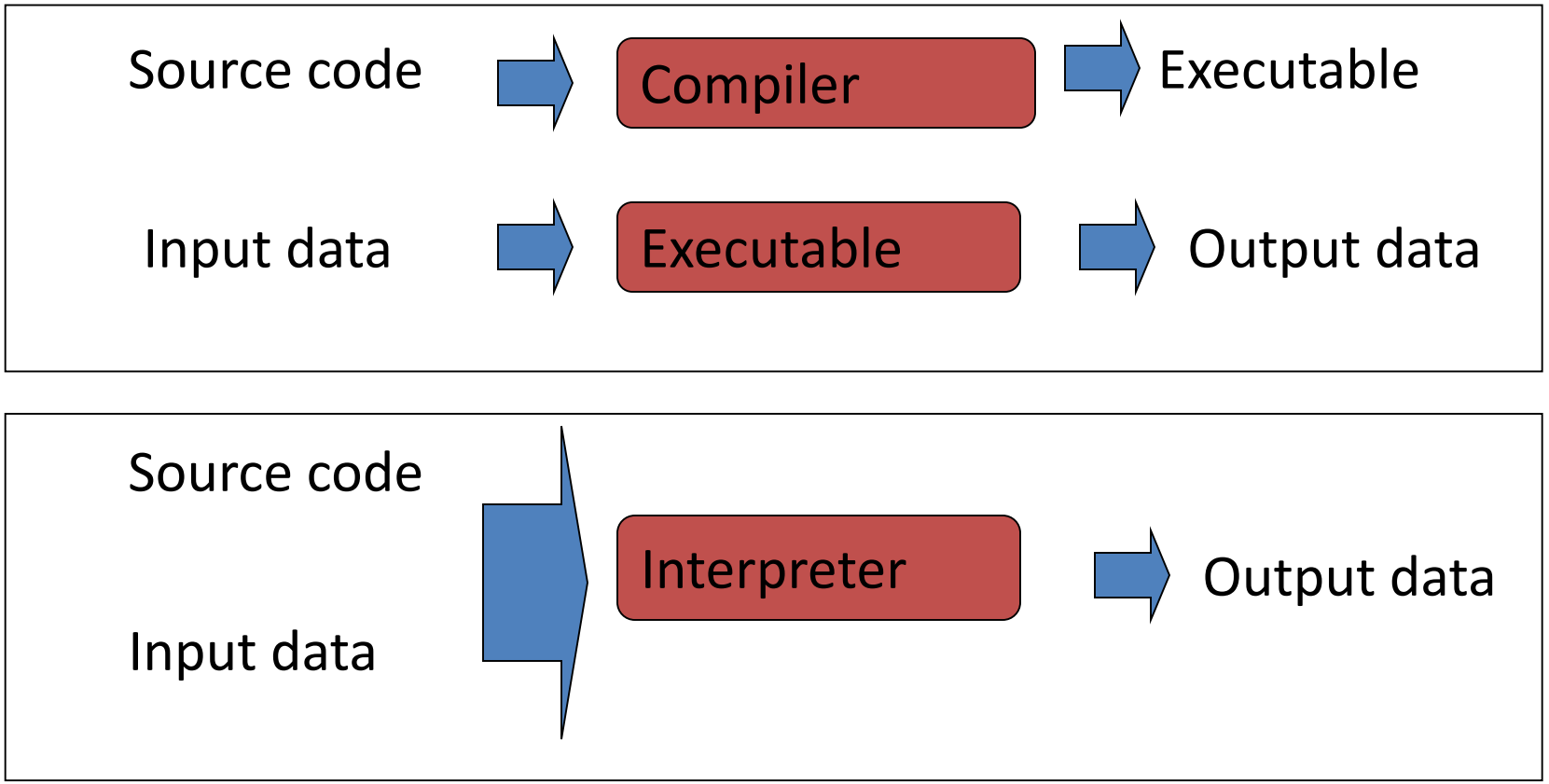


Compiler vs. Interpreter (1/5)

- Compilers: Translate a source (human-writable) program to an executable (machine-readable) program
- Interpreters: Convert a source program and execute it at the same time.

Compiler vs. Interpreter (2/5)

Ideal concept:

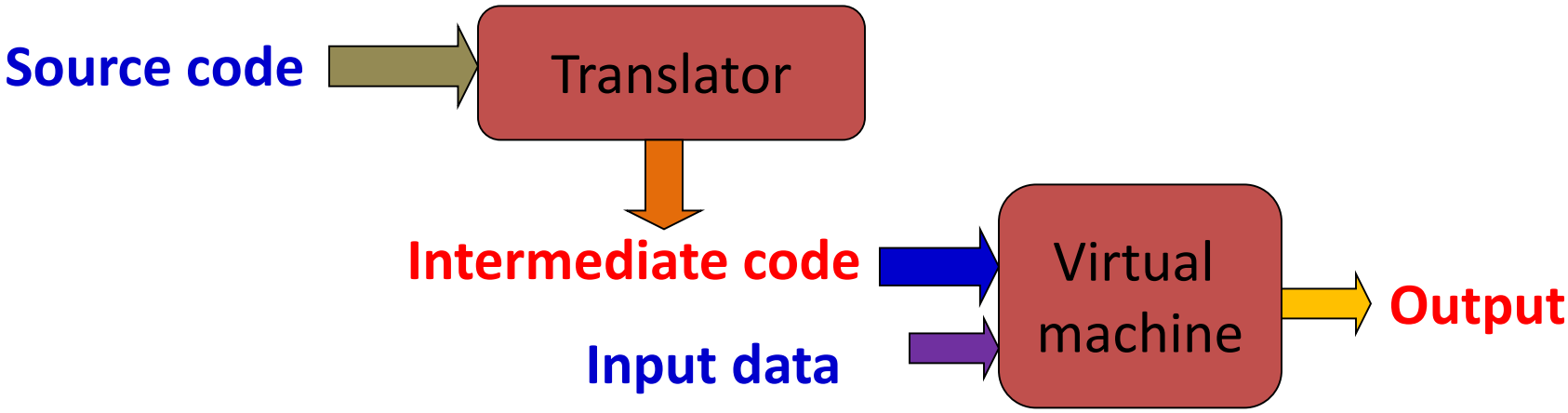


Compiler vs. Interpreter (3/5)

- Most languages are usually thought of as using either one or the other:
  - Compilers: FORTRAN, COBOL, C, C++, Pascal, PL/1
  - Interpreters: Lisp, scheme, BASIC, APL, Perl, Python, Smalltalk
- **BUT**: not always implemented this way
  - Virtual Machines (e.g., Java)
  - Linking of executables at runtime
  - JIT (Just-in-time) compiling

Compiler vs. Interpreter (4/5)

- Actually, no sharp boundary between them. General situation is a combo:



- Java source program may first be compiled into an intermediate code (***Bytecodes***)
- Bytecodes are then interpreted by a virtual machine.
- ❑ **Benefit**: Bytecodes compiled on one machine can be interpreted on another machine (network)

Compiler vs. Interpreter (5/5)

Compiler

- **Pros**
  - Less space
  - Fast execution
- **Cons**
  - Slow processing
    - Partly Solved (Separate compilation)
  - Debugging
    - Improved through IDEs

Interpreter

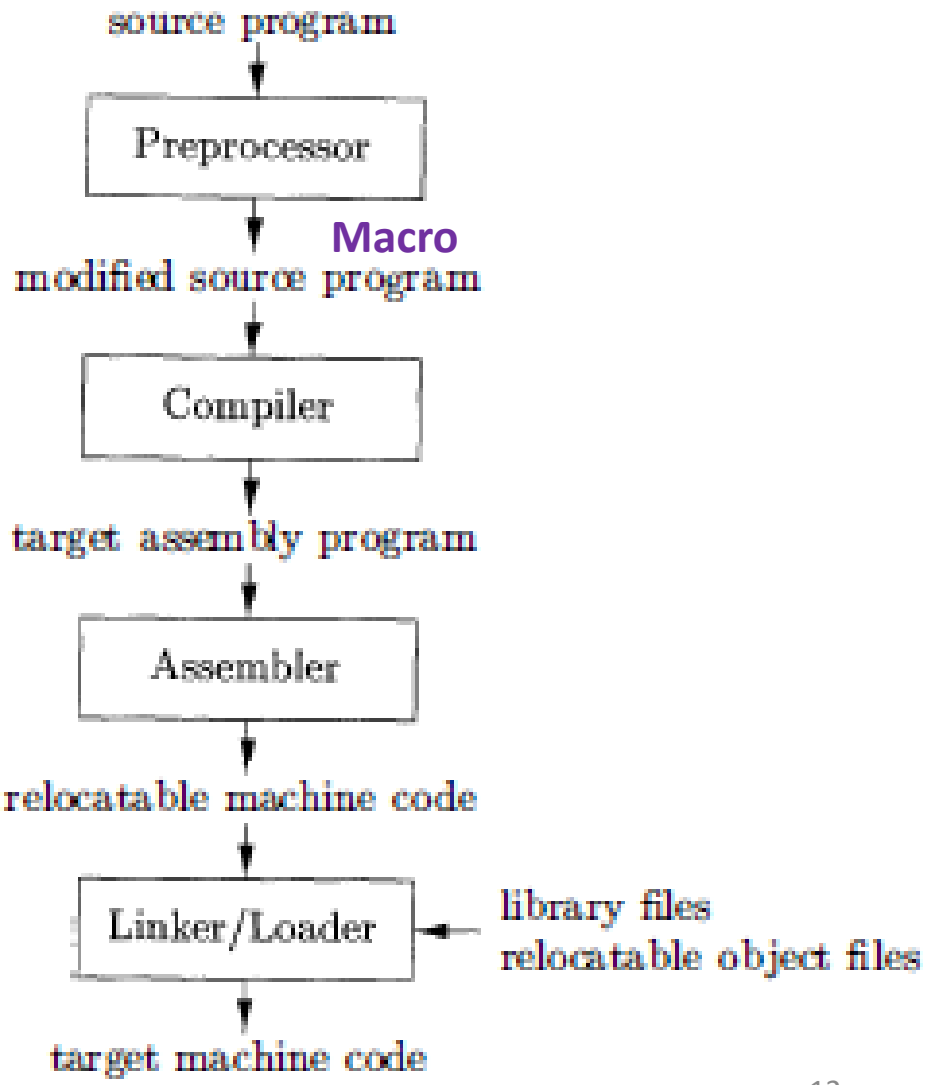
- **Pros**
  - Easy debugging
  - Fast Development
- **Cons**
  - Not for large projects
    - Exceptions: Perl, Python
  - Requires more space
  - Slower execution
    - Interpreter in memory all the time

A Language Processing System

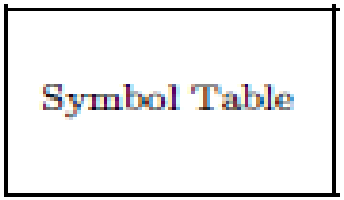
- The task of collecting the source program is sometimes entrusted to a separate program (**preprocessor**)

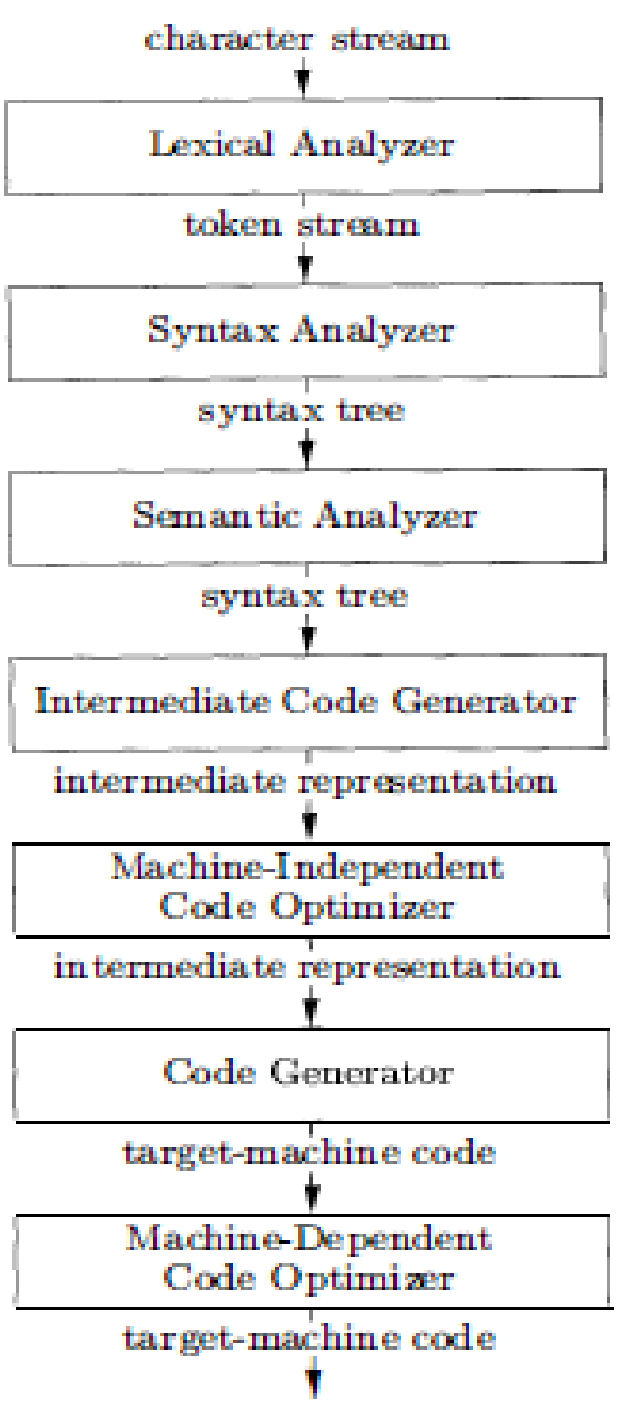
- The **linker** resolves external memory addresses, where the code in one file may refer to a location in another file.

- The **loader** then puts together all of the executable object files into memory for execution.



Phase of compilations





Scanning/Lexical analysis

- Break program down into its smallest meaningful symbols (tokens, atoms, lexemes)
- Tools for this include lex, flex
- Tokens include e.g.:
  - "Reserved words": do if float while
  - Special characters: **({ , + - = ! /**
  - Names & numbers: **myValue 3.07e02**
- Start symbol table with new symbols found

Scanning/Lexical analysis

- For each lexeme, lexical analyzer produces as output a token: <*token-name, attribute-value*>
  - *token- name:* abstract symbol that is used during syntax analysis ,
  - *attribute-value:* points to an entry in the symbol table for this token.

Scanning/Lexical analysis

- Assignment Statement (source program):

position = initial + rate * 60

- **Lexemes:**

1. **position** is a lexeme that would be mapped into a token **<id, 1>,** where **id** is an abstract symbol standing for identifier and **1** points to the symbol table entry for position.

- The symbol-table entry for an identifier holds information about the identifier, such as its name and type.

Scanning/Lexical analysis

- **2.** The assignment symbol **=** is a lexeme that is mapped into the token **<=>.**
- **3. initial** is a lexeme that is mapped into the token **<id, 2>** , where **2** points to the symbol-table entry for **initial**
- **4. +** is a lexeme that is mapped into token **<+>**
- **5. rate** is a lexeme that is mapped into the token **<id, 3>** , where **3** points to the symbol-table entry for **rate**.

Scanning/Lexical analysis

- **6.** * is a lexeme that is mapped into token **<*>**
- **7. 60** is a lexeme that is mapped into the token **<60>**
- Blanks (White Space) separating the lexemes would be discarded by the lexical analyzer.

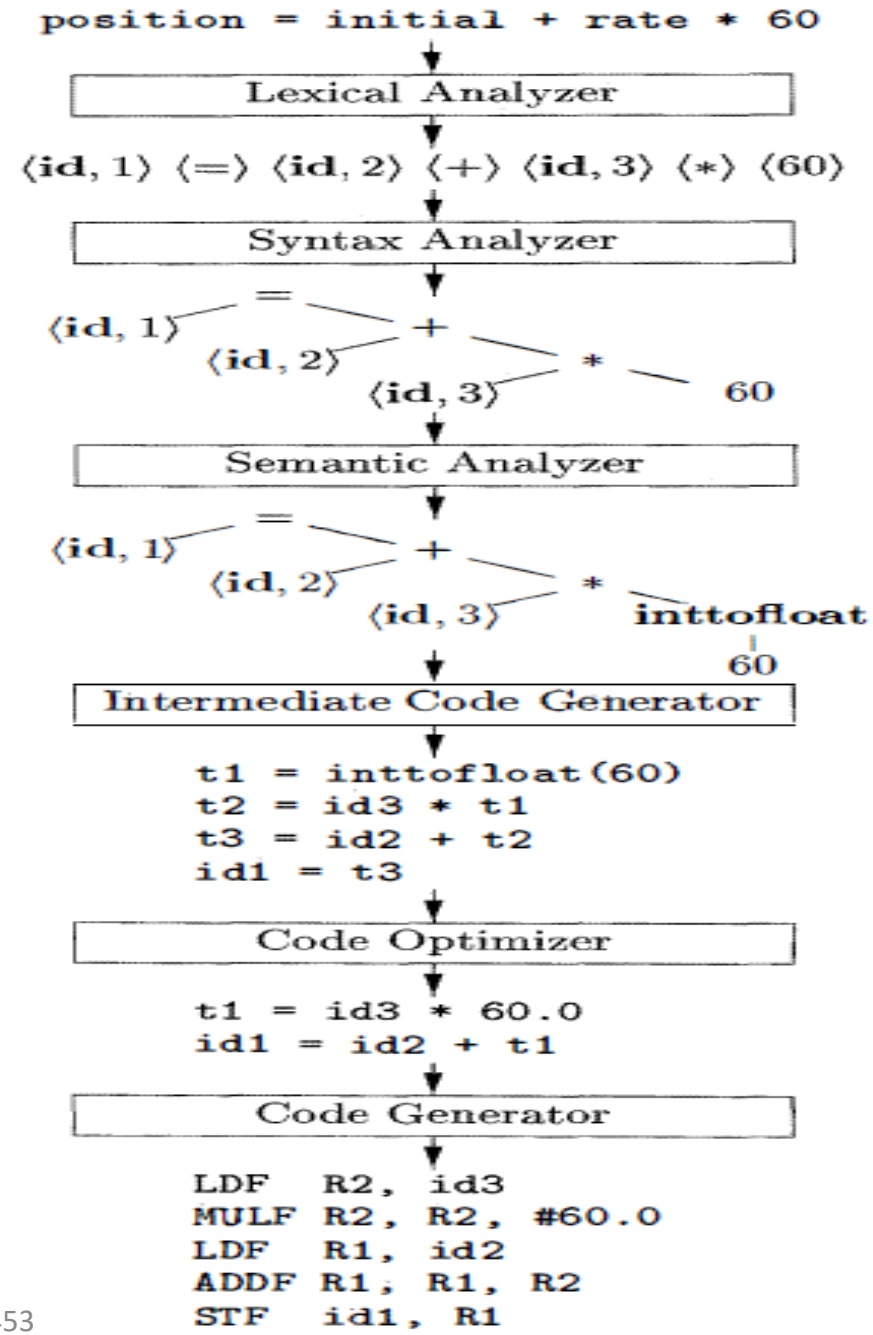
After lexical analysis as the sequence of tokens

<**id**, 1> < = > <**id**, 2> <+ > <**id**, 3> <*> <**60**>

Translation of an assignment statement

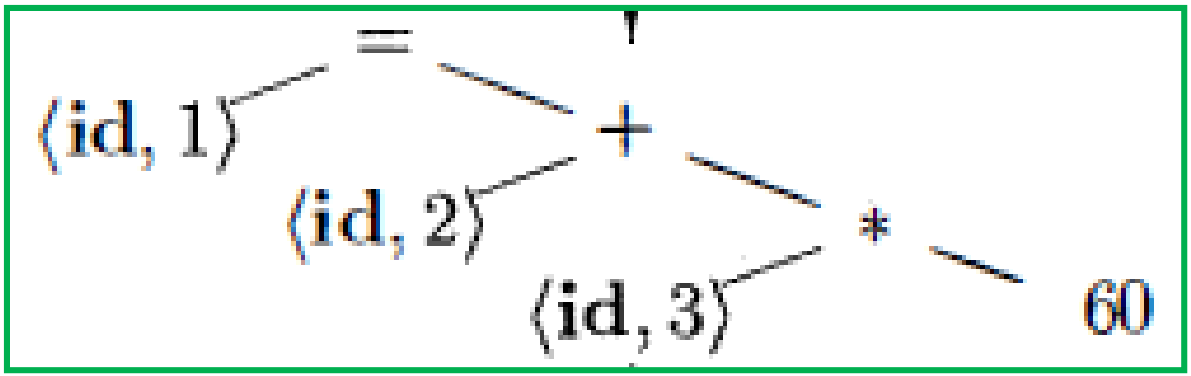
1	position	···
2	initial	···
3	rate	···

SYMBOL TABLE



Parsing/Syntax Analysis

- The parser create a tree-like intermediate representation that depicts the grammatical structure of the token stream.
- A typical representation is a **syntax/parse tree** in which each interior node represents an operation and the children of the node represent the arguments of the operation.



Parsing/Syntax Analysis

- This tree shows the order in which the operations in the assignment are to be performed:

position = initial + rate * 60

- The tree has an interior node labeled * with **<id, 3>** as its left child and the integer **60** as its right child. The node <id, 3> represents the identifier ***rate***.
- The node labeled * makes it explicit that we must first multiply the value of ***rate*** by **60**.
- The node labeled **+** indicates that we must add the result of this multiplication to the value of ***initial***.

Parsing/Syntax Analysis

- The root of the tree, labeled =, indicates that we must store the result of this addition into the location for the identifier ***position***.
- This ordering of operations is consistent with the usual conventions of arithmetic which tell us -
  - multiplication has higher precedence than addition, and hence that the multiplication is to be performed before the addition.

Semantic Analysis

- The semantic analyzer uses the syntax tree & the information in the symbol table to check the source program for semantic consistency with the language definition.
- It also gathers type information & saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

Semantic Analysis

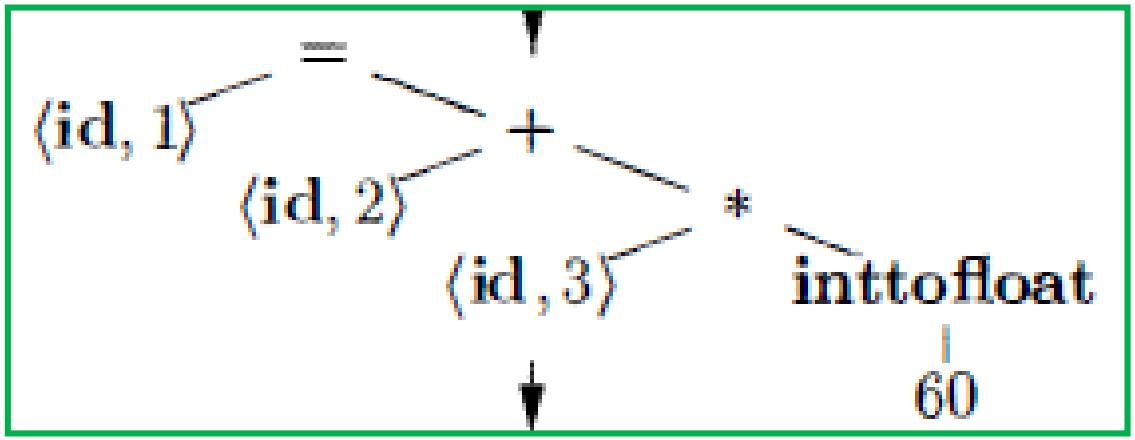
- Important part: **type checking**
- compiler checks that each operator has matching operands.
- Ex.: many programming language definitions require an array index to be an integer;
  - the compiler must report an error if a floating-point number is used to index an array.

Semantic Analysis

- The language specification may permit some type conversions called **coercions**.
- Suppose that ***position, initial***, and ***rate*** have been declared to be floating-point numbers, and that the lexeme **60** by itself forms an integer.
- The type checker discovers that the operator ***** is applied to a floating-point number ***rate*** & an integer **60**.

Semantic Analysis

- In this case, the integer may be converted into a floating-point number.
- The output of the semantic analyzer has an extra node for the operator **inttofloat**, which explicitly converts its integer argument into a floating-point number.



Intermediate Code Generation

- **one or more intermediate representations**, which can have a variety of forms.
- Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis.
- **02 important properties**:

➢ it should be easy to produce

➢ it should be easy to translate into the target machine

Intermediate Code Generation

- **Three-address code**: a sequence of assembly-like instructions with three operands per instruction.
- Each operand can act like a register.

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Intermediate Code Generation

- **Properties:**

➢ Each three-address assignment instruction has at most one operator on the right side.

➢ Compiler must generate a temporary name to hold the value computed by a three-address instruction.

➢ Some "three-address instructions“ have fewer than three operands.

Code Optimization

- The code-optimization phase attempts to improve the intermediate code so that better target code will result.
- Better means *faster*,

➢ shorter code, or target code that consumes less power.

- Ex.: a straightforward algorithm generates the intermediate code, using an instruction for each operator in the tree representation that comes from the semantic analyzer.

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Code Optimization

- The optimizer can deduce that the conversion of 60 from integer to floating point can be done once and for all at compile time,
  - ✓ so the **inttofloat** operation can be eliminated by replacing the integer 60 by the floating-point number 60.0.
- Moreover, t_3 is used only once to transmit its value to id_1, so the optimizer can transform into the shorter sequence

```
t1 = id3 * 60.0
id1 = id2 + t1
```

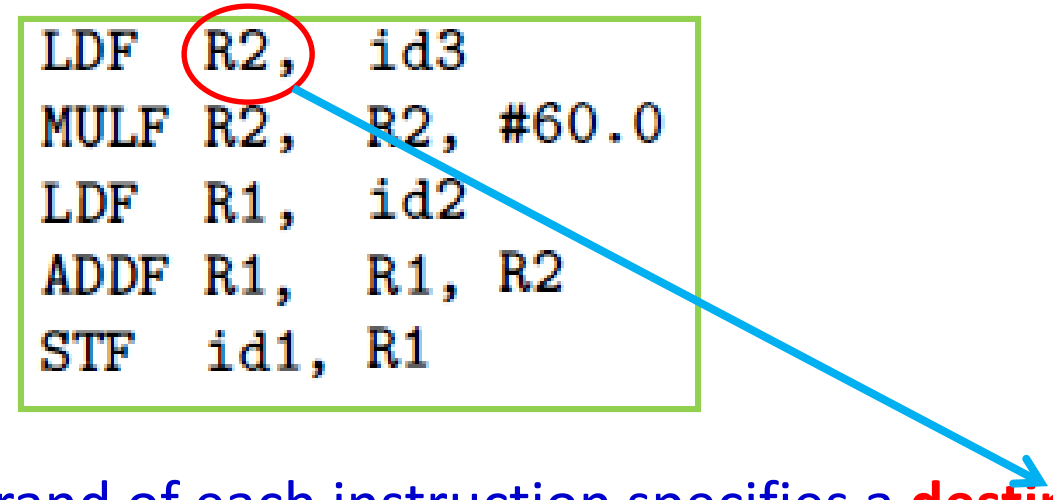
Code Generation

- The code generator takes as input an intermediate representation of the source program and maps it into the target language.
- If the target language is machine code, registers or memory locations are selected for each of the variables used by the program.
- Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.

Code Generation

- Ex.: using registers R1 and R2, the intermediate code might get translated into machine code:

```
LDF   R2,  id3
MULF  R2,  R2, #60.0
LDF   R1,  id2
ADDF  R1,  R1, R2
STF   id1, R1
```



- ❑ The first operand of each instruction specifies a **destination**.
- ❑ The **F** in each instruction tells us that it deals with floating-point numbers

Code Generation

- The code loads the contents of address **id3** into register **R2**,
- Then multiplies it with floating-point constant **60.0**.
- The **#** signifies that 60.0 is to be treated as an immediate constant
- The third instruction moves **id2** into register **R1**
- Fourth adds to it the value previously computed in register R2.
- Finally, the value in register **R1** is stored into the address of **id1**
- So the code correctly implements the assignment statement:

position = initial + rate * 60

Symbol-Table Management

- An essential function of a compiler
  - ✓ is to record the variable names used in the source program
  - ✓ collect information about various attributes of each name.

❑ These attributes may provide information about

- ➢ the storage allocated for a name, its type, its scope (where in the program its value may be used), and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned.

❑ The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name.

❑ The data structure should be designed to allow the compiler to find the record for each name quickly & to store or retrieve data from that record quickly.

Symbol-Table Management

Symbol Table:

Symbol	Type	Scope	Address	...
x	int	Global	0×1000	...
y	float	Local	0×2000	...
sum	int	Local	0×3000	...
counter	int	Global	0×4000	...
result	float	Local	0×5000	...
...	...	...	...	...

Compiler Construction Tools

- The compiler writer can profitably use modern software development environments:
- **Tools**: language editors, debuggers, version managers, profilers, test harnesses, and so on.
- **Properties of the most successful tools:**
  - Hide the details of the generation algorithm
  - Produce components that can be easily integrated into the remainder of the compiler.

Commonly used Compiler Construction Tools

1. **Parser generators** that automatically produce syntax analyzers from a grammatical description of a programming language.

2. **Scanner generators** that produce lexical analyzers from a regular-expression description of the tokens of a language.

3. **Syntax-directed translation engines** that produce collections of routines for walking a parse tree and generating intermediate code.

Commonly used Compiler Construction Tools

4. **Code-generators** that produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.

5. **Data-flow analysis engines** that facilitate the gathering of information about how values are transmitted from one part of a program to each other part.

   ✓ Data-flow analysis is a key part of code optimization.

6. **Compiler-construction toolkits** that provide an integrated set of routines for constructing various phases of a compiler.

The Evaluation of Programming Language

- **1940**: 1st electronic computer
- **Programmed**: machine language (by sequences of 0's and 1's) that explicitly told the computer what operations to execute and in what order.
- **Limitations**: Operations in very low level:
  - move data from one location to another, add the contents of two registers, compare two values , & so on.
- **Disadv.**: programming was slow, tedious, and error prone.
  - once written, the programs were hard to understand & modify.

The Move to Higher-level Language

- **Early 1950**: Assembly languages (mnemonic)
- Later, macro instructions were added to assembly languages so that a programmer could define parameterized shorthands for frequently used sequences of machine instructions.

The Move to Higher-level Language

- **Latter half of the 1950's**: A major step towards higher-level languages was made
  - Fortran for scientific computation,
  - Cobol for business data processing,
  - Lisp for symbolic computation.
- The philosophy behind these languages was to create higher-level notations with which programmers could more easily write numerical computations, business applications, and symbolic programs.
- These languages were so successful that they are still in use today.

Classification

- Today, there are thousands of programming languages.
- **Classification:**

1. **According to Generation**

- **First-generation:** machine languages
- **Second-generation:** assembly languages,
- **Third-generation:** higher-level languages (Fortran, Cobol, Lisp, C, C++, C#, and Java)
- **Fourth-generation:** designed for specific applications like NOMAD for report generation, SQL for database queries, and Postscript for text formatting.
- **Fifth-generation:** applied to logic- and constraint-based languages (Prolog and OPS5)

Classification

2. **Imperative for languages**

   - a program specifies how a computation is to be done and declarative for languages in which a program specifies what computation is to be done.

- Languages such as C, C++, C#, and Java are imperative languages.
- In imperative languages, there is a notion of program state and statements that change the state.
- Functional languages such as ML and Haskell and constraint logic languages such as Prolog are often considered to be **declarative languages**.

Classification

3. **von Neumann language**

- Computational model is based on the von Neumann computer architecture.
- Fortran and C are von Neumann languages.

Classification

4. **An object-oriented language**

- Supports object-oriented programming, a programming style in which a program consists of a collection of objects that interact with one another.

➢ Simula 67 and Smalltalk are the earliest major object-oriented languages.

➢ C++, C#, Java, and Ruby are more recent object-oriented languages.

Classification

5. **Scripting languages**

- interpreted languages with high-level operators designed for "gluing together" computations.
- These computations were originally called "scripts."
- Awk, JavaScript, Perl, PHP, Python, Ruby, and Tel are popular examples of scripting languages.
- Programs written in scripting languages are often much shorter than equivalent programs written in languages like C.

Application of Compiler Technology

- Implementation of High-level programming language
- *Optimizations for Computer Architecture: parallelism, Memory Hierarchies*
- Design of New Computer Architecture: RISC, Specialized architecture
- *Debugging*
- Fault location
- Model checking in formal analysis
- Model-driven development
- Optimization techniques in software engineering
- *Program Translation: Binary translation, Hardware synthesis, database query interpreters*
- Software productivity tools: Type checking, bounds checking, memory-management, software maintenance
- *Visualizations of analysis results*

Compiler Scientist

•The first compiler was written by Grace Hopper, in 1952, for the A-0 programming language. [**COBOL**]

(1906 – 1992)

(1925-2003)

•The first autocode and its compiler were developed by Alick Glennie in 1952 for the **Mark 1** computer at the University of Manchester and is considered by some to be the first compiled programming language.

•The FORTRAN team led by John Backus at IBM is generally credited as having introduced the first complete compiler in 1957.
•BNF

1924 – 2007

•The first *self-hosting* compiler – capable of compiling its own source code in a high-level language – was created in 1962 for Lisp by **Tim Hart** and Mike Levin at MIT.

John McCarthy

Dennis Ritchie

1941 – 2011

Bjarne Stroustrup [1983]

James Arthur Gosling [1995]

http://en.wikipedia.org/wiki/List_of_compilers

Thank You!