CSE-361:Compiler Design

AMINA KHATUN

ASSOCIATE PROFESSOR

DEPT. OF CSE

JAHANGIRNAGAR UNIVERSITY

Textbook

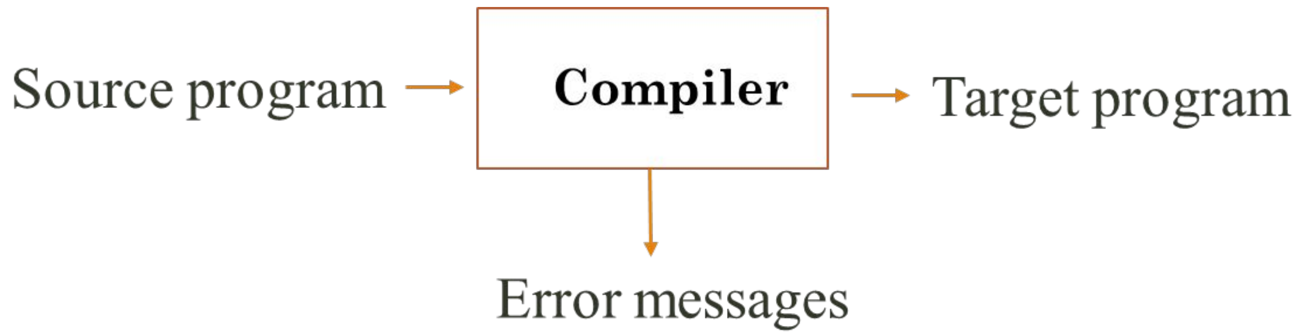
- Compilers: Principles, Techniques, and Tools
  - Aho, Lam, Sethi, Ullman
- Modern Compiler Implementation in C (The Tiger Book).
  - Andrew W. Appel

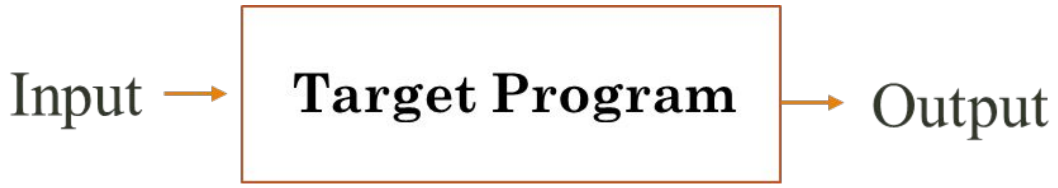
Language Processor

- ❖ A computer understands instructions in machine code, i.e. in the form of 0s and 1s. It is a tedious task to write a computer program directly in machine code.
- ❖ The programs are written mostly in high level languages like Java, C++, Python etc. and are called **source code**. These source code cannot be executed directly by the computer and must be converted into machine language to be executed.
- ❖ Hence, a special translator system software is used to translate the program written in high-level language into machine code is called **Language Processor** and the program after translated into machine code (object program / object code).
- ❖The language processors can be any of the following three types:
    1. Compiler
    2. Interpreter
    3. Assembler

Language Processor: Compiler

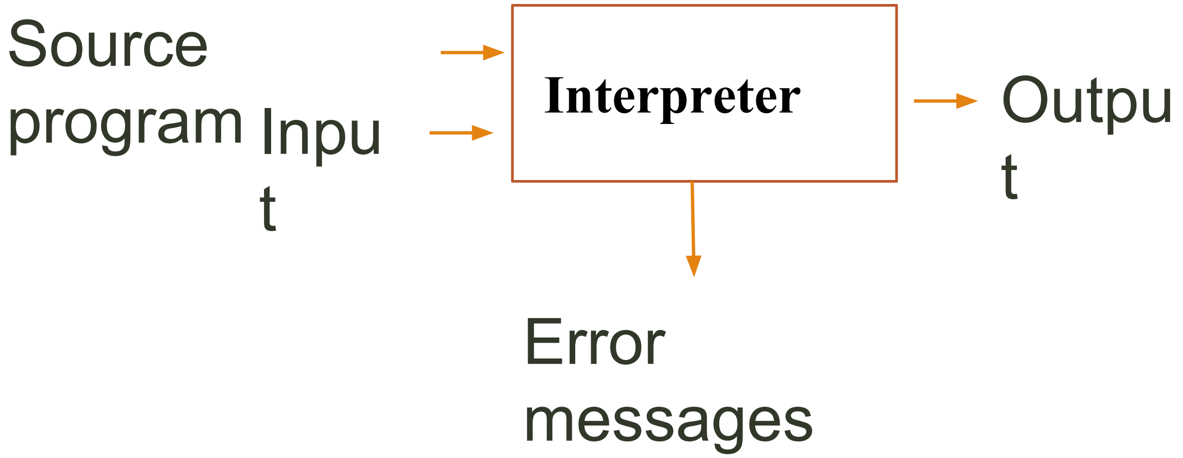
- A compiler is a program that reads the whole program written in a ***source language/high level language as a whole in one go*** and translates it into an equivalent program in a ***target language.***



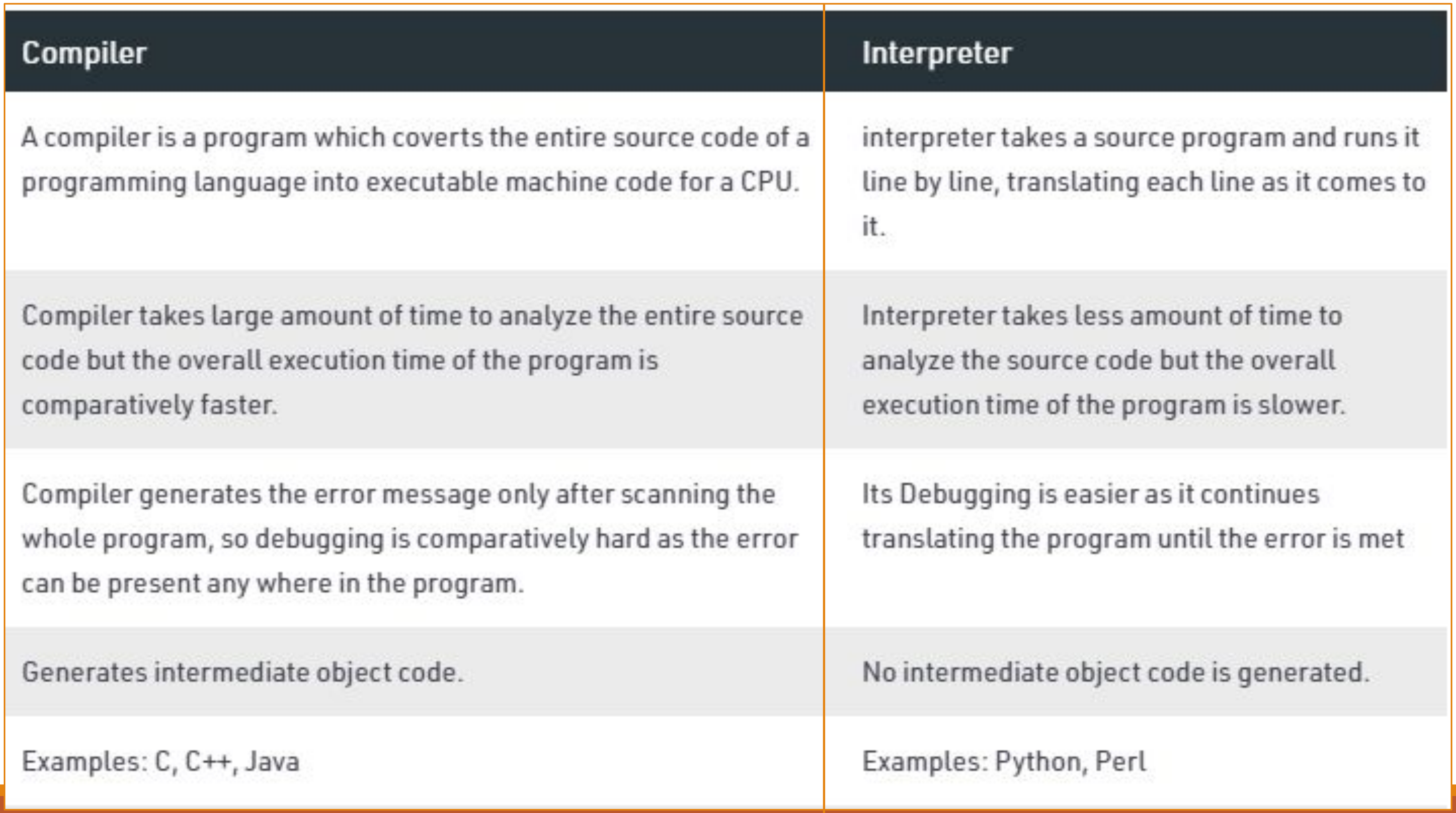


Language Processor: Interpreter

- An interpreter is another common kind of language processor.
- Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user.



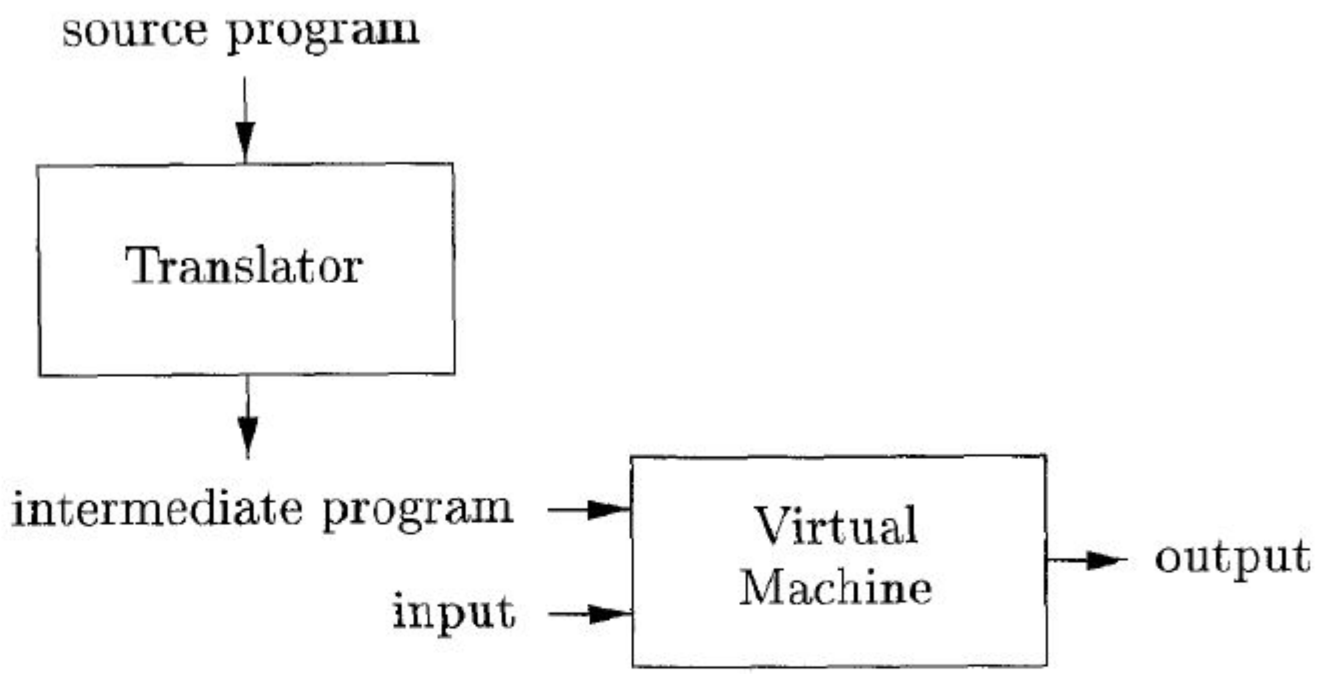
Compiler vs. Interpreter



Compiler	Interpreter
A compiler is a program which coverts the entire source code of a programming language into executable machine code for a CPU.	interpreter takes a source program and runs it line by line, translating each line as it comes to it.
Compiler takes large amount of time to analyze the entire source code but the overall execution time of the program is comparatively faster.	Interpreter takes less amount of time to analyze the source code but the overall execution time of the program is slower.
Compiler generates the error message only after scanning the whole program, so debugging is comparatively hard as the error can be present any where in the program.	Its Debugging is easier as it continues translating the program until the error is met
Generates intermediate object code.	No intermediate object code is generated.
Examples: C, C++, Java	Examples: Python, Perl

Hybrid Compiler

- Java language processors combine compilation and interpretation.

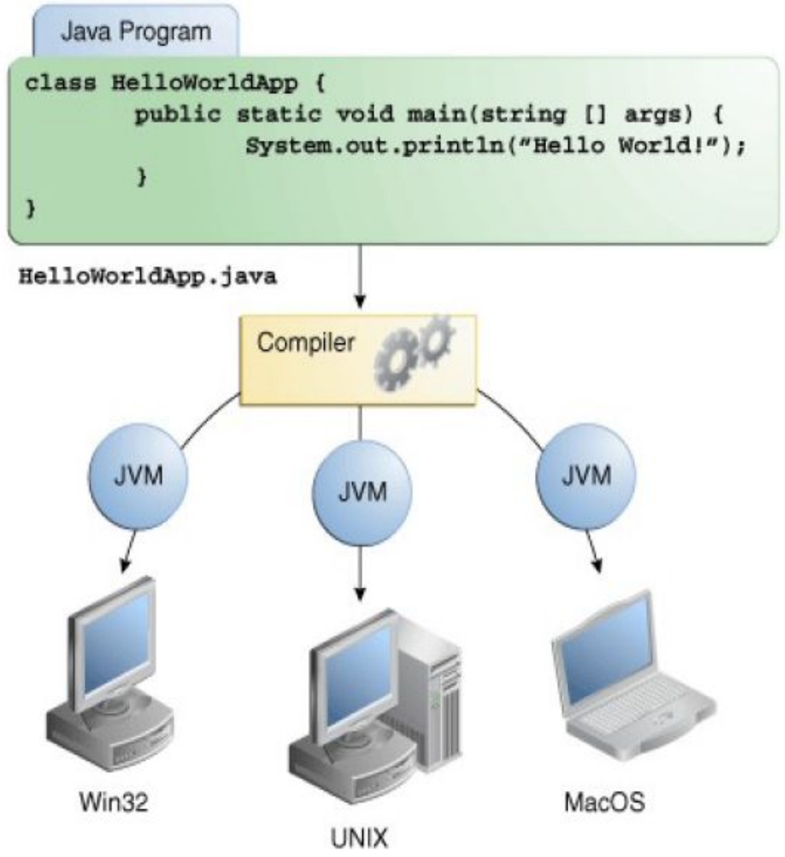


Hybrid Compiler

- A Java source program may first be compiled into an intermediate form called **bytecodes**.
- The **bytecodes** are then interpreted by a virtual machine.

- A benefit of this arrangement is that bytecodes compiled on one machine can be interpreted on another machine, perhaps across a network.

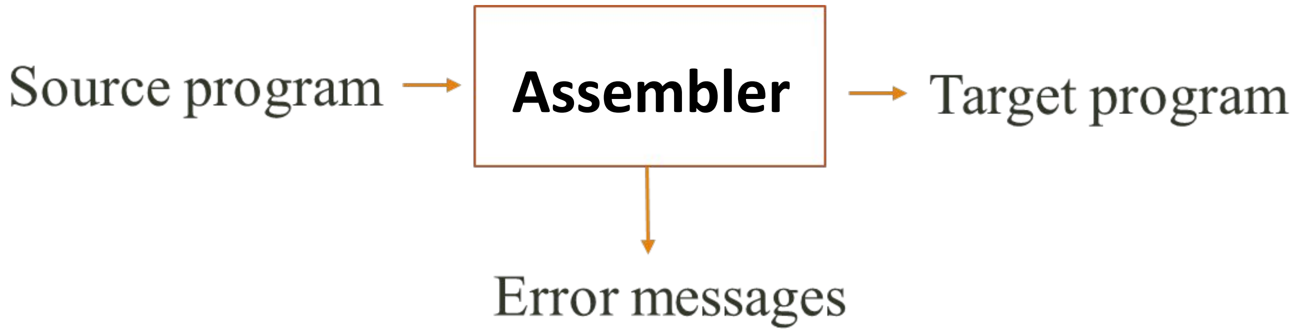
- In order to achieve faster processing of inputs to outputs, some Java compilers, called **just-in-time compilers,** translate the bytecodes into machine language immediately before they run the intermediate program to process the input.

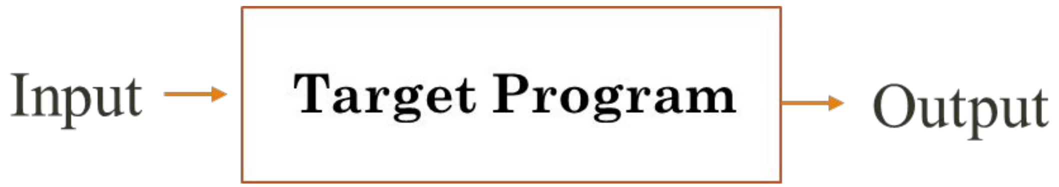


```
class HelloWorldApp {
        public static void main(string [] args) {
                System.out.println("Hello World!");
        }
}
```

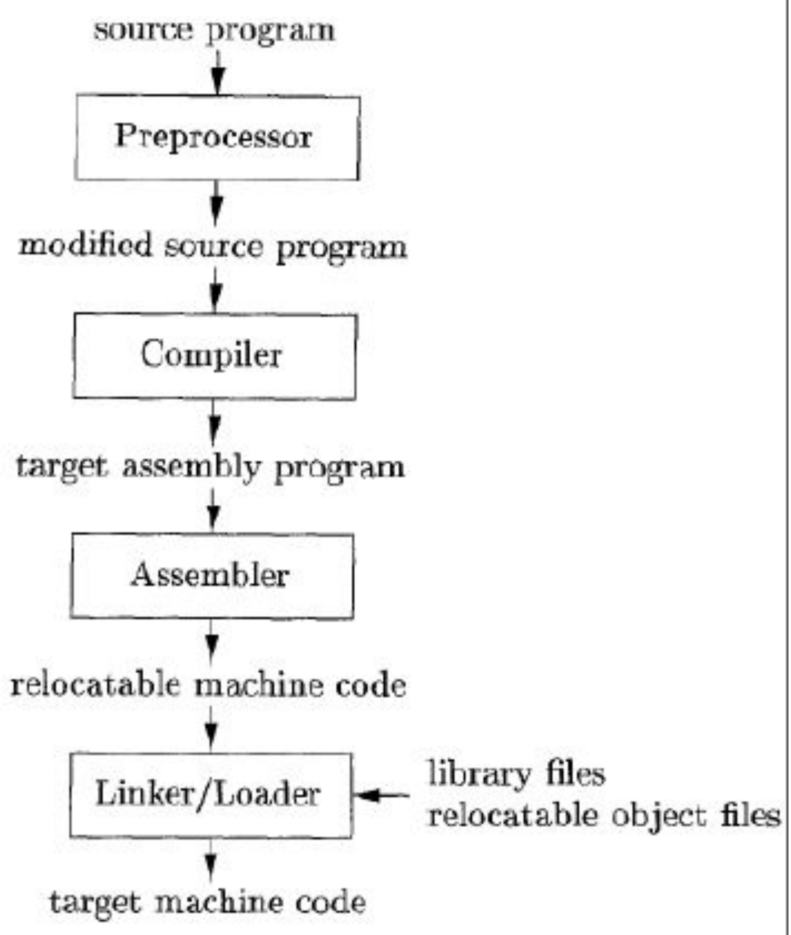
LANGUAGE PROCESSOR: ASSEMBLER

- The Assembler is used to translate the program written in Assembly language into machine code. Assembly language is machine dependent yet mnemonics that are being used to represent instructions in it are not directly understandable by machine and high level language is machine independent.





Other Language Processors:



A language-processing system

Compilation task is full of variety ??

- Thousands of source languages
  - Fortran, Pascal, C, C++, Java, ……
- Thousands of target languages
  - Some other lower level language (assembly language), machine language
- Compilation process has similar variety
  - Single pass, multi-pass, load-and-go, debugging, optimizing….
- Variety is overwhelming……

Good news is:

- Few basic techniques is sufficient to cover all variety
- Many efficient tools are available

Job of Compiler

- We will study compilers that take as input programs in a high-level programming language and give as output programs in a low-level assembly language.
- Such compilers have 3 jobs:
  - TRANSLATION
  - VALIDATION
  - OPTIMIZATION

Why Study Compiler?

- To be more effective users of compilers ... instead of treating a compiler as a "black box".
- To apply compiler techniques to typical SE tasks that require reading input and taking action.
- To see how your core CS courses fit together, giving you the knowledge to construct a compiler.
- To participate in R&D of high-level programming languages and optimizing compilers.

Challenges of Compiler Construction

Compiler construction poses challenging and interesting problems:

- Compilers must process large inputs, perform complex algorithms, but also run quickly
- Compilers have primary responsibility for run-time performance
- Compilers are responsible for making it acceptable to use the full power of the programming language
- Computer architects perpetually create new challenges for the compiler by building more complex machine
- Compilers must hide that complexity from the programmer

A successful compiler requires mastery of the many complex interactions between its constituent parts

Compiler and other areas

Compiler construction involves ideas from different parts of computer science

- **Artificial intelligence**: Greedy algorithms, Heuristic search techniques
- **Algorithms:** Graph algorithms, Dynamic programming
- **Theory of Automata**: DFAs & PDAs, pattern matching, regular expressions
- **Architecture:** Pipelining and Instruction set use

Requirement

- In order to translate statements in a language, one needs to understand both
  - **Structure of the language**: the way “sentences" are constructed in the language
  - **Meaning of the language**: what each “sentence" stands for.

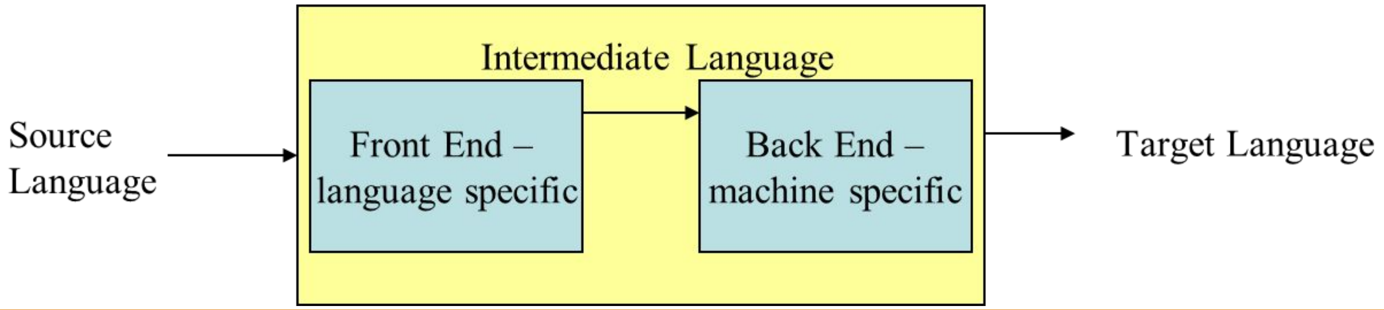
- Terminology:
  - Structure ≡ Syntax
  - Meaning ≡ Semantics

Analysis-Synthesis model of compilation

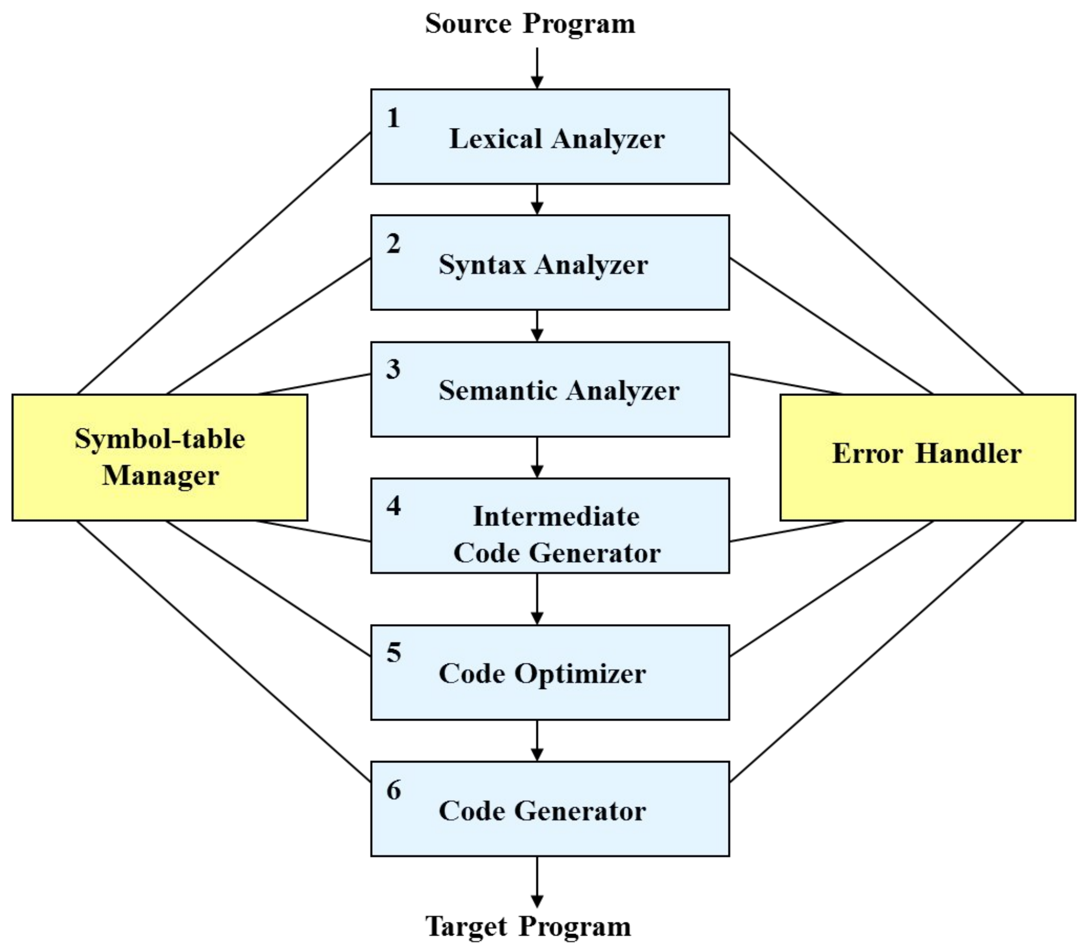
- Two major parts --
  - **Analysis**: an intermediate representation is created from the given source program.

    Lexical Analyzer, Syntax Analyzer and Semantic Analyzer
  - **Synthesis**: the equivalent target program is created from this intermediate representation

    Intermediate Code Generator, Code Optimizer, and Code Generator



Phases of Compiler



Compilation Steps/Phases

- **Lexical Analysis**: Generates the “tokens” in the source program
- **Syntax Analysis**: Recognizes “sentences" in the program using the syntax of the language
- **Semantic Analysis**: Infers information about the program using the semantics of the language
- **Intermediate Code Generation**: Generates “abstract” code based on the syntactic structure of the program and the semantic information
- **Optimization**: Refines the generated code using a series of optimizing transformations
- **Final Code Generation**: Translates the abstract intermediate code into specific machine instructions

Lexical Analysis

- Convert the stream of characters representing input program into a meaningful sequences called lexemes.
- For each lexeme, the lexical analyzer produces as output

A token of the form:

< **token-name, attribute-value** >

token-name □ an abstract symbol that is used during syntax analysis

attribute-value□ points to an entry in the symbol table for this token

- **Example**:

*Input: "*x++"* *Output: three tokens □ "*", "x", "++"*

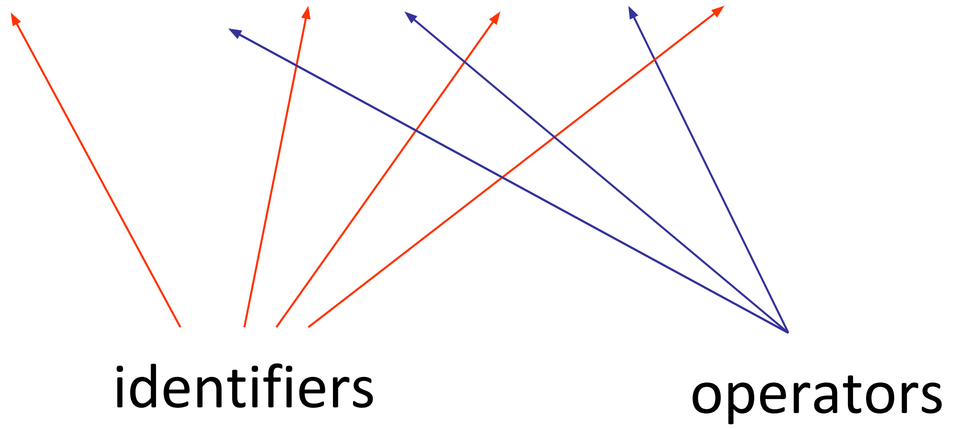
Input: "static int" *Output: two tokens: □ "static" , "int"*

- Removes the white spaces, comments

Lexical Analysis

- Input: result = a + b * 10
- Tokens :

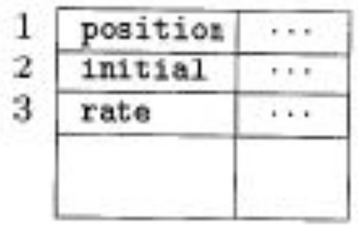
‘result’, ‘=‘, ‘a’, ‘+’, ‘b’, ‘*’, ‘10’



Lexical Analysis

- Input: `position = initial + rate * 60`
- Output: Sequence of tokens

$$\langle \mathbf{id}, 1\rangle\ \langle = \rangle\ \langle \mathbf{id}, 2\rangle\ \langle + \rangle\ \langle \mathbf{id}, 3\rangle\ \langle * \rangle \langle 60\rangle$$



1	position	...
2	initial	...
3	rate	...

SYMBOL TABLE

- In this representation, the token names =, +, and * are abstract symbols for the **assignment**, **addition**, and **multiplication** operators, respectively.

Syntax Analysis (Parsing)

- Build a tree called a parse tree that reflects the structure of the input sentence.
- A syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation.

Example:

- The Phrase : x = +y
- Four Tokens “x", “=“ ,“+" and “y“
- Structure x = (x+(y)) i.e., an assignment expression

Syntax Analysis: Grammars

- Expression grammar

Exp $\rightarrow$ Exp ‘+’ Exp

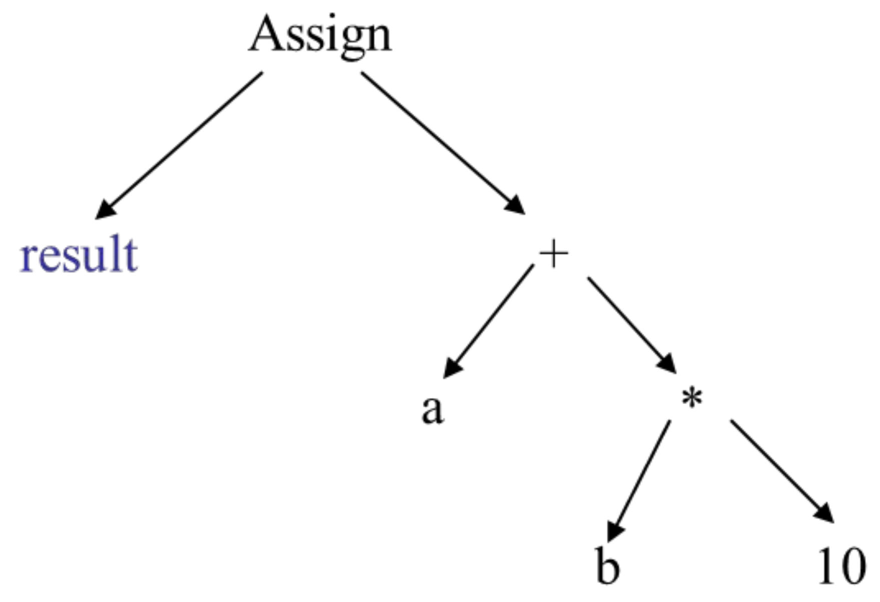
| Exp ‘*’ Exp

| ID

| NUMBER

Syntax Analysis: Syntax Tree

- **Input**: result = a + b * 10



Semantic Analysis

- Check the source program for semantic errors
- It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operands of expressions and statements
- Performs type checking
  - Operator operand compatibility

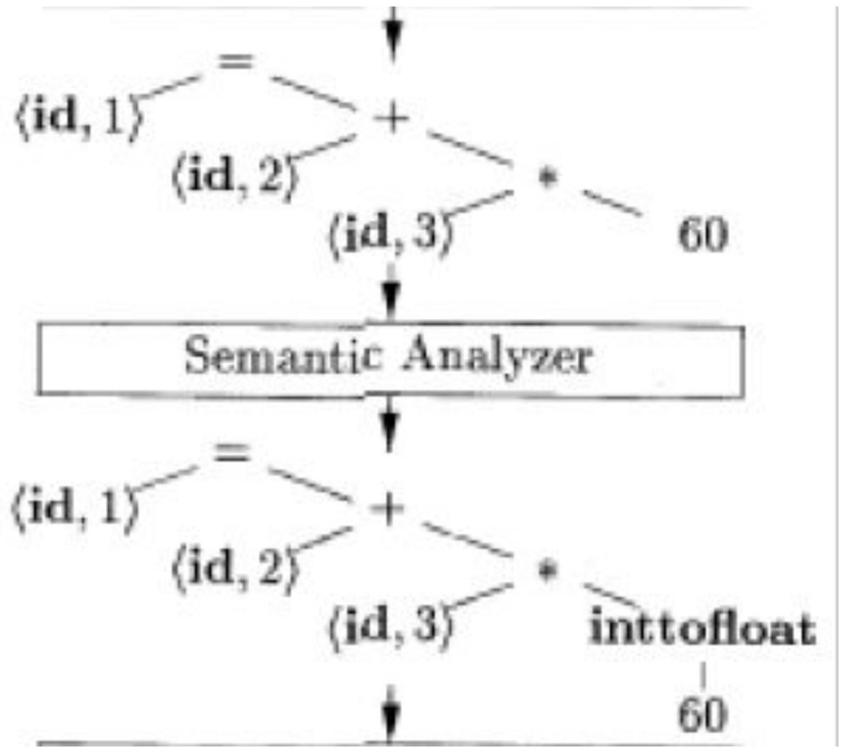
Example:

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**.
- **Example:**

  The compiler may **convert or coerce** the integer into a floating-point number.

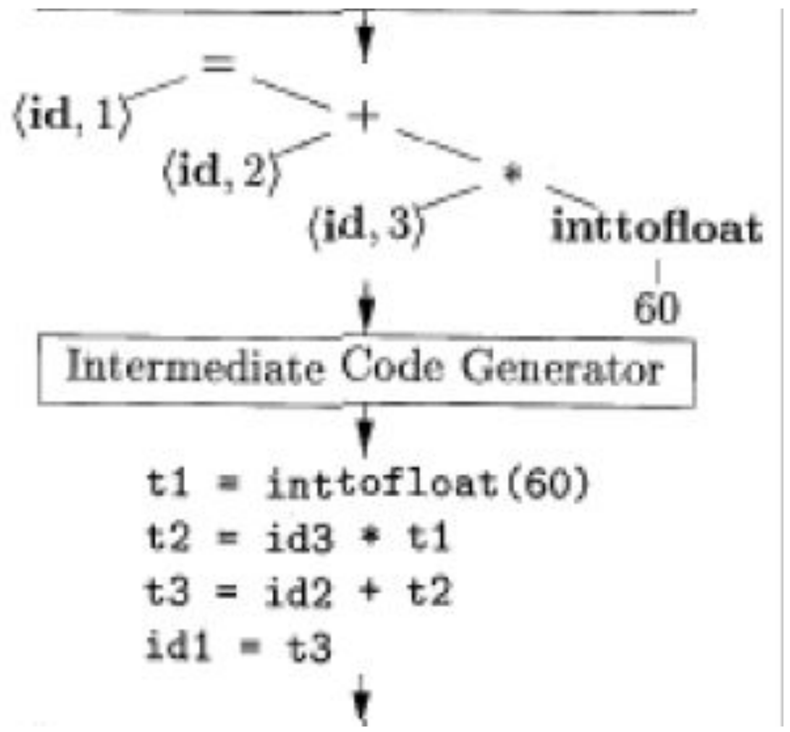


Intermediate Code Generation

- Translate each hierarchical structure decorated as tree into intermediate code
- A program translated for an abstract machine
- Properties of intermediate codes
  - Should be easy to produce
  - Should be easy to translate into the target program
- Intermediate code hides many machine-level details, but has instruction-level mapping to many assembly languages
- Main motivation: **portability**
- One commonly used form is **“Three-address Code”**

Intermediate Code Generation

- We consider an intermediate form called “three-address code”.
- Like the assembly language for a machine in which every memory location can act like a register.
- **Three-address code** consists of a sequence of instructions, each of which has at most three operands.

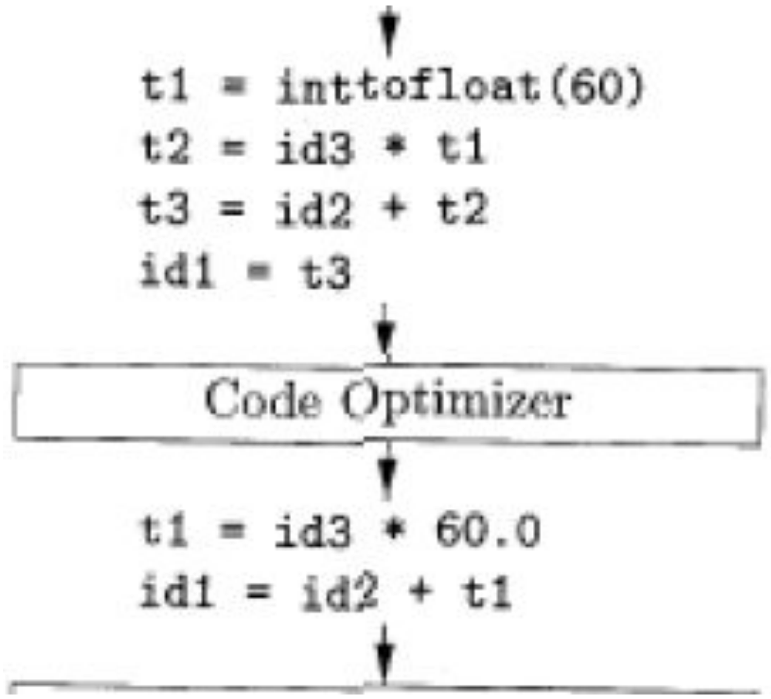


```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Code Optimization

- Apply a series of transformations **to improve the time and space efficiency of the generated code.**
- Peephole optimizations: generate new instructions by combining/expanding on a small number of consecutive instructions.
- Global optimizations: reorder, remove or add instructions to change the structure of generated code
- Consumes a significant fraction of the compilation time
- Optimization capability varies widely
- Simple optimization techniques can be vary valuable

Code Optimization



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

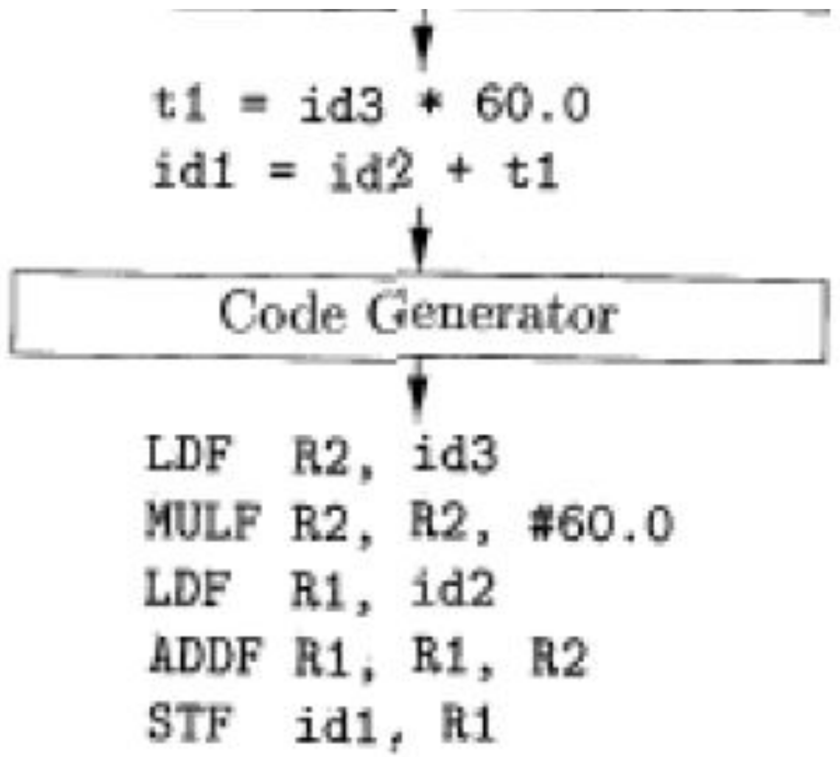
```
t1 = id3 * 60.0
id1 = id2 + t1
```

Code Generation

- Map instructions in the intermediate code to specific machine instructions.
- Memory management, register allocation, instruction selection, instruction scheduling, …
- Generates sufficient information to enable symbolic debugging.

Code Generation

For example, using registers R1 and R2, the intermediate code might get translated into the machine code



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

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

Symbol Table

- Records the identifiers used in the source program
  - Collect information about various attributes of each identifier
    - **Variables**: type, scope, storage allocation
    - **Procedure**: number and types of arguments, method of argument passing
- **It’s a data structure containing a record for each identifier**
  - Different fields are collected and used at different phases of compilation
- When an identifier in the source program is detected by the lexical analyzer, the identifier is entered into the symbol table

Symbol Table

- It is built in lexical and syntax analysis phases and It is used by compiler to achieve compile time efficiency.
- The information is collected by the analysis phases of compiler and is used by synthesis phases of compiler to generate code.

Items stored in Symbol table:

- Variable names and constants
- Procedure and function names
- Literal constants and strings
- Compiler generated temporaries
- Labels in source languages

Symbol Table

Information used by compiler from Symbol table:

- Data type and name
- Declaring procedures
- Offset in storage
- If structure or record then, pointer to structure table.
- For parameters, whether parameter passing by value or by reference
- Number and type of arguments passed to function
- Base Address

Symbol Table

It is used by various phases of compiler as follows :-

- **Lexical Analysis**: Creates new table entries in the table, example like entries about token.
- **Syntax Analysis**: Adds information regarding attribute type, scope, dimension, line of reference, use, etc in the table.
- **Semantic Analysis**: Uses available information in the table to check for semantics i.e. to verify that expressions and assignments are semantically correct(type checking) and update it accordingly.
- **Intermediate Code generation**: Refers symbol table for knowing how much and what type of run-time is allocated and table helps in adding temporary variable information.
- **Code Optimization**: Uses information present in symbol table for machine dependent optimization.
- **Target Code generation**: Generates code by using address information of identifier present in the table.

Error Detection, Recovery and Reporting

- Each phase can encounter error.
- The tasks of the **Error Handling** process are to detect each error, report it to the user, and then make some recover strategy and implement them to handle error.
- Specific types of error can be detected by specific phases
  - **Lexical Error** : *int abc, 1num ;*
  - **Syntax Error**: *total = capital + rate   year;*
  - **Semantic Error**: *value = myarray [realIndex];*
- Should be able to proceed and process the rest of the program after an error detected
- Should be able to link the error with the source program

Error Detection, Recovery and Reporting

Types or Sources of Error –

There are two types of error: Run-time and Compile-time error:

- **A Run-time error** is an error which takes place during the execution of a program, and usually happens because of adverse system parameters or invalid input data.
  - The lack of sufficient memory to run an application or a memory conflict with another program
  - Logical error is another example. Logic errors, occur when executed code does not produce the expected result. Logic errors are best handled by meticulous program debugging.

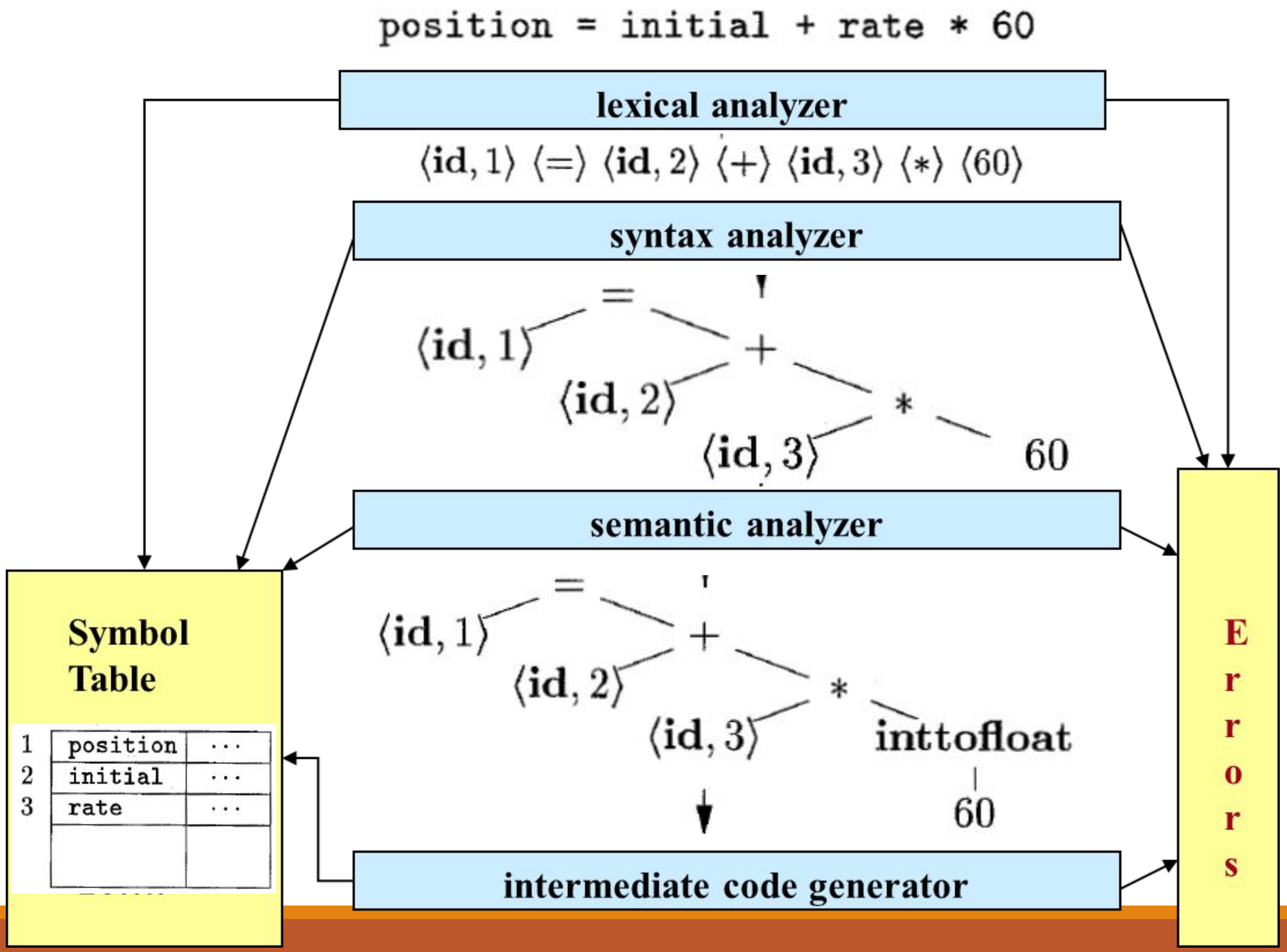
Error Detection, Recovery and Reporting

Types or Sources of Error –

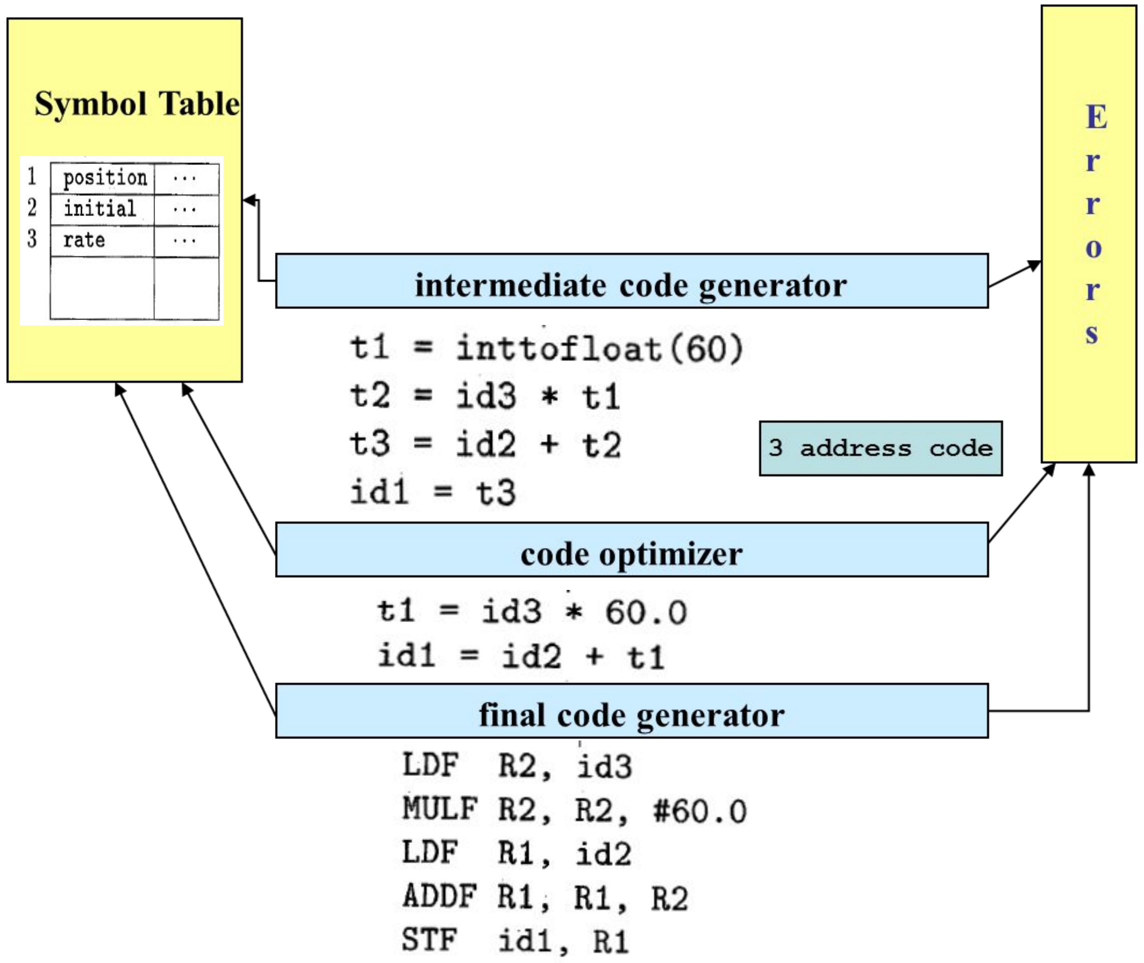
There are two types of error: Run-time and Compile-time error:

- **A Compile-time error** rises at compile time, before execution of the program. Syntax error or missing file reference that prevents the program from successfully compiling is the example of this.
  - Classification of Compile-time error –
    - **Lexical** : This includes misspellings of identifiers, keywords or operators
    - **Syntactical** : missing semicolon or unbalanced parenthesis
    - **Semantical** : incompatible value assignment or type mismatches between operator and operand
    - **Logical** : code not reachable, infinite loop.

Reviewing the Entire Process



Reviewing the Entire Process



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

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

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

Compiler Construction Tools

1) **Parser generators:** It produces syntax analyzers (parsers) from the input that is based on a grammatical description of programming language or on a context-free grammar.
2) **Scanner generators:** It generates lexical analyzers from the input that consists of regular expression description based on tokens of a language. It generates a finite automaton to recognize the regular expression.
3) **Syntax-directed translation engines:** It generates intermediate code with three address format from the input that consists of a parse tree. These engines have routines to traverse the parse tree and then produces the intermediate code.

Compiler Construction Tools

4) **Code-generator:** It generates the machine language for a target machine.
5) **Data-flow analysis engines:** A key part of the code optimization that gathers the information, that is the values that flow from one part of a program to another.
6) **Compiler-construction toolkits:** It provides an integrated set of routines that aids in building compiler components or in the construction of various phases of compiler.

Reading Materials

- Chapter -1 of your Text book:
  - Compilers: Principles, Techniques, and Tools
- https://www.geeksforgeeks.org/compiler-design-tutorials/

THE END