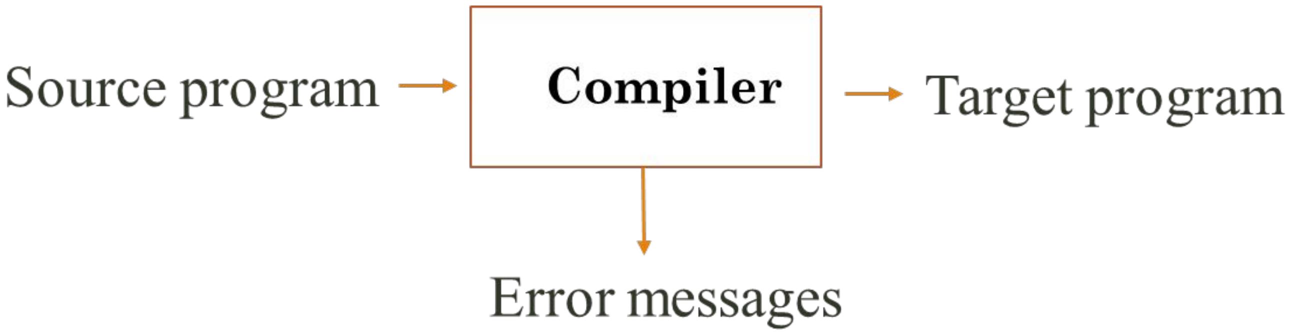
CSE-331:Compiler Design

Textbook

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

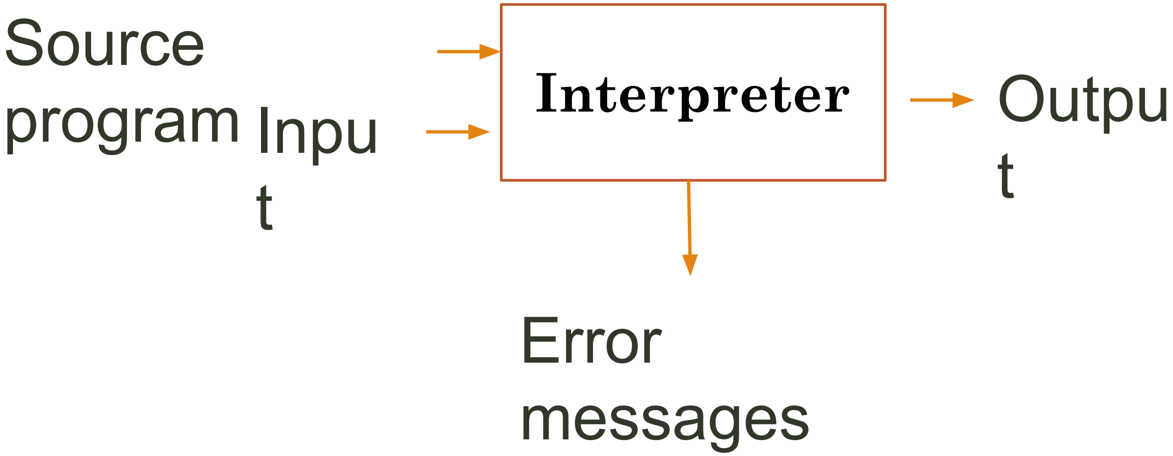
Language Processor: Compiler

- A compiler is a program takes a program written in a ***source language*** 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

- The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs.

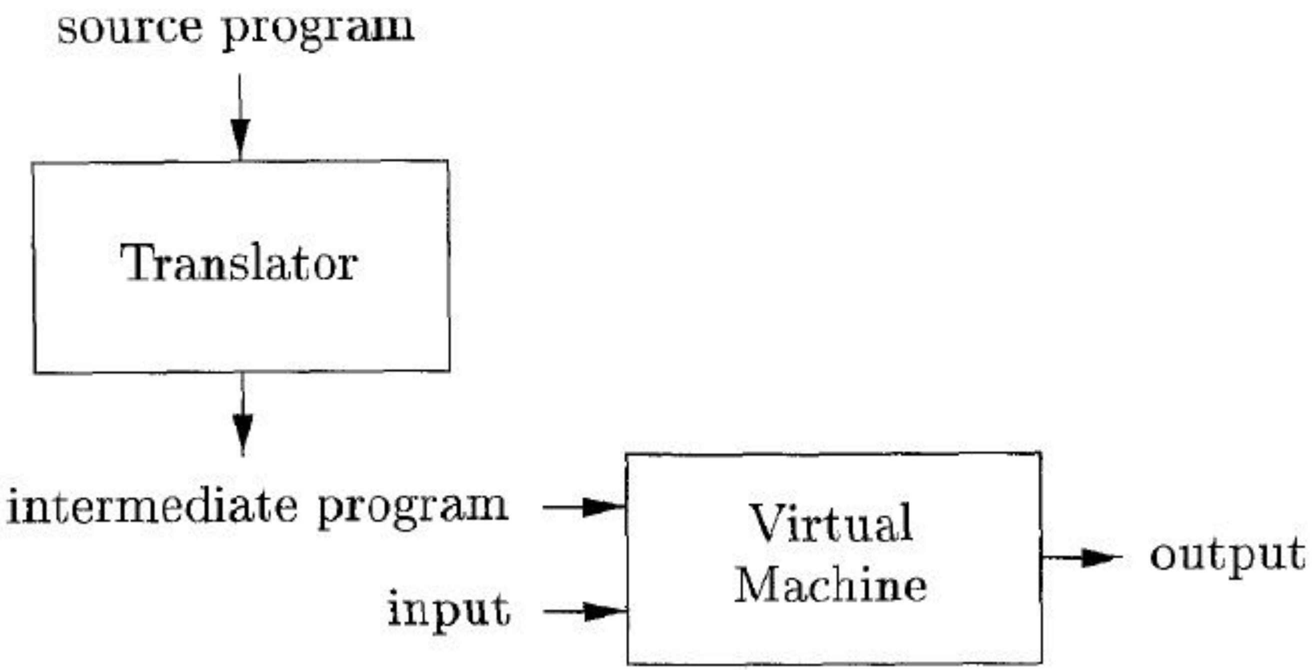
  Example: Programming language like C, C++ use compilers.

- An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

  Example: Programming language like Python, Ruby use interpreters.

Hybrid Compiler

- Java language processors combine compilation and interpretation.

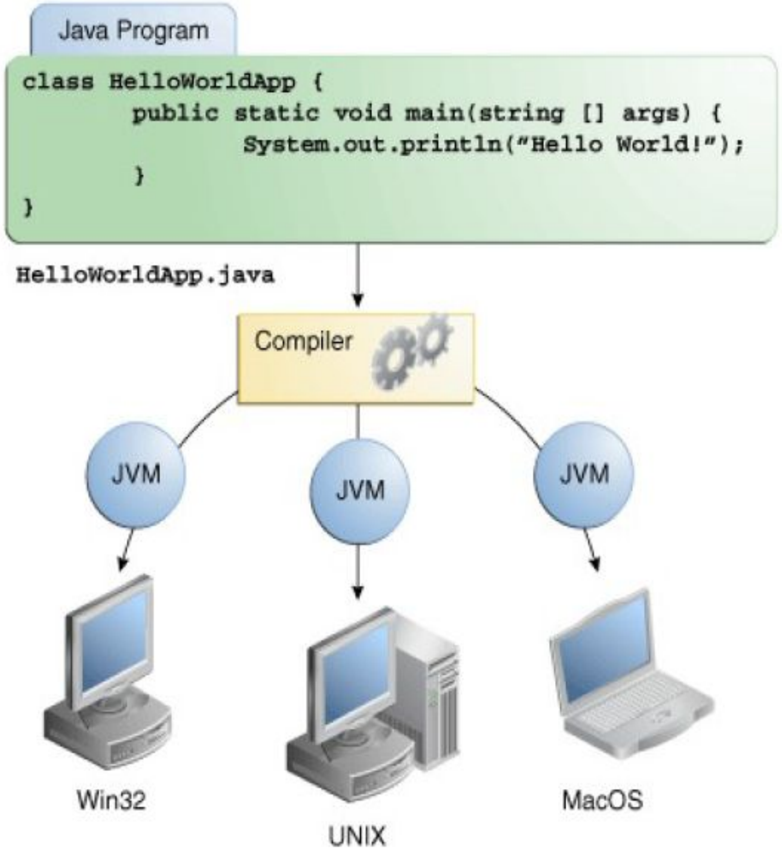


Hybrid Compiler

- 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.



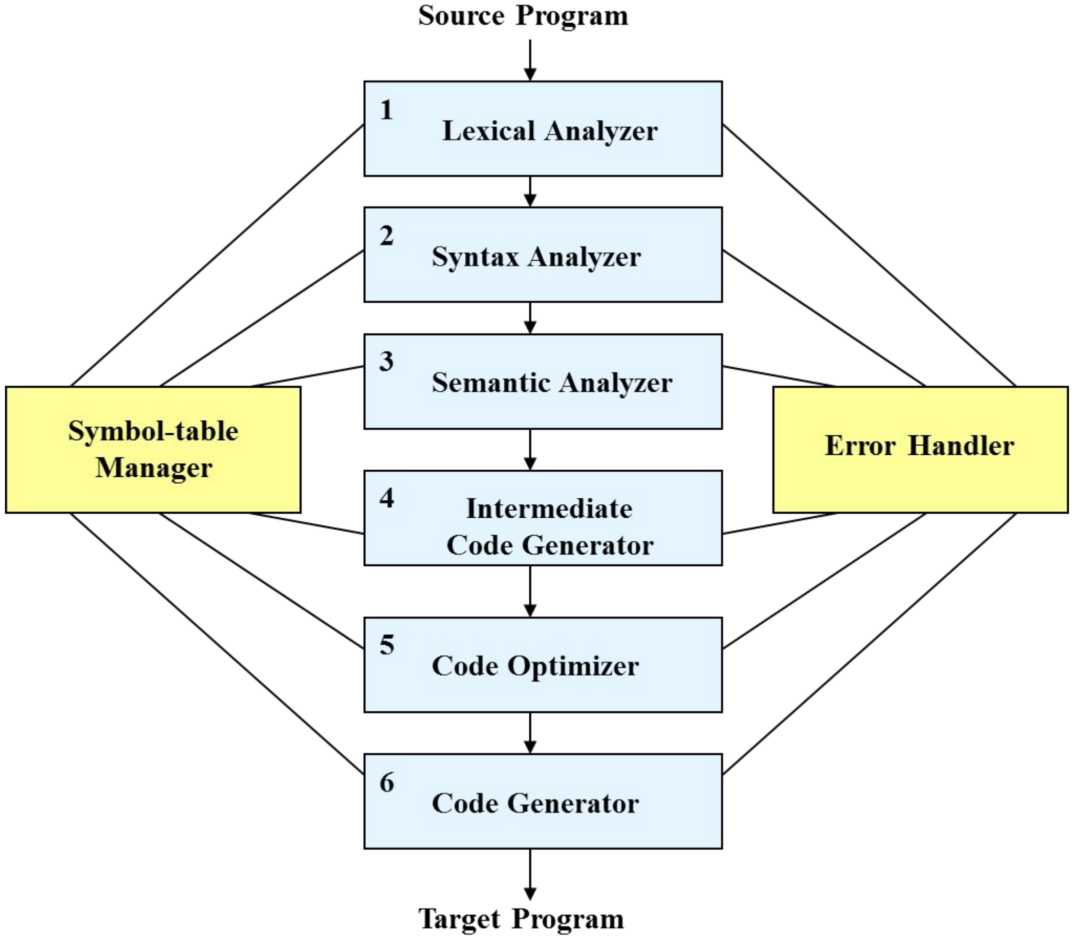
Job of Compiler

- 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

Applications of Compiler Technology

- Implementation of High-Level Programming Languages.
- Optimizations for Computer Architectures.
- Design of New Computer Architectures.
- Program Translations.
- Software Productivity Tools.

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", "++"*

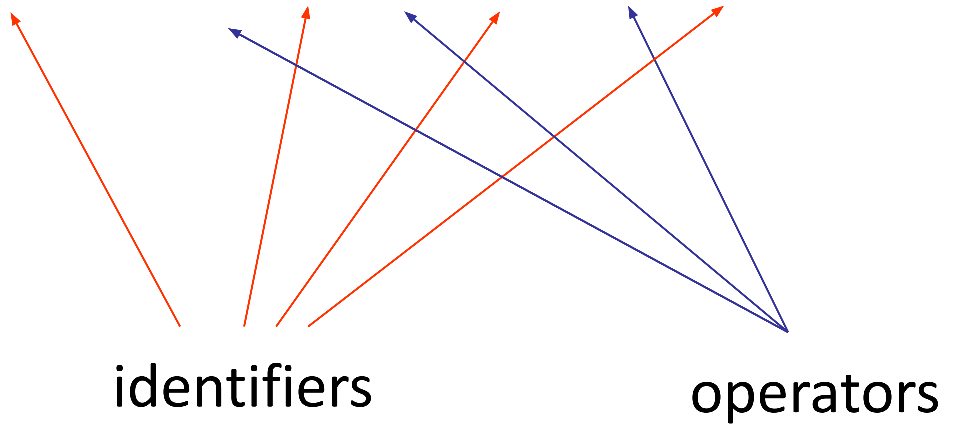
Input: "static int" *Output: two tokens: □ "static" , "int"*

- 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

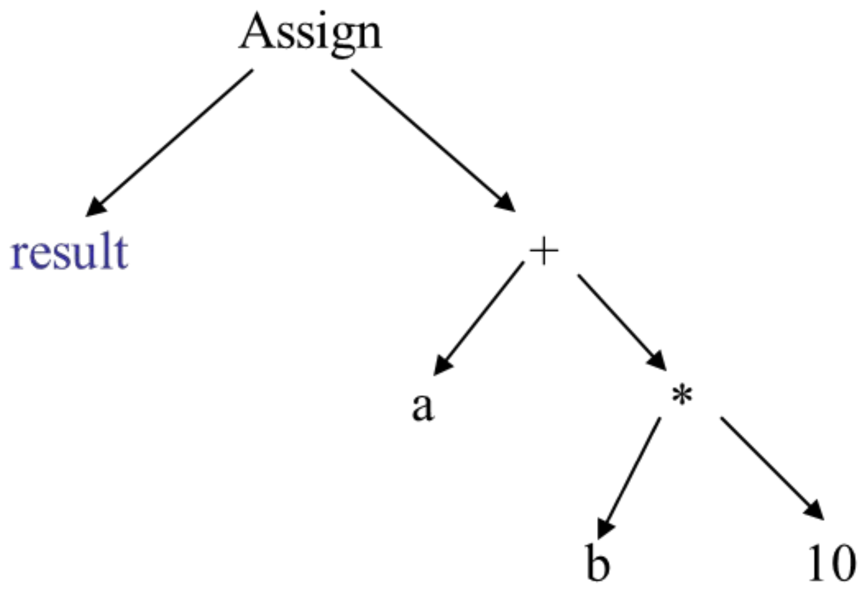
Syntax Analysis: Grammars

- Expression grammar

$$
\begin{aligned}
\text{Exp} \;\rightarrow\;& \text{Exp '+' Exp} \\
|\;& \text{Exp '*' Exp} \\
|\;& \text{ID} \\
|\;& \text{NUMBER}
\end{aligned}
$$

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

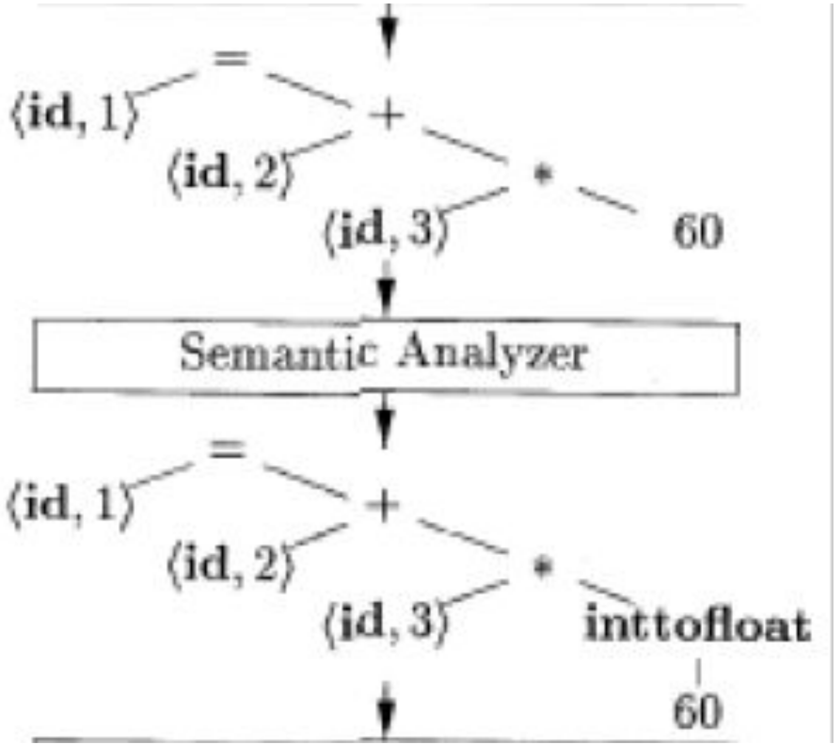
Example:

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.

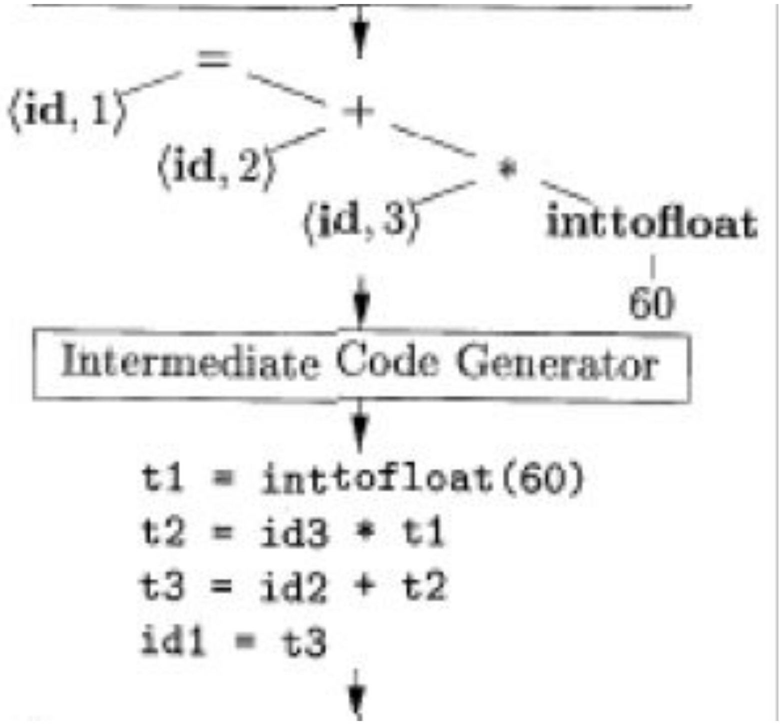


Intermediate Code Generation

- 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.



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
```

Code Optimizer

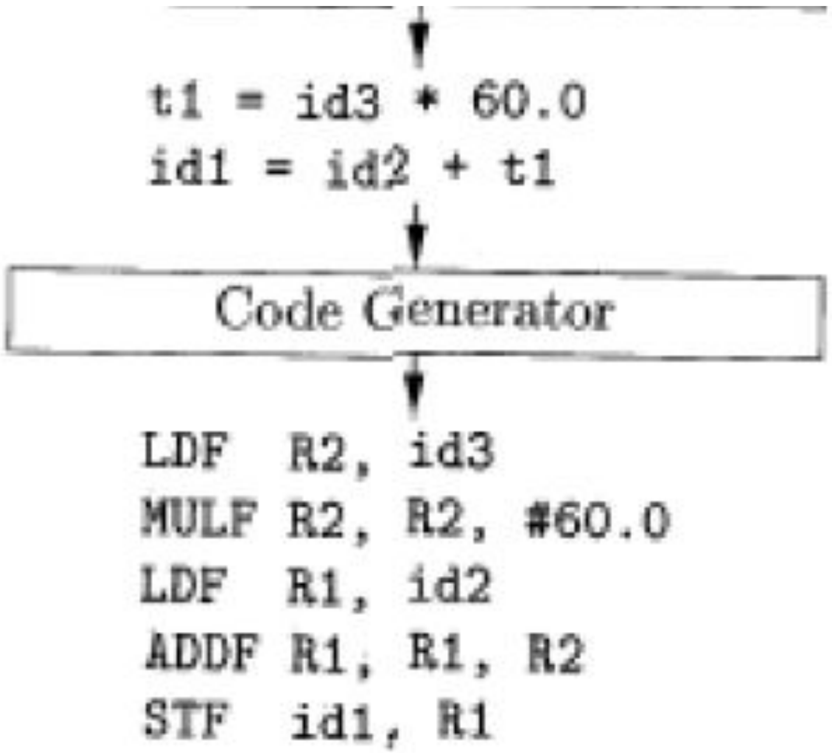
```
t1 = id3 * 60.0
id1 = id2 + t1
```

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



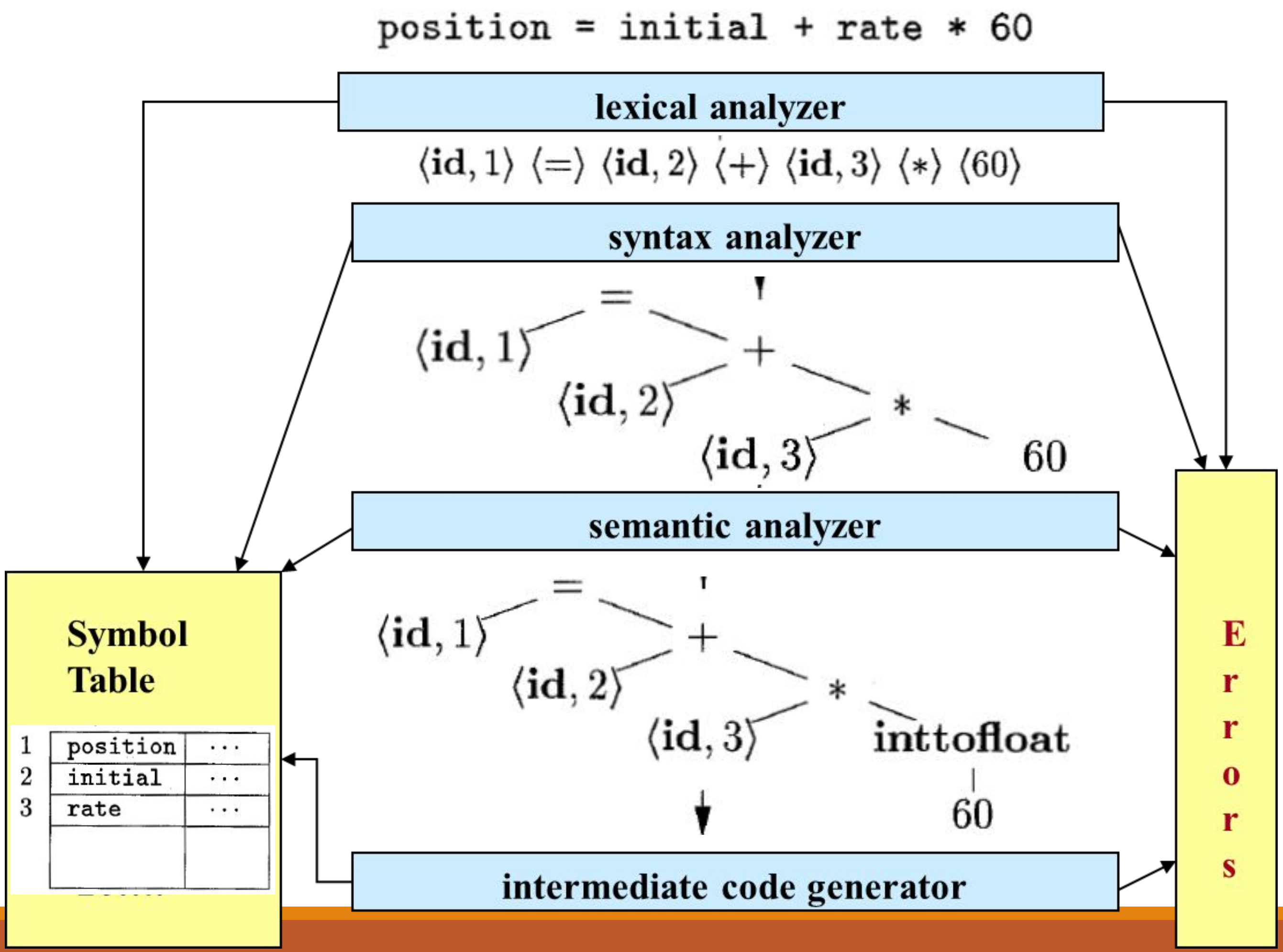
Symbol Table

- 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

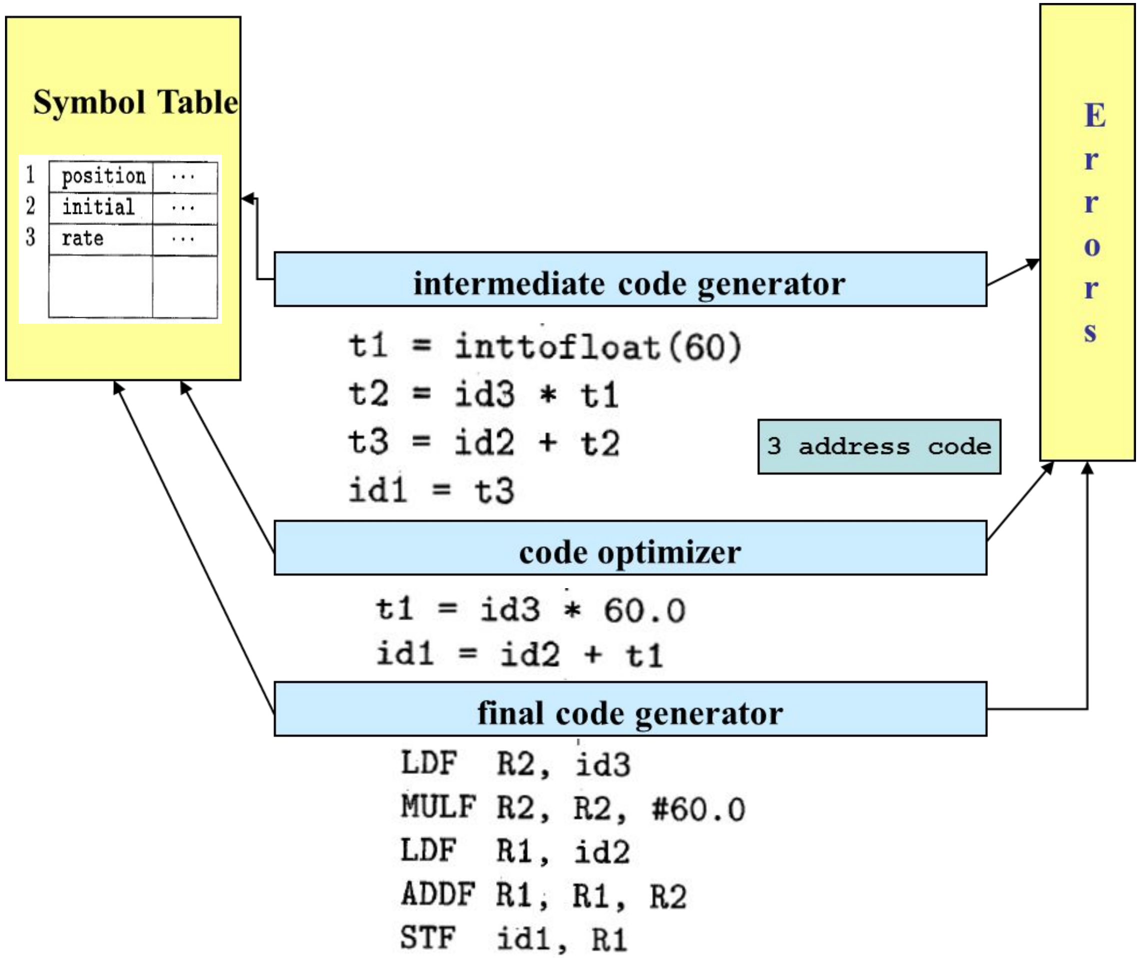
Error Detection, Recovery and Reporting

- Each phase can encounter 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

Reviewing the Entire Process



Reviewing the Entire Process



Compiler Construction Tools

1) Parser generators
2) Scanner generators
3) Syntax-directed translation engines
4) Code-generator
5) Data-flow analysis engines
6) Compiler-construction toolkits

Error handler and symbol table is connected with all the phases of compiler. Why it is needed?

- Symbol table is used to store all the information about identifiers used in the program.
- It is a data structure containing a record for each identifier, with fields for the attributes of the identifier. It allows finding the record for each identifier quickly and to store or retrieve data from that record.
- Whenever an identifier is detected in any of the phases, it is stored in the symbol table.
- Each phase can encounter errors. After detecting an error, a phase must some how deal with the error, so that compilation can proceed.

Exercise

#!/usr/bin/python

var1 = ‘Hello World’

var2 = ‘Compiler Design’

print(“var1[0]: ”, var1[0])

print(“var2[1:5]: ”, var2[1:5])

Compilation of the above code produces lexemes at its first step. Show the all possible set of lexemes can be produced.