CS419 Compiler Construction

Lecture 2

Edited by Dr. Ismail Hababeh

Adapted from slides by Dr. Mohammad Daoud

German Jordanian University

Originally from slides by Dr. Robert A. Engelen

Programming Language Basics

- Static and dynamic decision policy
- Environments and states
- Static scope and block structure
- Parameter passing mechanisms

Static and Dynamic Decision

- **Static policy:** the language rules allow the compiler to decide during *compile time*
- **Dynamic policy:** the language rules allow the compiler to take a decision when the program is executed at *run time.*

Environments and States

- **Environment:** mapping from names to locations in the memory.

Example: maps x to a memory location that changes x.

- **State:** mapping from location in the memory to their values
- Example: y = x + 1;

return the value stored in memory location.

names

environment

locations (variables)

state

values

Environments and states - Example

```
….
int i;                    // global i
 …
void f(…){
      int i; //local i
      …
      i = 3; //use of local i
      …
}
…
      x = i +1;  //use of global i
```

Implicit declaration

Static Scope and Block Structure

Static scope: the scope of a declaration can be determined:

- *Implicitly:* by the location of the declaration inside the program.
- *Explicitly:* using language keywords such as public, private, protected (in C++, Java,…)

Explicit Declaration

- *Public*: accessible from outside the class
- *Protected*: scope is limited to the declaring class and subclasses (inheritance).
- *Private*: scope is limited to the declaring class and friend* classes.

*A **friend class** in C++ can access the "private" and "protected" members of the class in which it is declared as a friend.

Static Scope and Block Structure

- **Block:** a sequence of declarations followed by a sequence of statements, all surrounded by braces { }
- Blocks can be nested inside each other.

Question: Given the following a and b variables' declarations structure, determine the scope block(s) of each declaration.

```
main(){
    int a = 1;                              Block 1
    int b = 1;
    {
        int b = 2;                          Block 2
        {
            int a = 3;                      Block 3
            cout << a << b;
        }
        {
            int b = 4;                      Block 4
            cout << a  <<  b;
        }
        cout << a << b;
    }
    cout << a << b;
}
```

Question Solution

Declaration	Scope
int a = 1;	Block 1 – Block 3
int b = 1;	Block 1 – Block 2
int b = 2;	Block 2 – Block 4
int a = 3;	Block 3
int b = 4;	Block 4

Parameter Passing Mechanisms

- Call-by-value
  - The **value** of the *actual* parameter in the calling function is copied to the *formal* parameter in the called procedure.
  - All computations involving the formal parameters done by the called procedure is local to that procedure.

Call-by-value - Example

```
…..
int a =3;
int b = 4;
Function_1 (a , b);

…
Void Function_1(paramter_1, parameter_2)
{
            …..
}
```

Parameter Passing Mechanisms

- Call-by-reference
  - The *address of the actual parameter* in the calling function is passed to the called procedure as the *value of the formal parameter.*
  - Changes to the formal parameter appear in the called procedure as changes to the actual parameter in the calling function
  - Used to reduce the memory requirements of passing large arrays and objects by passing their addresses only.

Parameter Passing Mechanisms

- Call-by-pointer
  - Call by pointer do the same thing as call-by-reference. The only difference between them is the fact that a *pointer can be null*, or maybe *pointing to invalid places* in memory, while references are never be null.

Question: What will be the value of x after executing the following program?

```
// by value
void by_value(int a){
        a+=10;
}
// by pointer
void by_pointer(int *a){
        (*a)+=10;
}
// by reference
void by_ref(int &a){
        a+=10;
}
```

```
int main(){
        int x=40;
        by_value(x);
        //x = ?
        by_pointer(&x);
        //x = ?
        by_ref(x);
        //x = ?
        return 0;
}
```

Question Solution

```
void by_value(int a){
        a+=10;
}

void by_pointer(int *a){
        (*a)+=10;
}

void by_ref(int &a){
        a+=10;
}
```

Solution:

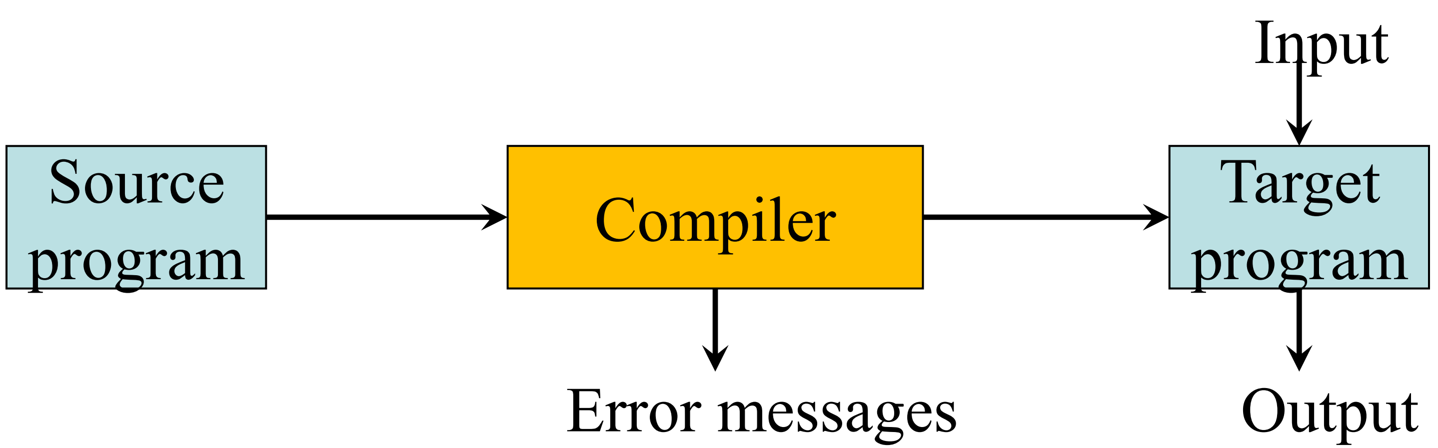
```
int main(){
        int x=40;
        by_value(x);
        //x=40
        by_pointer(&x);
        //x=50
        by_ref(x);
        //x=60
        return 0;
}
```

Compilers and Interpreters

- Both compilers and interpreters are language processors
- The target program produced by a compiler is faster than an interpreter
- An interpreter provides better error diagnostics than a compiler (executes the source code)

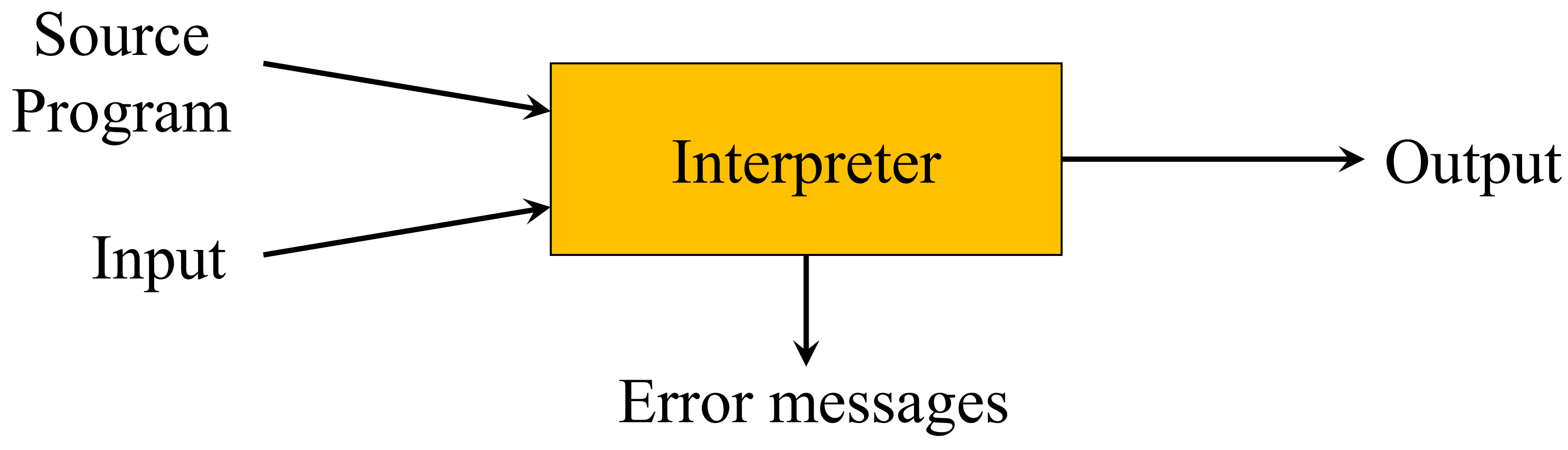
Compilers

- **Compiler**
  - A software (program) that translates a program in the *source* language into a semantically equivalent program in the *target* language (machine language)
  - Reports the source program errors that are detected during translation



Interpreters

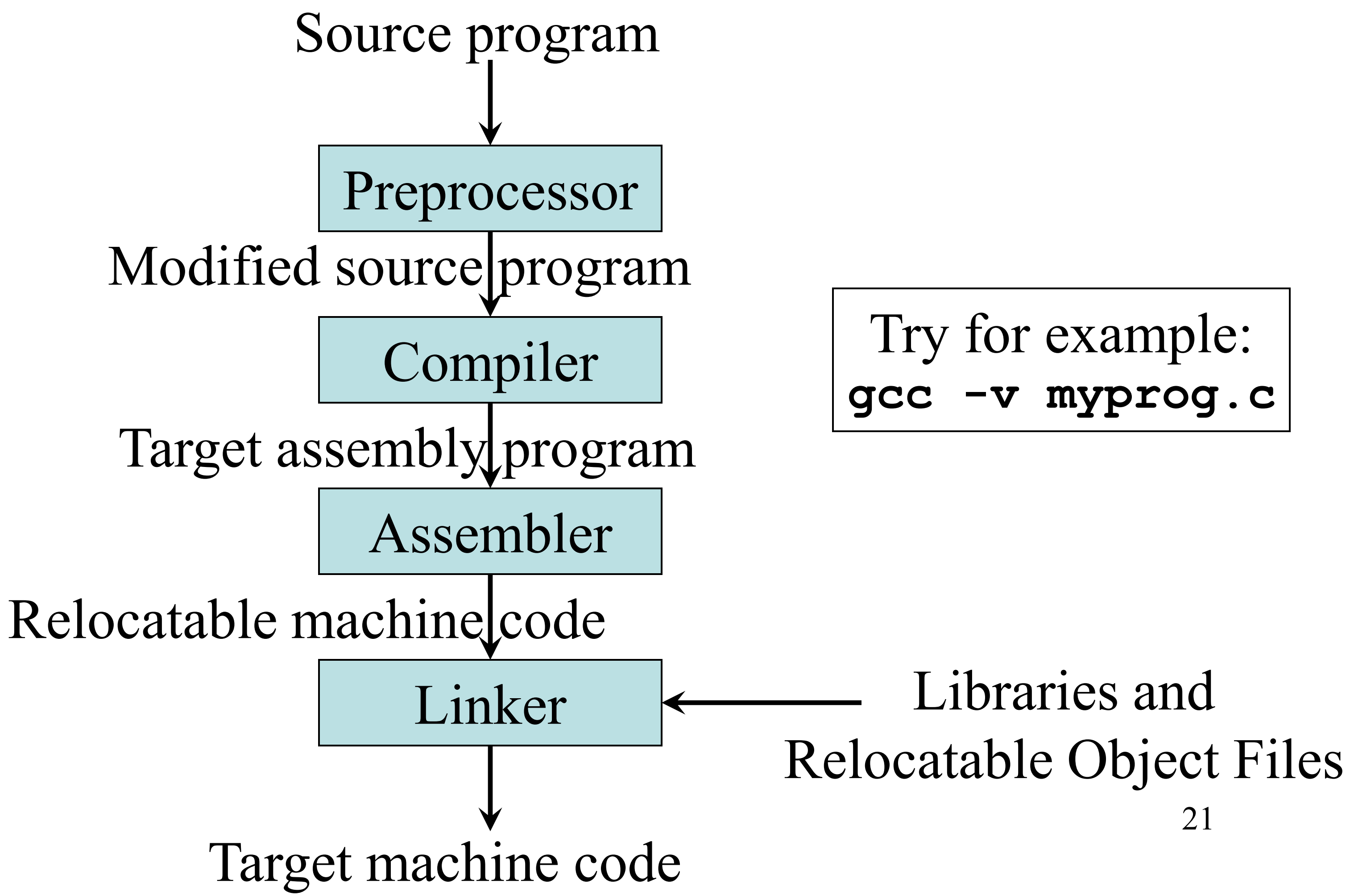
- Interpreter
  - Directly performs the operations implied by the source program without producing a target program



Preprocessors, Compilers, Assemblers, and Linkers

- If the source program is divided into modules stored in separate files, the **preprocessor** collects the source program
- The **compiler** translates the source program to produce an assembly-language program
- The assembly-language program processed by an **assembler** to produce relocatable machine code
- The **linker** links the relocatable machine code with other object files and library files to produce a code that runs on the machine

Preprocessors, Compilers, Assemblers, and Linkers



Try for example:
`gcc -v myprog.c`

The Analysis-Synthesis Model of Compilation

- There are two main parts of compilation:
  - *Analysis* breaks up the source program into basic pieces with grammatical structure. This structure is then represented as a tree.
  - *Synthesis* takes the tree structure and translates its operations into the target program

The Phases of a Compiler

- The compilation process operates as a sequence of phases
- Each compilation phase transforms one form of the source program to another representation form

The Phases of a Compiler

Phase	**Output**	**Example**
Programming	Source string	**`A=B+60`**
Lexical analysis (scanning)	Token string	**`'A', '=', 'B', '+', '60'`** And *symbol table* for identifiers

Syntax analysis (parsing)
(creates a tree representation that depicts the grammatical structure of the token string)
	Syntax tree
	(each interior node represents an operation and the children of the node represent the arguments of the operation)
		`=`
		`/ \`
		`A   +`
		`/ \`
		`B   60`

Semantic analysis
(checks the source program for semantic consistency with the language definition: type checking and type correction)
	Syntax tree	`=`
		`/ \`
		`A   +`
		`/ \`
		`B   `**`inttofloat`**
		`|`
		`60`

Intermediate code generation (generates a machine-like representation of the source program. This representation should be easy to translate into the target machine)	Three-address code	**`t1 = inttofloat(60)`** **`t2 = B + t1`** **`A = t2`**

The Phases of a Compiler …Continued

Phase	**Output**	**Example**
Code optimization (improves the intermediate code to make it faster, shorter , etc)	Three-address code, quads, or RTL	`t1 = inttofloat(60)` `A = B + t1`

Code generation
(maps the intermediate code into the target language)
	Assembly code	`MOVF  #60.0,r1`
		`ADDF r1,r2`
		`MOVF  r2,A`

Peephole optimizer	Assembly code	`ADDF #60.0,r2` `MOVF  r2,A`