# CSCI-C311 Programming Languages

## An Overview of Compilation

Dr. Hang Dinh

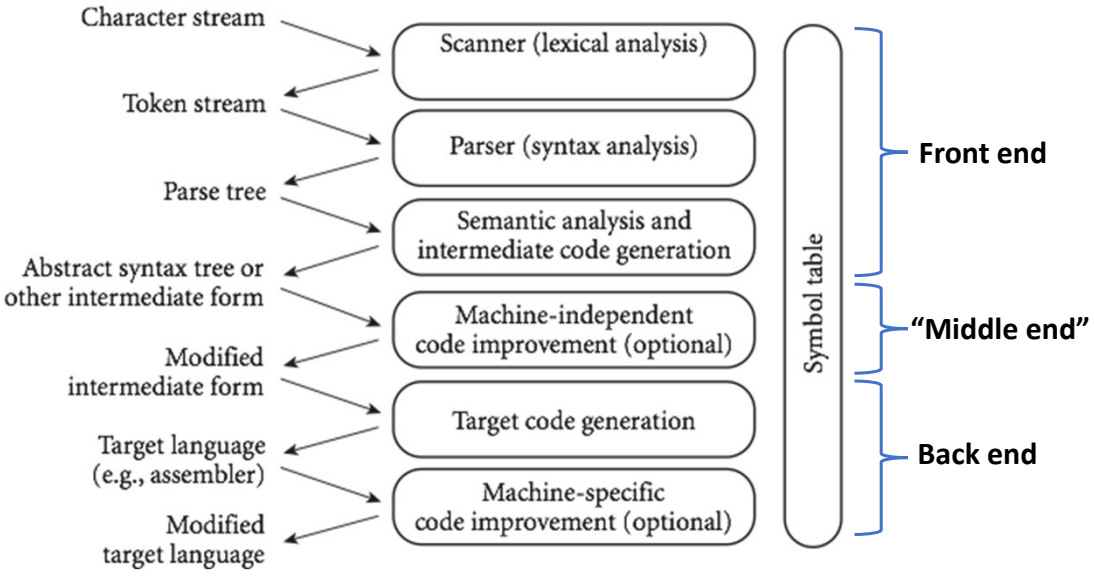
1

## Outline and Reading

- After this lecture, you will learn
  - Phases of compilation
  - An example of scanning or lexical analysis
  - An example of parsing or syntax analysis
- Reading
  - Scott 4e - Section 1.6

2

# Phases of Compilation




3

# Scanning (Lexical Analysis)


- Scanner reads characters from source and groups them into tokens
  - Tokens are the smallest meaningful units of the program
- Scanner's principal purpose: to simplify parser's task
  - by reducing the size of the input (there are many more characters than tokens)
  - by removing extraneous characters like white space.
- Scanner's additional tasks
  - removes comments
  - tags tokens with lines and column numbers, to make it easier to generate good diagnostic in later phases.

4

## Scanning (Lexical Analysis): Example

- GCD program in C

```
int main() {
   int i = getint(), j = getint();
   while (i != j) {
     if (i > j) i = i - j;
     else j = j - i;
   }
   putint(i);
}
```

5

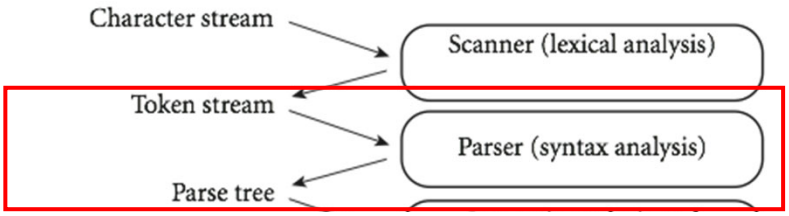
## Scanning (Lexical Analysis): Example

- Tokens of the GCD program:

```
int      main   (   )        {
int      i      =   getint   (   )   ,   j   =   getint   (   )
;
while    (      i   !=       j   )   {
if       (      i   >        j   )   i   =   i   -       j   ;
else     j      =   j        -   i   ;
}
putint   (      i   )        ;
}
```

6

# Parsing (Syntax Analysis)




- Parsing discovers the "*context free*" structure of the program
  - Informally, it finds the structure you can describe with syntax diagrams
- Context-Free Grammar and Parsing
  - Parsing organizes tokens into a ***parse tree*** that represents higher-level constructs (statements, expressions, functions, etc) in terms of their constituents
  - Potentially recursive rules known as ***context-free grammar*** define the ways in which these constituents combine

7

# Parsing (Syntax Analysis): Example

- A (partial) context-free grammar defining `while` loop in C

*iteration-statement* → *while ( expression ) statement*

*statement* → *compound-statement*
*compound-statement* → *{ block-item-list_opt }*

*block-item-list_opt* → *block-item-list*
*block-item-list_opt* → $\epsilon$

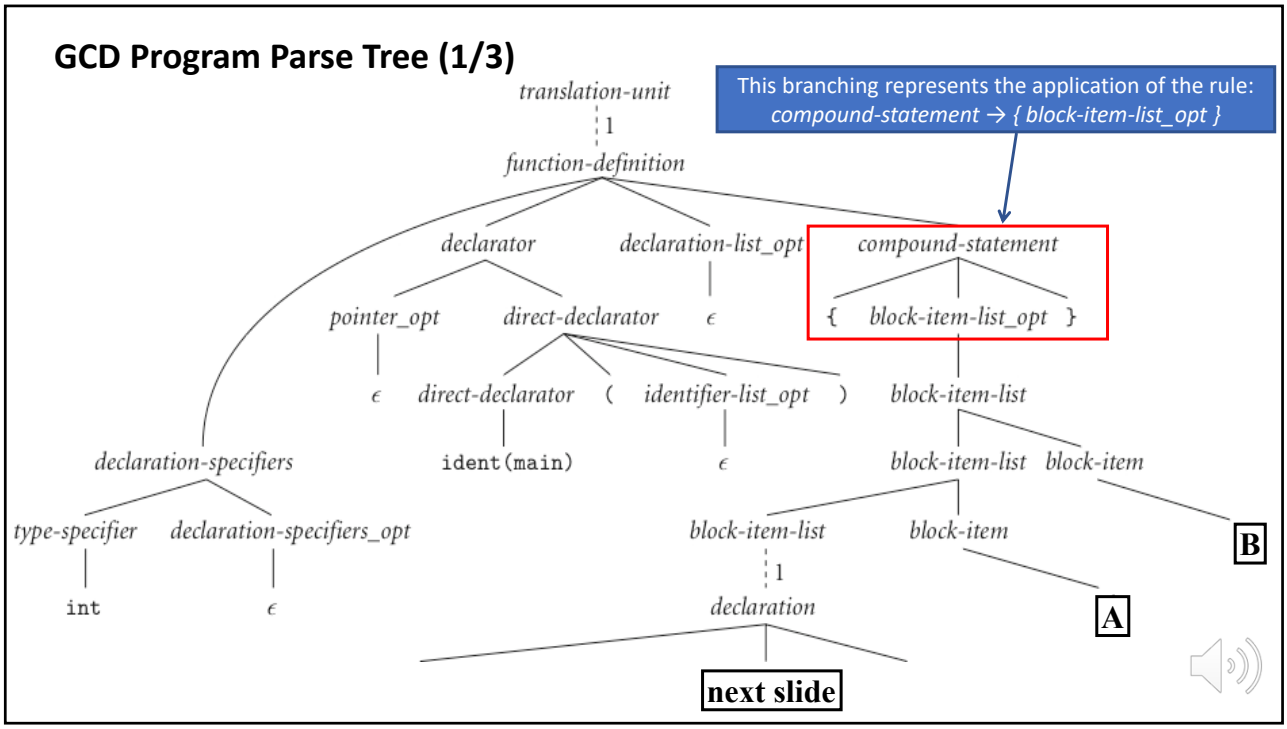
*block-item-list* → *block-item*
*block-item-list* → *block-item-list block-item*
*block-item* → *declaration*
*block-item* → *statement*

statement is often a list enclosed in braces

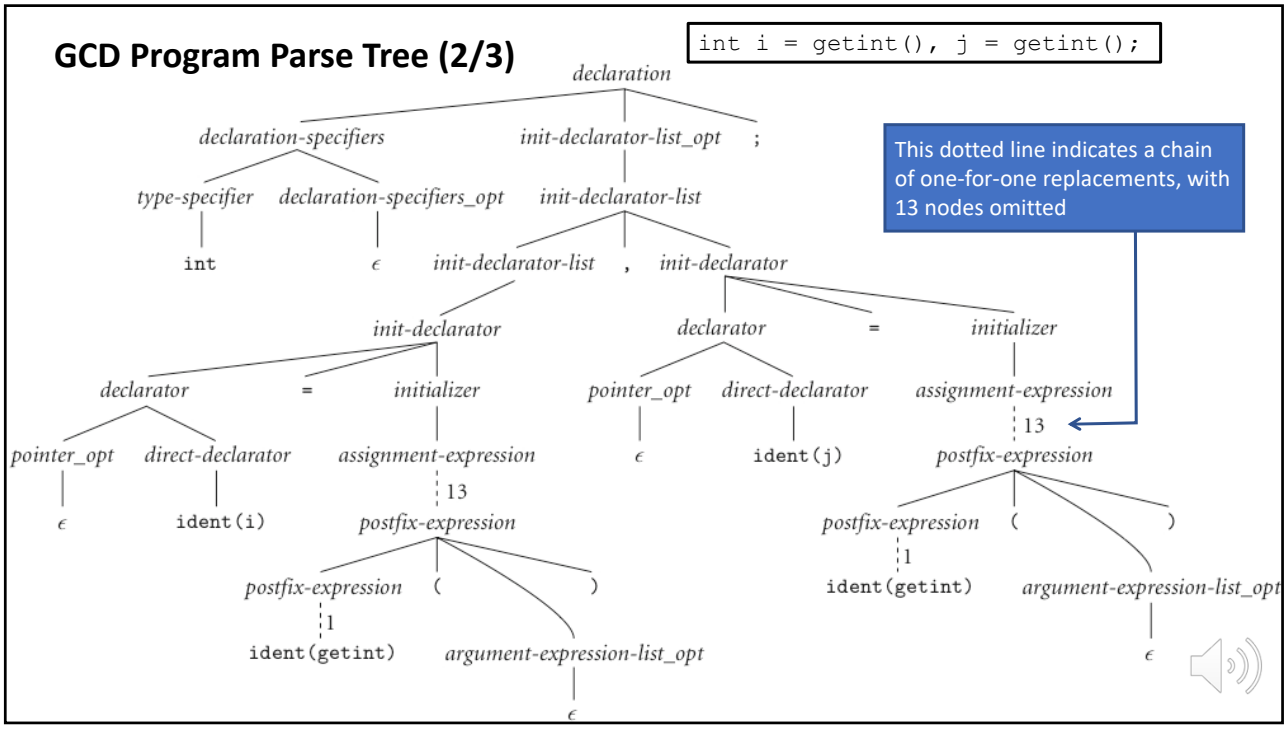
$\epsilon$ represents the empty string

Recursive rule

8

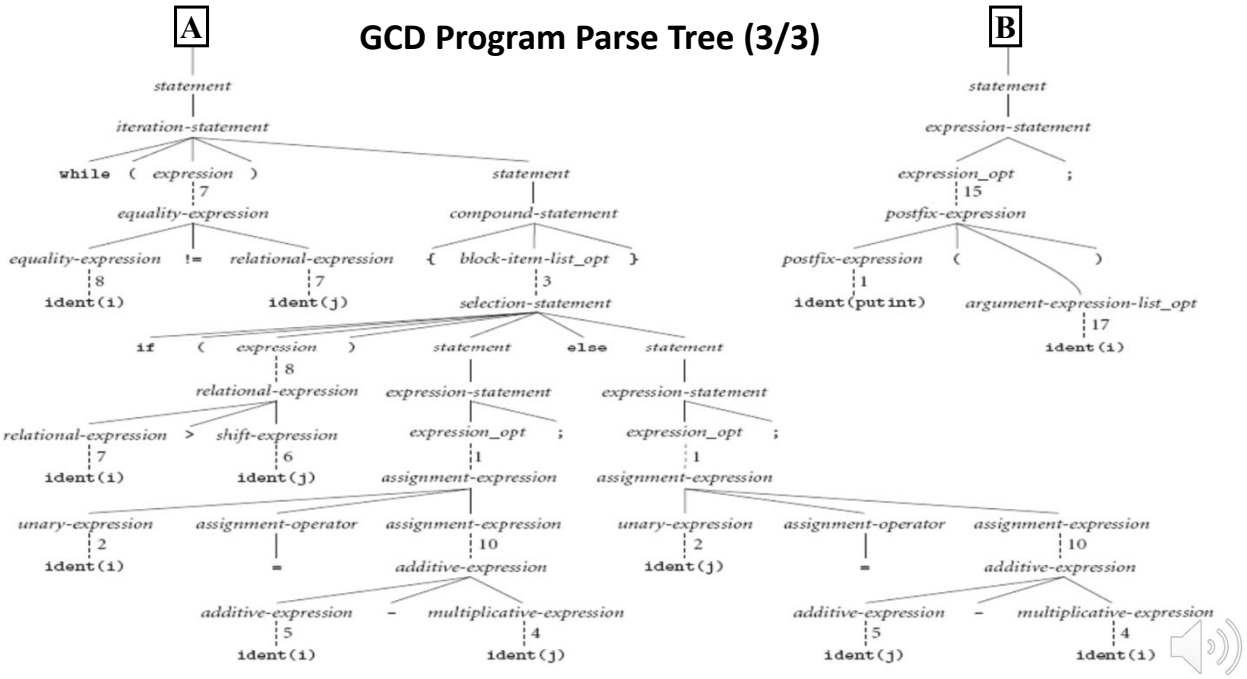

GCD Program Parse Tree (1/3)
This branching represents the application of the rule: compound-statement → { block-item-list_opt }
translation-unit
1
function-definition
declaration-specifiers
type-specifier
int
declaration-specifiers_opt
ϵ
declarator
pointer_opt
ϵ
direct-declarator
ident(main)
( identifier-list_opt )
ϵ
declaration-list_opt
ϵ
compound-statement
{ block-item-list_opt }
block-item-list
block-item-list block-item
block-item-list
1
declaration
block-item
A
B
next slide

9


GCD Program Parse Tree (2/3)
int i = getint(), j = getint();
declaration
declaration-specifiers
type-specifier
int
declaration-specifiers_opt
ϵ
init-declarator-list_opt
;
init-declarator-list
init-declarator-list , init-declarator
init-declarator
declarator
pointer_opt
ϵ
direct-declarator
ident(i)
= initializer
assignment-expression
13
postfix-expression
postfix-expression ( )
1
ident(getint)
argument-expression-list_opt
ϵ
declarator
pointer_opt
ϵ
direct-declarator
ident(j)
= initializer
assignment-expression
13
postfix-expression
postfix-expression ( )
1
ident(getint)
argument-expression-list_opt
ϵ
This dotted line indicates a chain of one-for-one replacements, with 13 nodes omitted

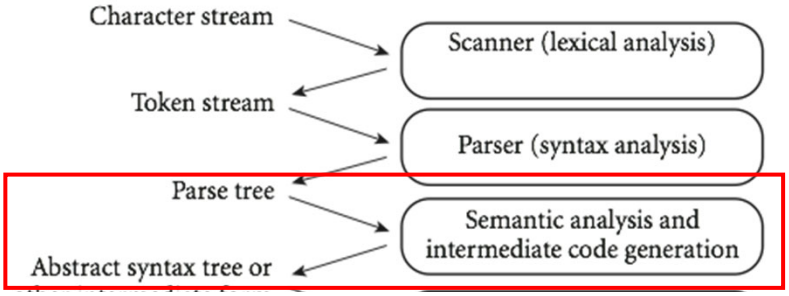
10

## GCD Program Parse Tree (3/3)




11

## Semantic Analysis




- Semantic analysis is the discovery of *meaning* in the program
  - recognize when multiple occurrences of the same identifier are meant to refer to the same program entity
  - ensure that the uses of the identifier are consistent.
  - track the types of both identifiers and expressions

12

## Semantic Analysis Tool: Symbol Table

- Semantic analyzer typically builds and maintains a *symbol table*
  - that maps each identifier to information (e.g., type, scope) known about it.
- Using symbol table, semantic analyzer enforces semantic rules.
- Examples of semantic rules in C
  1. Every identifier is declared before it is used
  2. No identifier is used in an inappropriate context
  3. Subroutine calls provide the correct number and types of arguments
  4. Labels on the arms of a `switch` statement are distinct constants
  5. Any function with a non-`void` return type returns a value explicitly.

13

## Static Semantic vs. Dynamic Semantic

- Compiler actually does **static** semantic analysis
- Static semantic
  - Semantic rule that can be checked at compile time
- Dynamic semantic
  - Semantic rule that can't be checked until run time.
- Examples of dynamic semantics
  1. Variables are never used unless they have been given a value.
  2. Array subscript must lie in bound.
  3. Arithmetic operations do not overflow.

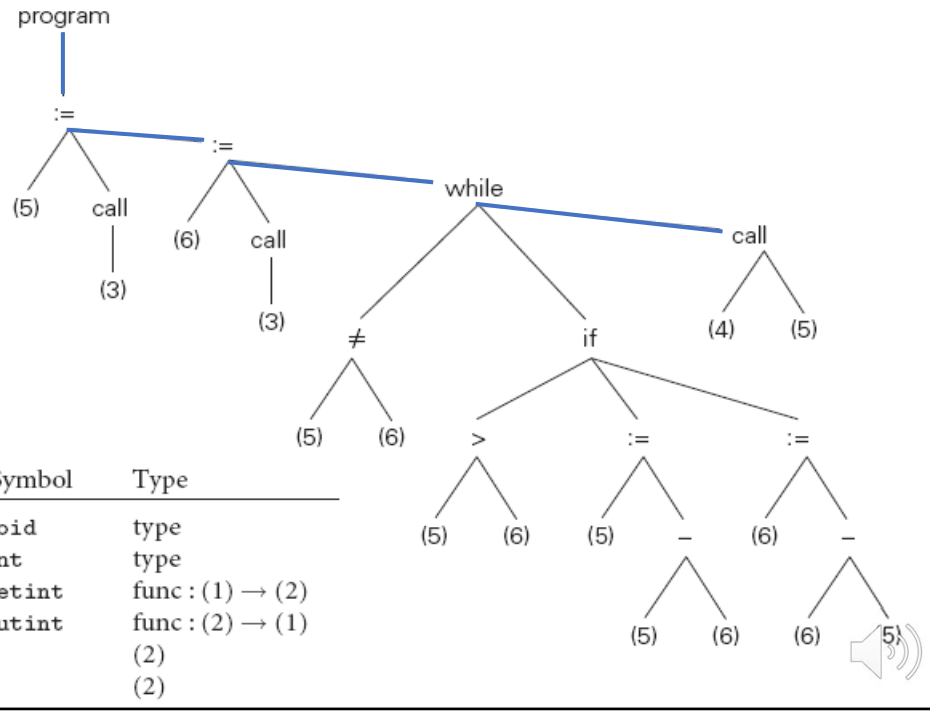
14

# Semantic Analysis Output: Syntax Tree

- Input to semantic analyzer is a parse tree (or *concrete syntax tree*)
  - Once we know that a token sequence is valid, much of the information in the parse tree is irrelevant to further phases of compilation.
- While checking semantic rules, the semantic analyzer
  - transforms the parse tree into an *abstract syntax tree* (or simply a *syntax tree*) by removing most of the "artificial" nodes in the parse tree's interior.
  - annotates the remaining nodes with useful information (e.g., pointers from identifiers to their symbol table entries)
    - The annotations attached to a particular node are known as its *attributes*.
- Annotated syntax tree represents the running program
  - "execution" amounts to tree traversal

15

- Example: **Syntax tree** and **symbol table** for the GCD program




| Index | Symbol | Type |
|---|---|---|
| 1 | `void` | type |
| 2 | `int` | type |
| 3 | `getint` | func : (1) → (2) |
| 4 | `putint` | func : (2) → (1) |
| 5 | `i` | (2) |
| 6 | `j` | (2) |



16

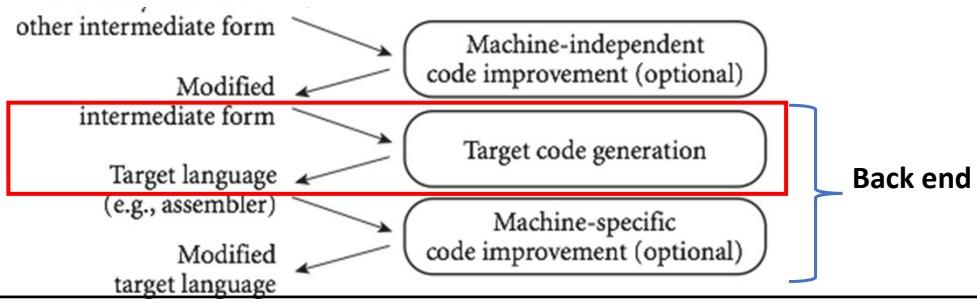
## Intermediate Form

- Intermediate form done after semantic analysis (if the program passes all checks)
  - In many compilers, annotated syntax tree constitutes the intermediate form that is passed from the front end to the back end.
  - In other compilers, semantic analysis ends with a traversal of the tree that generates some other intermediate form (e.g., *control flow graph*)
- Intermediate forms are often chosen for machine independence, ease of optimization, or compactness (these are somewhat contradictory)
- In a suite of related compilers (like Visual Studio)
  - the front ends for several languages and the back ends for several machines would share a common intermediate form

17

## Target Code Generation

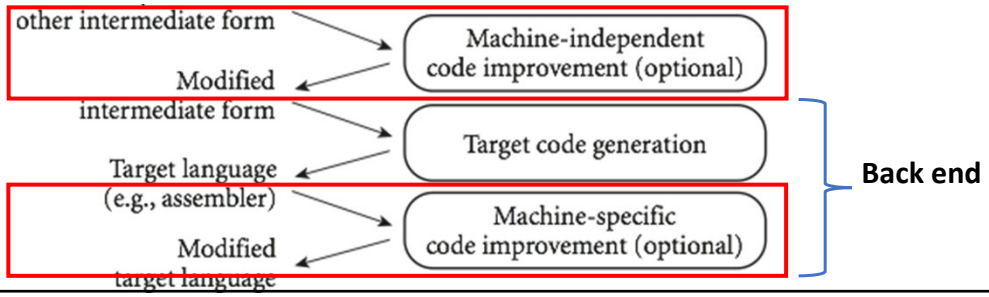
- Code generation translates the intermediate form into target language
  - produces assembly language or (sometime) relocatable machine language
- To generate assembly or machine language, the code generator
  - traverses the symbol table to assign locations to variables,
  - traverses intermediate form, generating loads and stores for variable references




18

# Code Improvement (Optimization)

- ***Optimization*** transforms a program into a new version that does the same thing faster, or in less space
  - The term is a misnomer; we just improve code
- Machine-independent optimizations performed on intermediate form
- Machine-specific optimizations performed on target program




19