COMPILER CONSTRUCTION

Zhihua Jiang

Reminder

- QQ group: 374296679

- Teacher’s email:
  22799204@qq.com

群名称：CC2024
群　号：374296679

Grade System

- Attendance
- Homework
- Final exam (closed-book)

Reference Materials

There is no required text for this course. The following books are recommended as optional reading:

- <Compiler Construction – Principles and Practice>, Kenneth C. Louden, China Machine Press, ￥58.0, ISBN：978-7-111-10842-9
- <Compiler>, Alfred V. Aho *et al.*, China Machine Press, ￥78.0, ISBN：978-7-111-32674-8
- Stanford CS143 - Compilers (http://web.stanford.edu/class/cs143/)

CS143 Compilers

Welcome to CS143! Assignments and handouts will be available here. Discussion will happen through Ed Discussion on **Canvas**. Written assignments will be handed in through **Gradescope**.

Lectures are held Tuesday and Thursday mornings at 10:30-11:50 in Gates B1.

More Information

- **Schedule/Syllabus**
- **Course Information**
- **Course Policies**
- **Canvas**
- **Ed Discussion**
- **Gradescope (WAs)**
- Stanford myth: ssh to myth.stanford.edu

Handouts

- **Final 2022** (**Solutions**)
- **Final 2021** (**Solutions**)
- **Midterm 2023** (**Solutions**)
- **Midterm 2022** (**Solutions**)
- **Midterm 2021** (**Solutions**)

Assignments

- **Programming Assignment 1**
- **Written Assignment 1** (**LaTeX template**)

Resources

- **Cool Reference Manual**
- **Tour of the Cool Support Code**
- **Flex Manual**
- **Bison Manual**
- **Cool Runtime**
- **SPIM Manual**

Lectures

1. **Course Overview**
2. **Cool: The Course Project**
3. **Lexical Analysis**
4. **Implementation of Lexical Analysis**
5. **Introduction to Parsing**
6. **Syntax-Directed Translation**
7. **Top-Down Parsing & Bottom-Up Parsing I**
8. **Bottom-Up Parsing II**
9. **Semantic Analysis & Type Checking I**
10. **Type Checking II**
11. **Run-time Environments**
12. **Code Generation**
13. **Operational Semantics**
14. **Intermediate Code & Local Optimization**
15. **Global Optimization**
16. **Register Allocation**
17. **Automatic Memory Management**

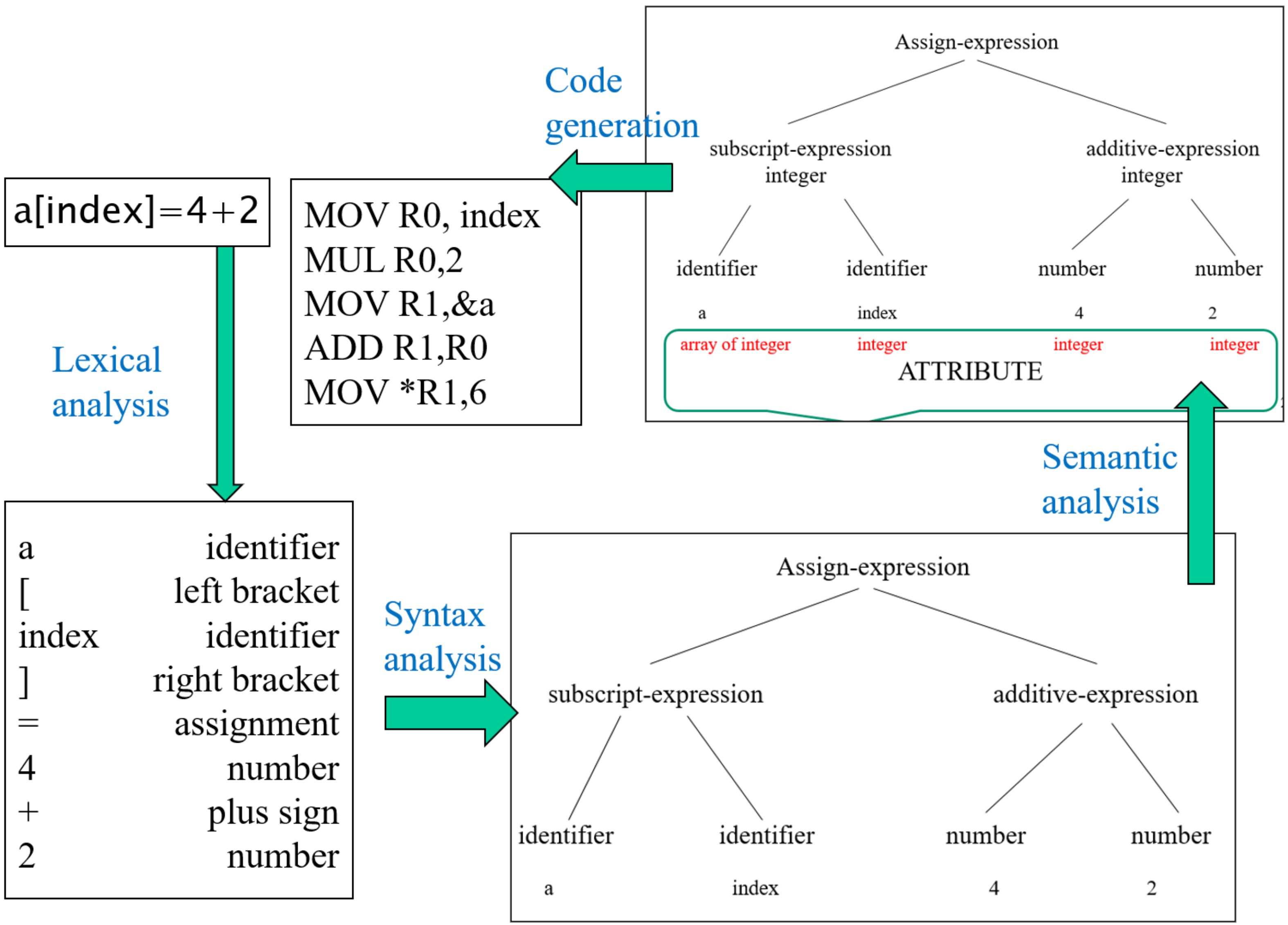
Outline

1. INTRODUCTION

2. **SCANNING**
3. CONTEXT-FREE GRAMMARS AND **PARSING**
4. TOP-DOWN PARSING
5. BOTTOM-UP PARSING

6. **SEMANTIC ANALYSIS**
7. RUNTIME ENVIRONMENT
8. **CODE GENERATION**

A simple example:



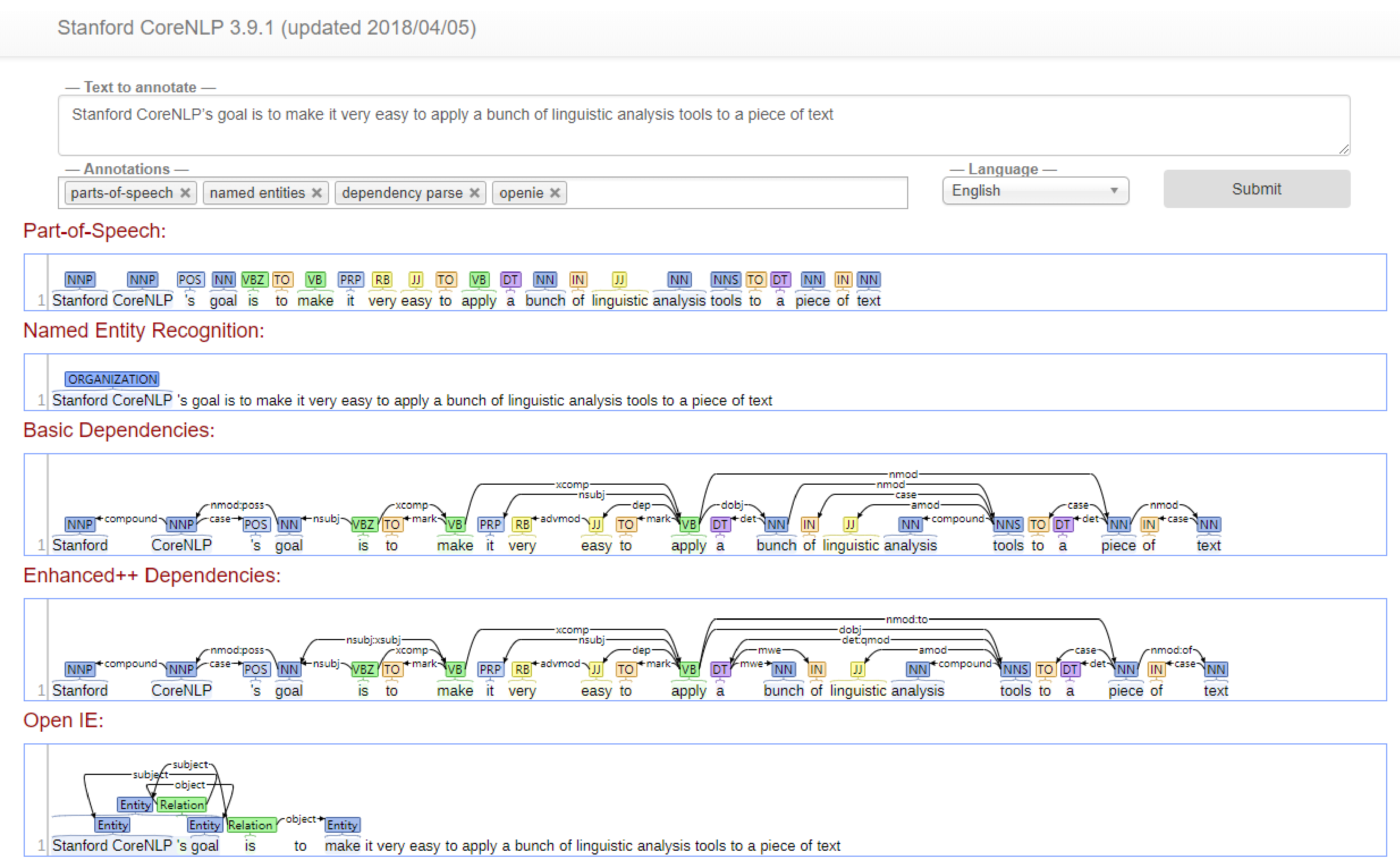
1. INTRODUCTION

What is a compiler?

- A **computer program** translates one language to another

Source Program → ***Compiler*** → Target Program

- A compiler is a complex program
  - From 10,000 to 1,000,000 lines of codes
- Compilers are used in many forms of computing
  - Programming language (PL) vs. natural language (NL)



https://corenlp.run/

What is the purpose of this course

- This course is to provide basic knowledge
  - Theoretical techniques, such as automata theory
- This course is to give necessary tools and practical experience
  - A series of simple examples
  - Sample languages: TINY, C-Minus

Main Topics

1.1 Why Compilers? A Brief History
1.2 Programs Related to Compilers
1.3 The Translation Process
1.4 Major Data Structures in a Compiler
1.5 Other Issues in Compiler Structure
1.6 Bootstrapping and Porting
1.7 The TINY Sample Language and Compiler
1.8 C-Minus: A Language for a Compiler Project

1.1 Why Compilers? A Brief History

Why Compiler

- Writing machine language codes is time consuming and tedious
- The assembly language has a number of defects
  - Not easy to write, read and understand

C7 06 0000 0002

Mov x, 2

X=2

same meaning but different codes

Brief History of Compiler

- The **first compiler** was developed between 1954 and 1957
  - The FORTRAN language and its compiler by a team at IBM led by John Backus
  - The structure of natural language was studied at about the same time by Noam Chomsky

Brief History of Compiler

- The related **theories and algorithms** in the 1960s and 1970s
  - The **classification** of language: Chomsky hierarchy
  - The **parsing** problem was pursued:
    - Context-free language, parsing algorithms
  - The **symbolic methods** for expressing the structure of the words of a programming language:
    - Finite automata, Regular expressions
  - Methods have been developed for **generating efficient object code**:
    - Optimization techniques or code improvement techniques

Brief History of Compiler

- Programs were developed to **automate the complier** development for parsing
  - Parser generators,
    - such as Yacc by Steve Johnson in 1975 for Unix
  - Scanner generators,
    - such as Lex by Mike Lesk for Unix about same time

Brief History of Compiler

- Projects focused on automating the generation of other parts of a compiler
  - Code generation was undertaken during the late 1970s and early 1980s
  - Less success due to our less than perfect understanding of them

Brief History of Compiler

- **Recent advances** in compiler design
  - **More sophisticated algorithms** for inferring and/or simplifying the information contained in program,
  - Window-based Interactive Development Environment,
    - **IDE**, that includes editors, linkers, debuggers, and project managers.
  - However, the basic of compiler design have **not changed** much in the last 20 years.

1.2 Programs related to Compiler

Interpreters

- **Execute** the source program **immediately** rather than generating object code
- Examples: BASIC, LISP, used often in educational or development situations
- Speed of execution is **slower** than compiled code
- Share many of their operations with compilers

Assemblers

- A **translator for the assembly language** of a particular computer
- Assembly language is a **symbolic form** of one machine language
- A compiler may generate assembly language as its target language and an assembler finished the translation into object code

Other programs

- Linker
- Loader
- Preprocessor
- Editor
- Debugger
- Project manager

……

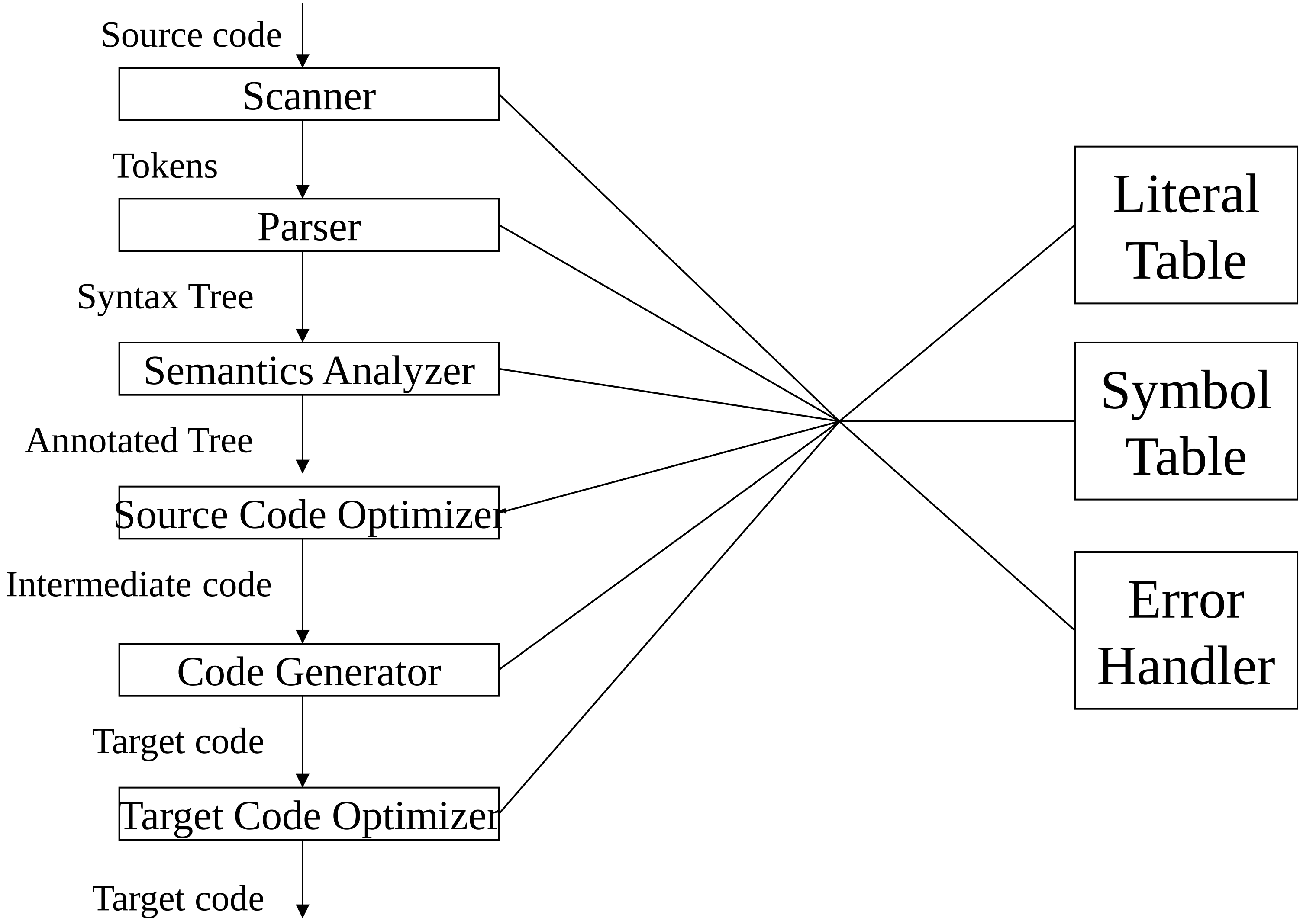
1.3 The Translation Process

The phases of a compiler

- Six phases
  - Scanner
  - Parser
  - Semantics Analyzer
  - Source code optimizer
  - Code generator
  - Target Code Optimizer

- Three auxiliary components
  - Literal table
  - Symbol table
  - Error Handler

The Phases of a Compiler



The Scanner

- ***<u>Lexical analysis</u>***: it collects **sequences of characters** into meaningful units called **tokens**
- An example: a[index]=4+2

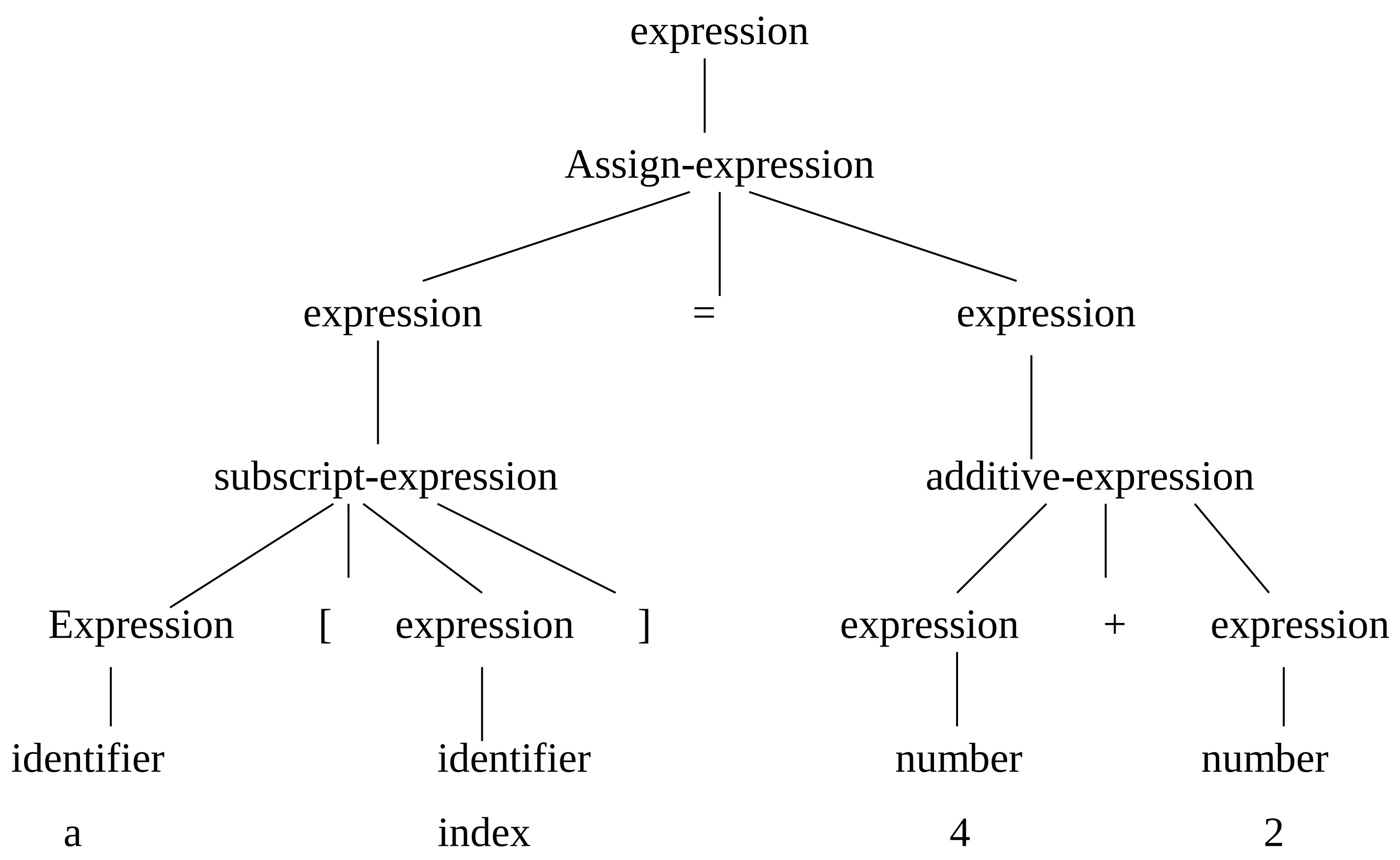
• a	identifier
• [	left bracket
• index	identifier
•]	right bracket
• =	assignment
• 4	number
• +	plus sign
• 2	number

- Other operations: it may enter literals into the literal table

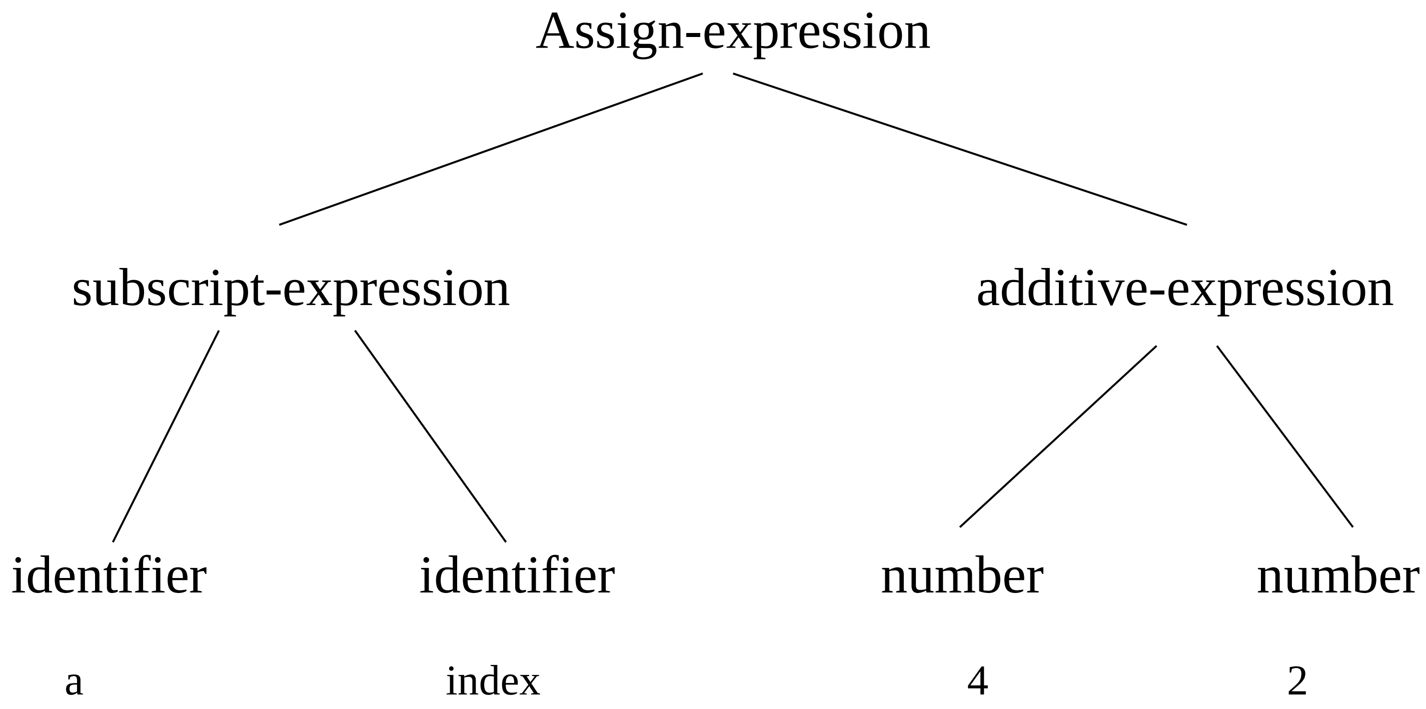
The Parser

- ***Syntax analysis***: it determines the **structure** of the program
- The results of syntax analysis are a **parse tree** or a **syntax tree**
- An example: a[index]=4+2
  - Parse tree
  - Syntax tree (i.e. abstract syntax tree)

Parse Tree


expression
Assign-expression
expression
=
expression
subscript-expression
additive-expression
Expression
[
expression
]
expression
+
expression
identifier
identifier
number
number
a
index
4
2

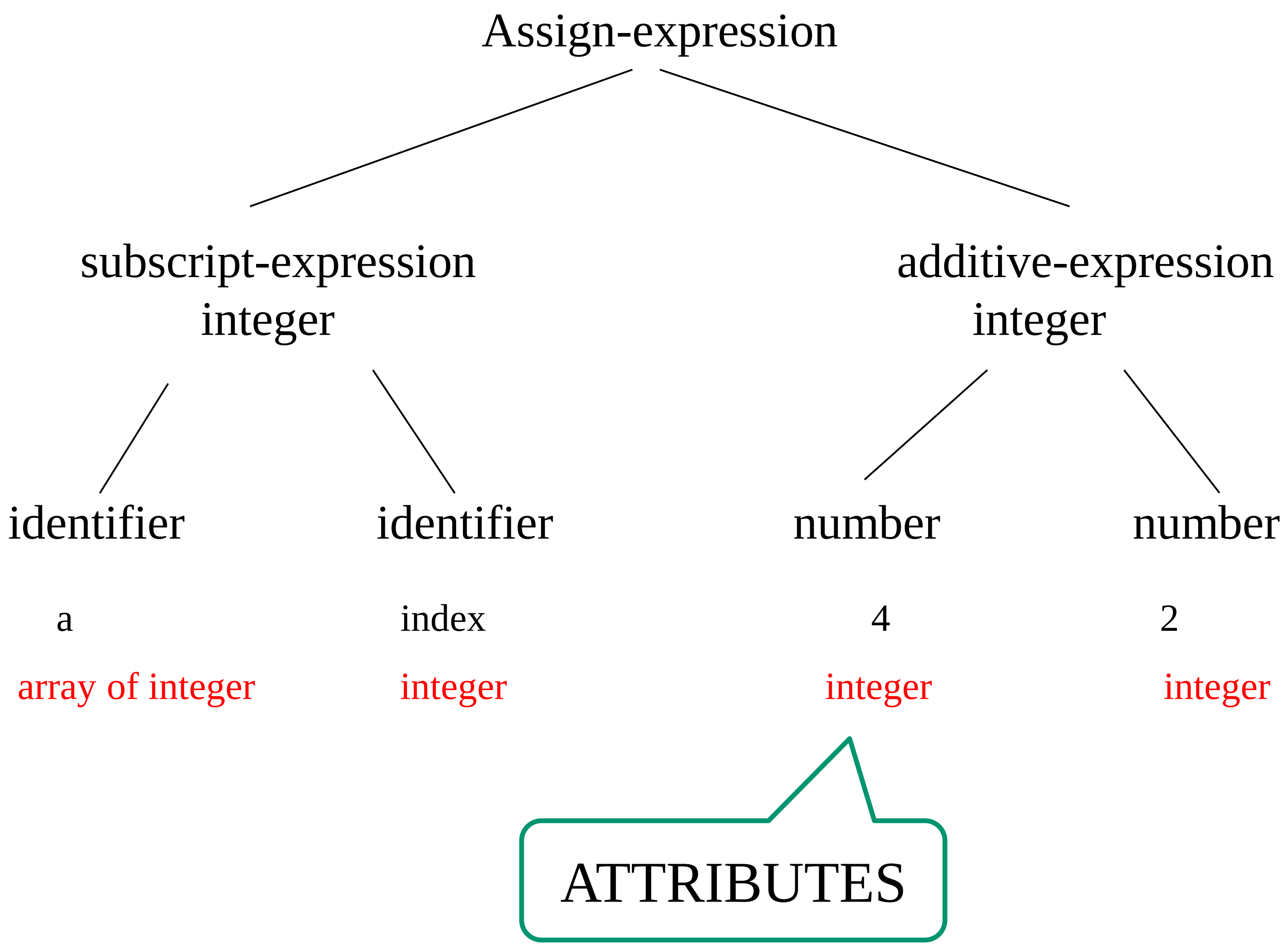
Abstract Syntax Tree



The Semantic Analyzer

- The **semantics** of a program is its "**meaning**", as opposed to its **syntax**, or **structure**, that
  - determines some of its **running time behaviors** prior to execution.
- **Static semantics**: declarations or type checking
- **Attributes**: The extra pieces of information computed by semantic analyzer
- An example: a[index]=4+2
  - The syntax tree annotated with attributes

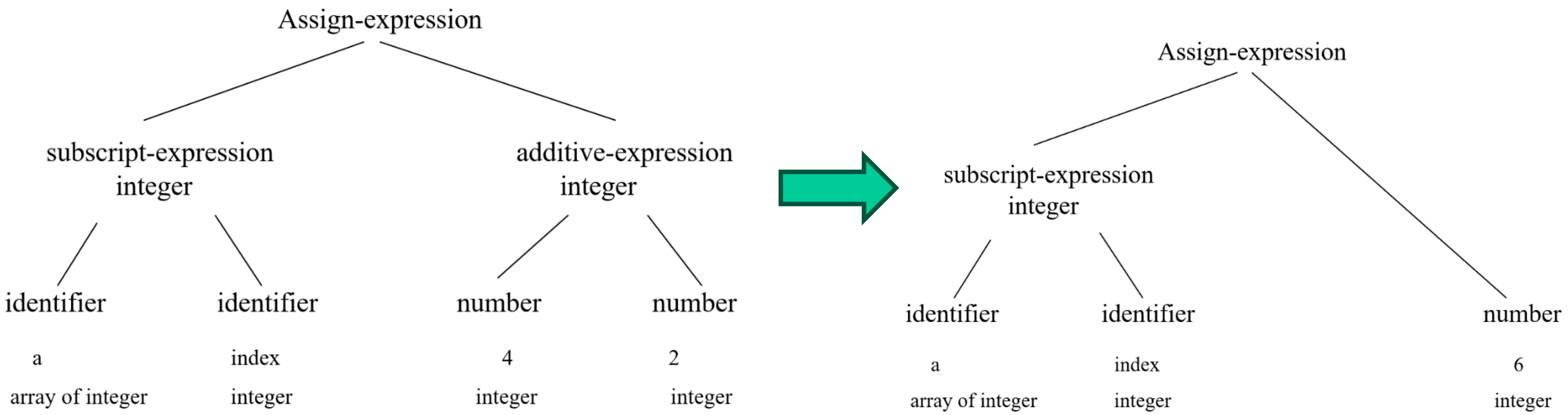
Annotated Syntax Tree



The Source Code Optimizer

- The **earliest point** of optimization steps is just after semantic analysis
- The code improvement depends only on the source code, and as a separate phase
- Ex. a[index]=4+2
  - Constant folding performed directly on annotated tree
  - Using intermediate code: three-address code or p-code

Optimizations on Annotated Tree



Optimization on Intermediate Code

```
t = 4 + 2
a[index]=t
```

```
t = 6
a[index]=t
```

```
a[index]=6
```

Code Generation

- It takes the intermediate code and generates code for target machine
- The properties of the target machine become the major factor:
  - Using instructions and representation of data
- An example: a[index]=4+2
  - Code sequence in a hypothetical assembly language

A possible code sequence

a[index]=6

```
MOV R0, index
MUL R0,2
MOV R1,&a
ADD R1,R0
MOV *R1,6
```

The Target Code Optimizer

- It improves the target code generated by the code generator:
  - Address modes choosing
  - Instructions replacing
  - Redundant eliminating

```
MOV R0, index
MUL R0,2
MOV R1,&a
ADD R1,R0
MOV *R1,6
```

→

```
MOV R0, index
SHL R0
MOV &a[R0],6
```

1.4 Major Data Structure in a Compiler

Principle Data Structure for Communication among Phases

- TOKEN
  - A scanner collects characters into a token, as a value of an enumerated data type for tokens
  - May also preserve the string of characters or other derived information, such as name of identifier, value of a number token
- SYNTAX TREE
  - A standard pointer-based structure generated by parser
  - Each node represents information collected by parser or later, which maybe dynamically allocated or stored in symbol table
  - The node requires different attributes depending on kind of language structure, which may be represented as variable record.

Principle Data Structure for Communication among Phases

- SYMBOL TABLE
  - Keeps information associated with identifiers, functions, variables, constants, and data types
  - Interacts with almost every phase of compiler
  - Access operation need to be constant-time
  - One or several hash tables are often used
- LITERAL TABLE
  - Stores constants and strings, reducing size of program
  - Quick insertion and lookup are essential

Principle Data Structure for Communication among Phases

- INTERMEDIATE CODE
  - Kept as an array of text string, a temporary text, or a linked list of structures, depending on kind of intermediate code
  - Should be easy for reorganization
- TEMPORARY FILES
  - Holds the product of intermediate steps during compiling
  - Solve the problem of memory constraints or back-patch addressed during code generation

1.5 Other Issues in Compiler Structure

The Structure of Compiler

- Multiple views from different angles
  - Logical Structure
  - Physical Structure
  - Sequencing of the operations
- A major impact of the structure
  - Reliability, efficiency
  - Usefulness, maintainability

Analysis and Synthesis

- The analysis part of the compiler analyzes the source program to compute its properties
  - Lexical analysis, syntax analysis and semantics analysis, as well as optimization
  - More mathematical and better understood
- The synthesis part of the compiler produces the translated codes
  - Code generation, as well as optimization
  - More specialized
- The two parts can be changed independently of the other

Front End and Back End

- The operations of the front end depend on the source language
  - The scanner, parser, and semantic analyzer, as well as intermediate code synthesis
- The operations of the back end depend on the target language
  - Code generation, as well as some optimization analysis
- The intermediate representation is the medium of communication between them
- This structure is important for compiler portability

Passes

- The repetitions to process the entire source program before generating code are referred as passes.
- Passes may or may not correspond to phases
  - A pass often consists of several phases
  - A compiler can be one pass, which results in efficient compilation but less efficient target code
  - Most compilers with optimization use more than one pass
    - One Pass for scanning and parsing
    - One Pass for semantic analysis and source-level optimization
    - The third Pass for code generation and target-level optimization

Error Handling

- Static (or compile-time) errors must be reported by a compiler
  - Generate meaningful error messages and resume compilation after each error
  - Each phase of a compiler needs different kind of error handling
- Exception handling
  - Generate extra code to perform suitable runtime tests to guarantee all such errors to cause an appropriate event during execution.

End of Chapter One

Thanks