Compiler Principle and Technology

Prof. Dongming LU

Mar. 1st, 2013

Textbook:

COMPILER CONSTRUCTION Principle and Practice

by Kenneth C. Louden

(China Machine Press)

Course Web：

http://netmedia.zju.edu.cn/compiler

Email: ldm@cs.zju.edu.cn (Prof. Lu Dongming)

Reference books:

1. Compilers -- Principles, Techniques and Tools, (Dragon Book), by Aho, Sethi and Ullman (1986)
2. Modern Compiler Implementation in Java, by Andrew Appel (2002)

3.程序设计语言编译原理（第3版），
国防工业出版社,陈火旺等

4.Compiler Design in C, Prentice Hall, Allen I. Holub

5. 编译原理与技术,浙江大学出版社,冯雁等编著

TA: zhgshuai@foxmail.com (Zhang Shuai)

Evaluation:

1. Homeworks = 15%
2. Quizzes = 10%
3. Mid-Term Exam = 20%
4. Final Exams = 55%

Content

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**
  - Command interpreters, interface programs

The purpose of this text

- To provide **basic knowledge**
  - Theoretical techniques, such as automata theory

- To **give necessary tools** and **practical experience** (In Summer Term)
  - A series of simple examples
  - 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

1.1 Why? A Brief History

Why Compiler

- Writing machine language-numeric codes is **time consuming and tedious**

C7 06 0000 0002

Mov x, 2

X=2

- The assembly language has a **number of defects**
  - **Not easy to write**
  - Difficult to read and understand

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: **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**, 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 the Unix system
  - **Scanner generators**: such as Lex by Mike Lesk for Unix system 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:
    - The unification algorithm of Hindley-Milner type checking
  - **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 by a factor of 10 or more
- 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

Linkers

- **Collect separate object files into a directly executable file**
  - **Connect an object program to the code for standard library functions and to resource supplied by OS**

- **Becoming one of the principle activities of a compiler, depends on OS and processor**

Loaders

- **Resolve all re-locatable address relative to a given base**
  - **Make executable code more flexible**
- **Often as part of the operating environment, rarely as an actual separate program**

Preprocessors

- Delete comments, include other files, and perform macro substitutions

- Required by a language (as in C) or can be later add-ons that provide additional facilities

Editors

- Compiler have been **bundled together with** editor and other programs into an interactive development environment (**IDE**)
  - **Oriented toward the format or structure** of the programming language, called structure-based

- May include some operations of a compiler, **informing some errors**

Debuggers

- Used to determine execution error in a compiled program
  - Keep tracks of most or all of the source code information
  - Halt execution at pre-specified locations called breakpoints

- Must be supplied with appropriate symbolic information by the compiler

Profiles

- Collect statistics on the behavior of an object program during execution
  - Called Times for each procedures
  - Percentage of execution time
- Used to improve the execution speed of the program

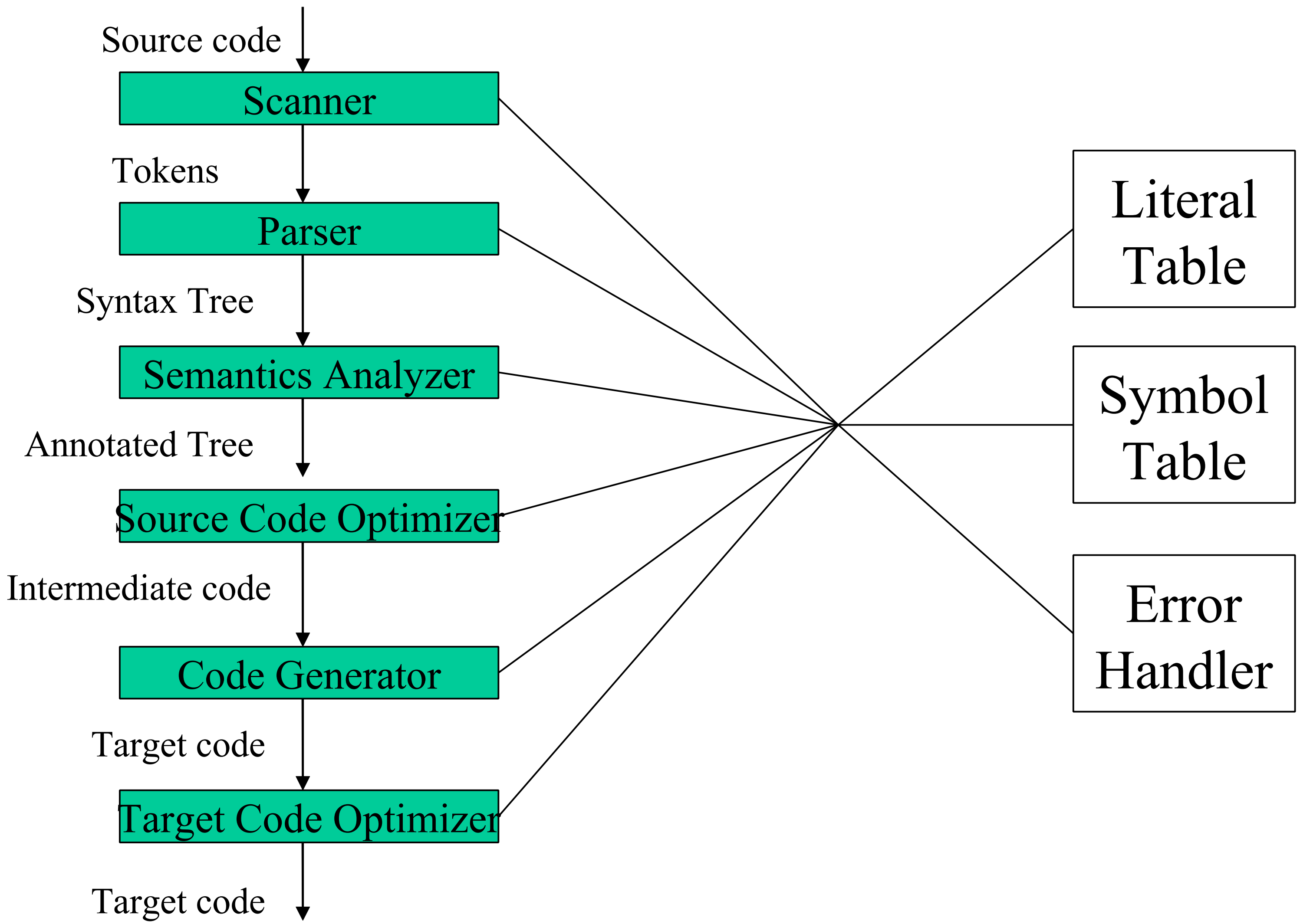
Project Managers

- Coordinate the files being worked on by different people, maintain coherent version of a program
  - Language-independent or bundled together with a compiler

- Two popular project manager programs on Unix system
  - Sccs (Source code control system)
  - Rcs (Revision control system)

1.3 The Translation Process

The Phases of a Compiler



The **phases** of a compiler

- **Six phases**
  - Scanner
  - Parser
  - Semantic Analyzer
  - Source code optimizer
  - Code generator
  - Target Code Optimizer
- **Three auxiliary components**
  - Literal table
  - Symbol table
  - Error Handler

The Scanner

- **Lexical analysis: 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

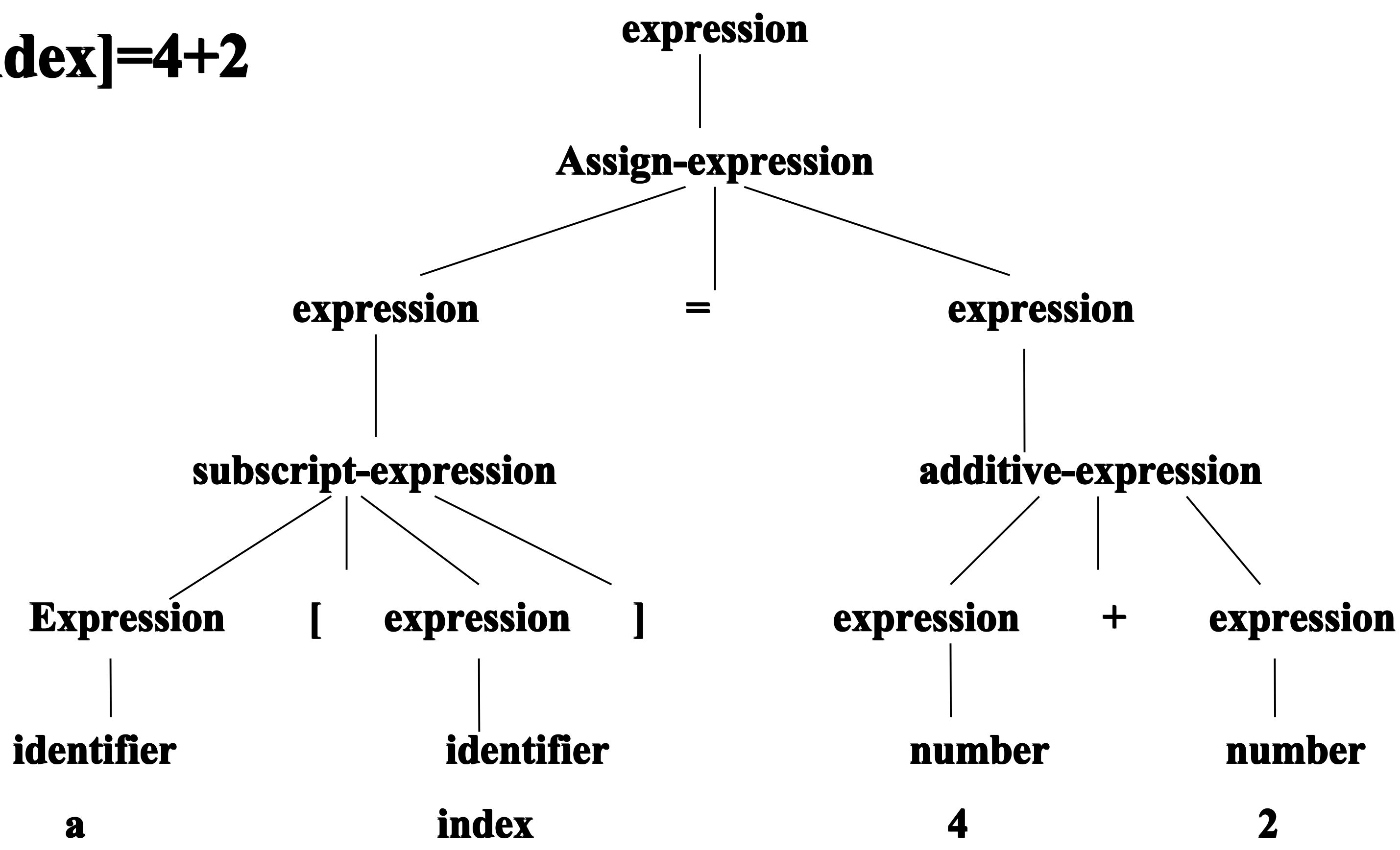
- **Other operations: it may enter literals into the literal table**

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 (abstract syntax tree)

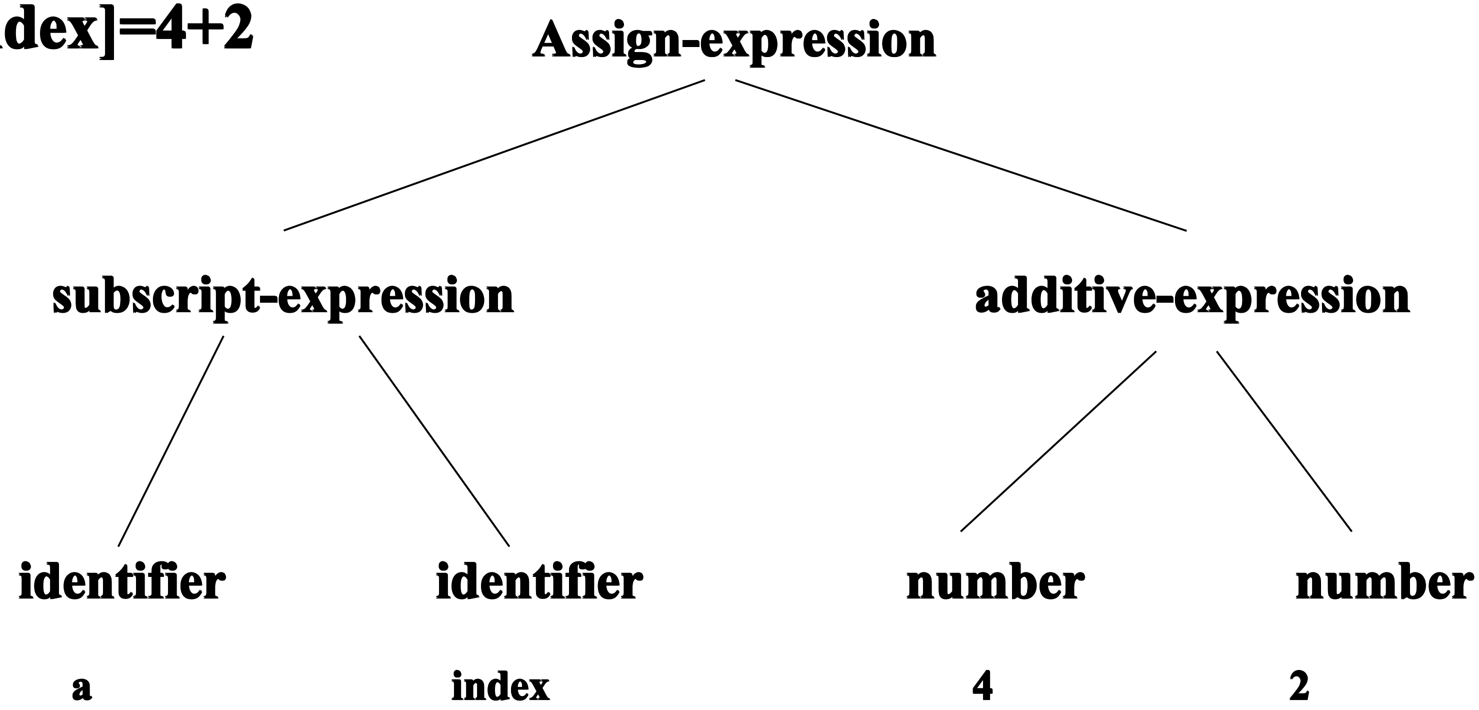
The Parse Tree

a[index]=4+2



The Syntax Tree

a[index]=4+2



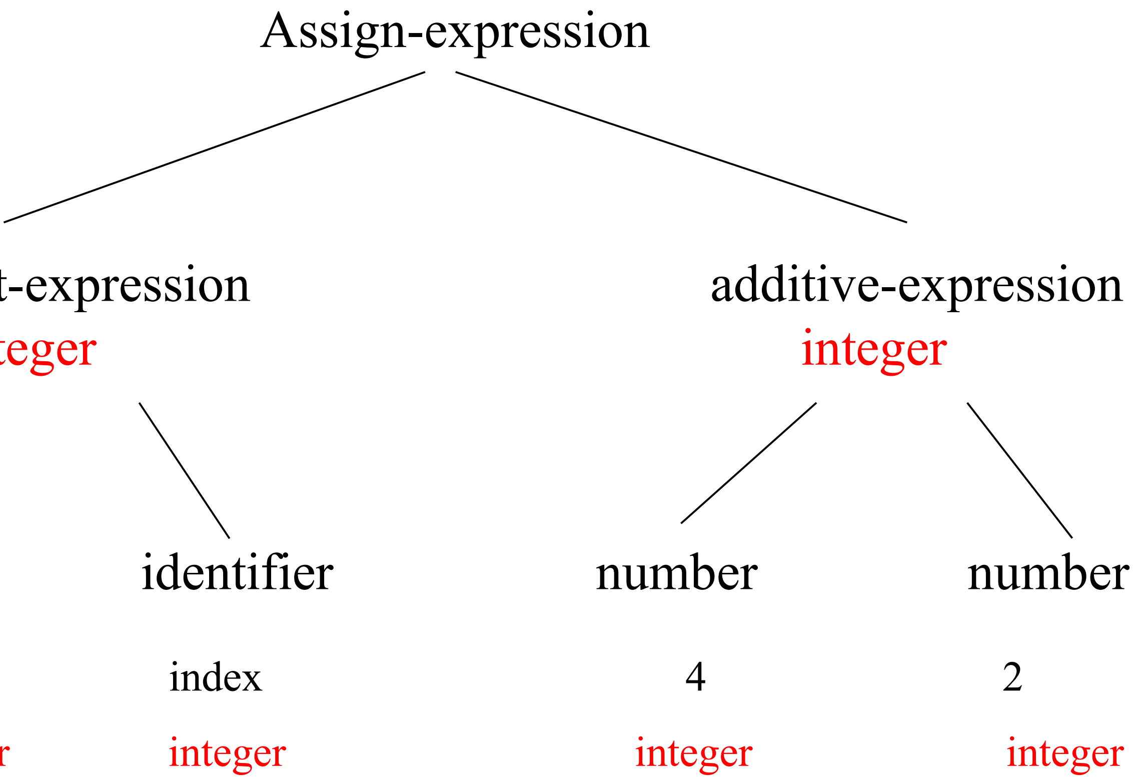
The Semantic Analyzer

- The semantics of a program are its “meaning”, as opposed to its syntax, or structure

- Determining some of its running time behaviors prior to execution.
  - Static semantics: declarations and type checking
  - Attributes: The extra pieces of information computed by semantic analyzer
- An example: a[index]=4+2
  - The syntax tree annotated with attributes

The Annotated Syntax Tree

a[index]=4+2



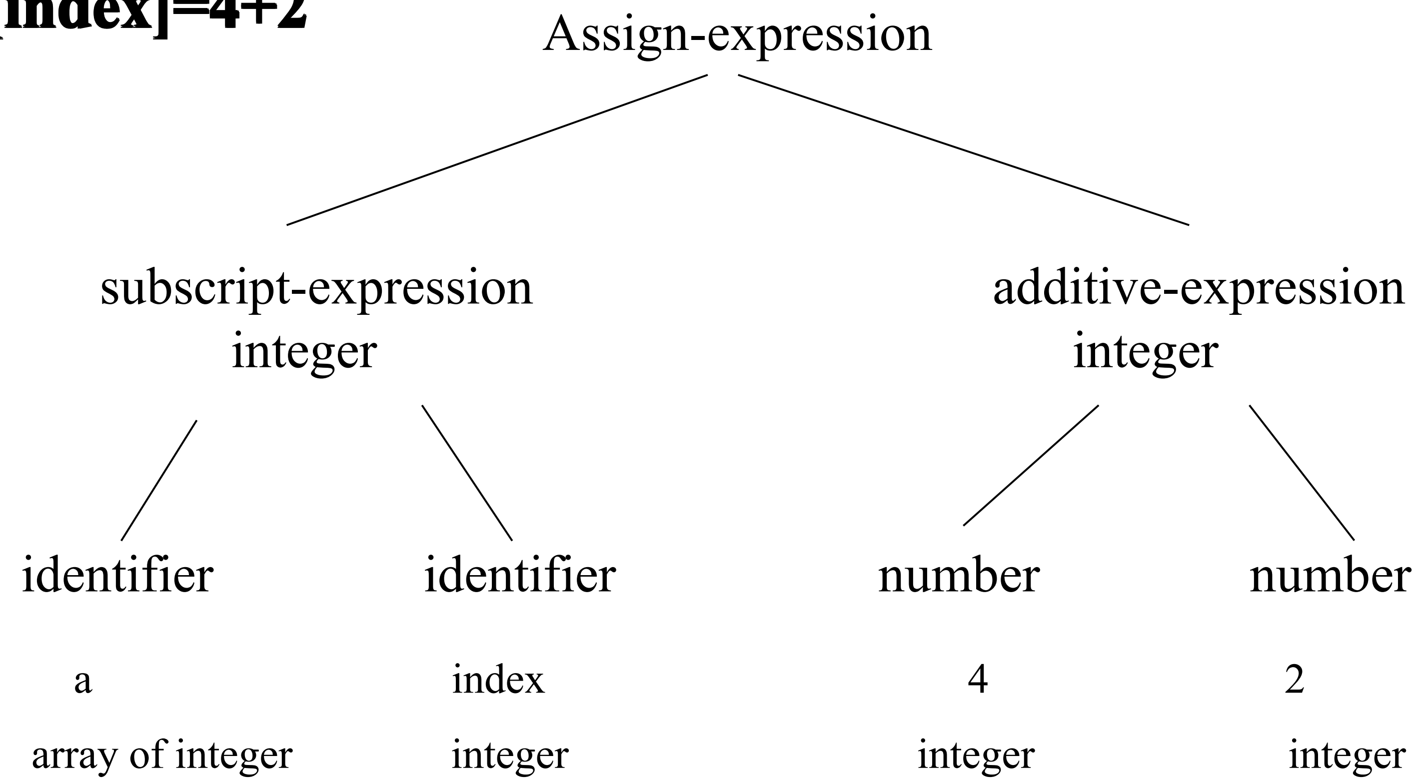
The Source Code Optimizer

- The earliest point of most optimization steps is just after semantic analysis
  - The code improvement depends only on the source code, and as a separate phase
- Individual compilers exhibit a wide variation in optimization kinds as well as placement

- An example: a[index]=4+2
  - Constant folding performed directly on annotated tree
  - Using intermediate code: three-address code, p-code

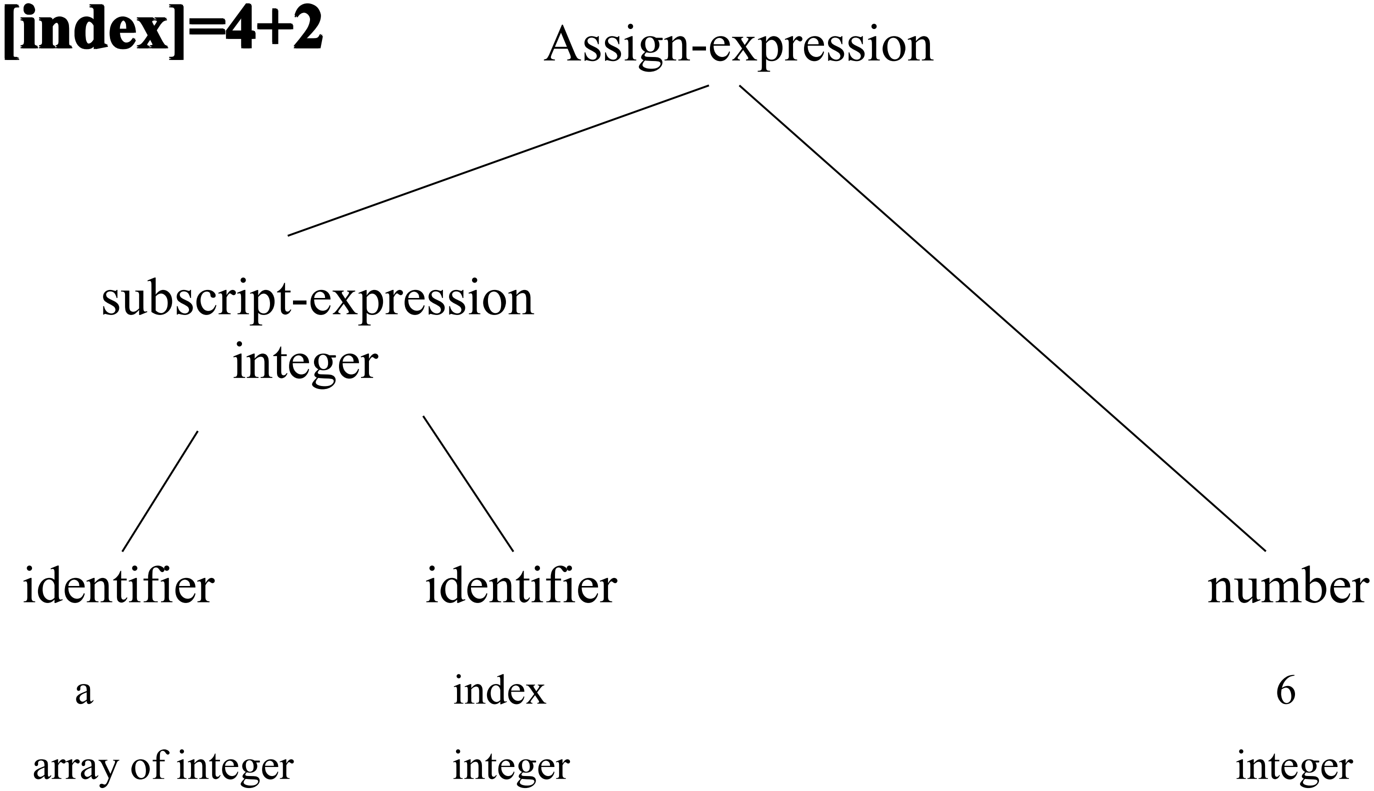
Optimizations on Annotated Tree

a[index]=4+2



Optimizations on Annotated Tree

a[index]=4+2



Optimization on **Intermediate** Code

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

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

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

The Code Generate

- It takes the intermediate code or IR 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
  - As well as redundant eliminating

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

```
MOV R0, index
SHL R0
MOV &a[R1],6
```

1.4 Major Data Structure in a Compiler

Principle Data Structure for Communication among Phases

- **TOKENS**
  - 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
  - A single global variable or an array of tokens

Principle Data Structure for Communication among Phases

- **THE SYNTAX TREE**
  - A standard pointer-based structure generated by parser
  - Each node represents information collect 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

- **THE SYMBOL TABLE**
  - Keeps information associated with identifiers
    - function, variable, 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
- **THE 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 (e.g. three-address code and p-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

Language Definition and compilers

- The lexical and syntactic structure of a programming language
  - Regular expressions
  - Context-free grammar
- The semantics of a programming language in English descriptions
  - Language reference manual, or language definition.

Language Definition and compilers

- A language definition and a compiler are often **developed simultaneously**
  - The techniques have a major impact on definition
  - The definition has a major impact on the techniques

Language Definition and compilers

- The language to be implemented is well known and has an **existing definition**
  - Compiler-conforming-definition is not an easy task
  - A set of standard test programs
- A language occasionally has it semantics given by a formal definition in mathematical term
  - So-called denotational semantics in function programming community
  - Given a mathematical proof that a compiler conforms to the definition

Language Definition and compilers

- The structure and behavior of the runtime environment affect the compiler construction
  - Static runtime environment
  - Semi-dynamic or stack-based environment
  - Fully-dynamic or heap-based environment

Compiler options and interfaces

- Mechanisms for interfacing with the operation system
  - Input and output facilities
  - Access to the file system of the target machine
- Options to the user for various purposes
  - Specification of listing characteristic
  - Code optimization options

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 handing
- Exception handling
  - Generate extra code to perform suitable runtime tests to guarantee all such errors to cause an appropriate event during execution.

1.6 Bootstrapping and Porting

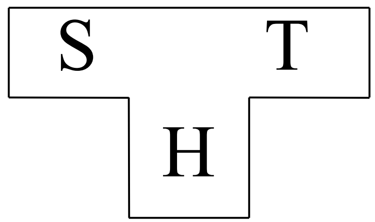
Third Language for Compiler Construction

- Machine language
  - Compiler to execute immediately

- Another language with existed compiler on the same target machine : (First Scenario)
  - Compile the new compiler with existing compiler
- Another language with existed compiler on different machine : (Second Scenario)
  - Compilation produce a cross compiler

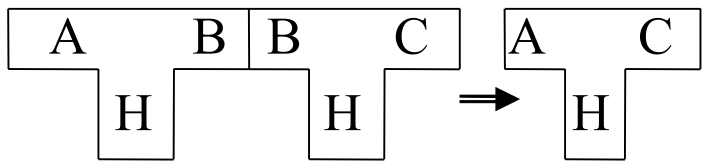
T-Diagram Describing Complex Situation

- A compiler written in language H that translates language S into language T.



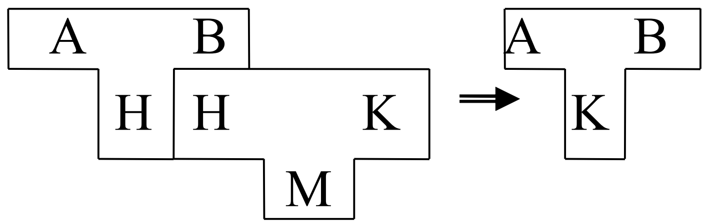
- T-Diagram can be combined in two basic ways.

The First T-diagram Combination



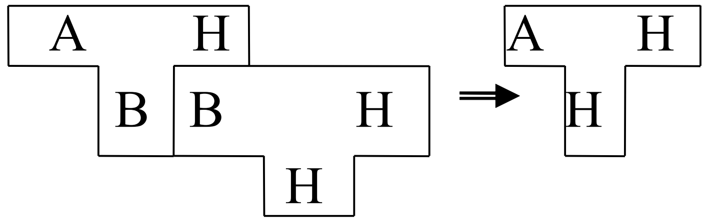
- Two compilers run on the same machine H
  - First from A to B
  - Second from B to C
  - Result from A to C on H

The Second T-diagram Combination



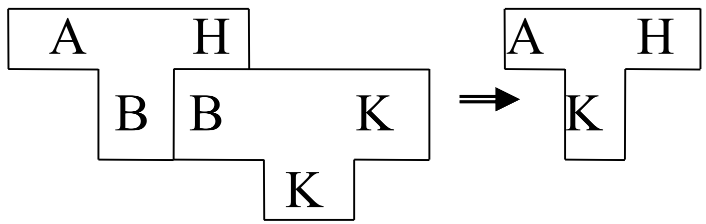
- Translate implementation language of a compiler from H to K
- Use another compiler from H to K

The First Scenario



- Translate a compiler from A to H written in B
  - Use an existing compiler for language B on machine H

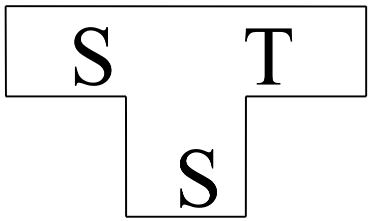
The Second Scenario



- Use an existing compiler for language B on different machine K
  - Result in a cross compiler

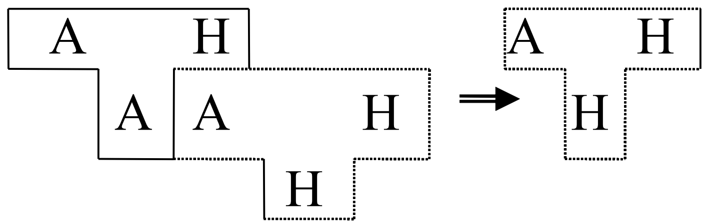
Process of Bootstrapping

- Write a compiler in the same language



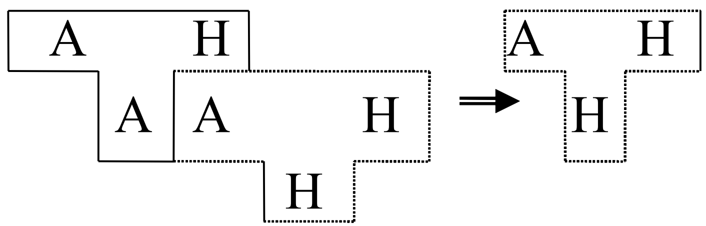
- No compiler for source language yet
- Porting to a new host machine

The First step in bootstrap



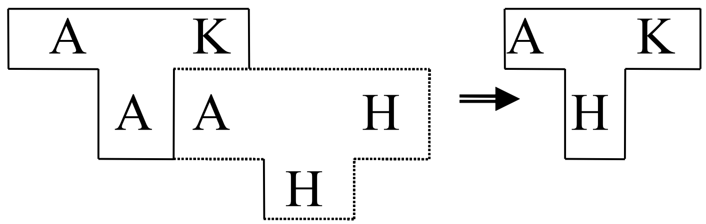
- "quick and dirty" compiler written in machine language H
- Compiler written in its own language A
- Result in running but inefficient compiler

The Second step in bootstrap



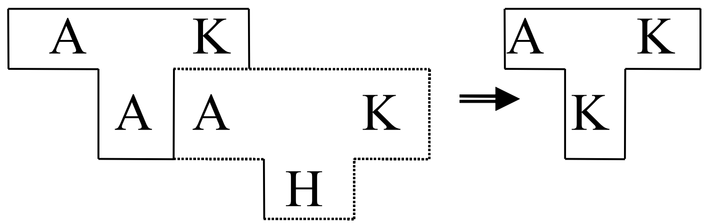
- Running but inefficient compiler
- Compiler written in its own language A
- Result in final version of the compiler

The step 1 in porting



- Original compiler
- Compiler source code retargeted to K
- Result in Cross Compiler

The step 2 in porting



- Cross compiler
- Compiler source code retargeted to K
- Result in Retargeted Compiler

End of Chapter One
Thanks

Mar. 1st , 2013