Compiler Principle and Technology

2022 Spring&Summer

Chapter 1 introduction

Outline

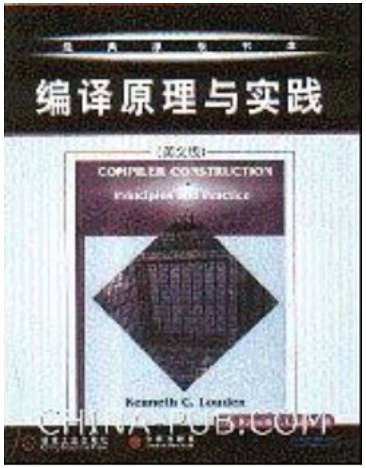
- Course Organization
  - – General course information
  - – Homework
- Introduction to Compilers
  - Why do we need compilers?
  - What are compilers?
  - General compiler structure

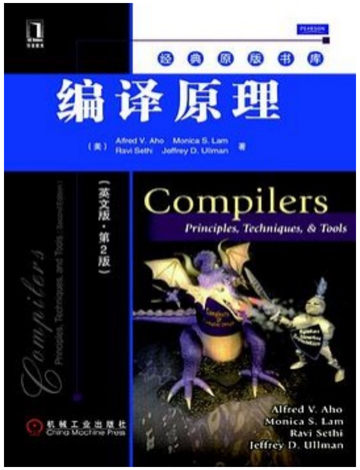
General course information

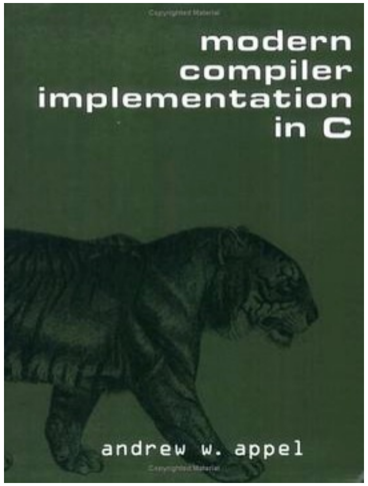
Teacher	Li, Ying(李莹), cnliying@zju.edu.cn
Teaching assistant	Zhou, Bowen(周博文), discord9@163.com

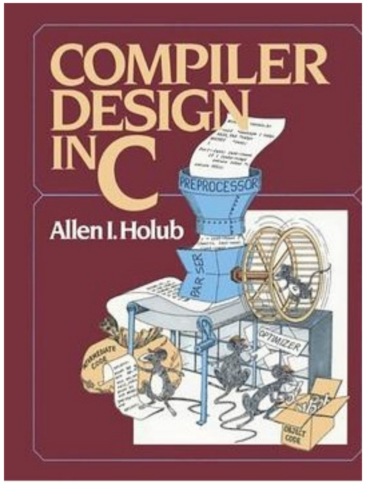
Reference book

- Compiler Construction Principles and Practice, Kenneth C. Louden
- Compilers -- Principles, Techniques and Tools,(Dragon Book), by Aho, Sethi and Ullman (1986)
- Modern Compiler Implementation in C(Java,ML), by,Andrew Appel (2002)
- Compiler Design in C. Prentice Hall, Allen I. Holub (1990)









Work Distribution

– Homework = 10%

– Class Quizzes = 10% （online）

– Mid-term Exam = 15% （online）

– Project = 25%

– Final Exam = 40%

Note: Final Exam < 40/100 ➔ Final Grade < 60/100

Chapter 1 introduction

Why take this course?

Understand compilers/languages:

- Understand the code structure
- Understand the language semantics
- Understand the relation between source code and generated machine code
- Become a better programmer

Why take this course? (Ctd.)

Nice balance of theory and practice:

Theory:

- Lots of mathematical models: regular expressions
- automata, grammars, graphs
- Lots of algorithms that use these models

Practice:

- Apply theoretical notions to build a real compiler
- Better understand why “theory and practice are the same in theory; in practice they are different”

Why take this course? (Ctd.)

Programming experience

- Write a large program that manipulates complex data structures
- Learn how to be a better programmer in groups
- Learn more about c/c++ and Intel x86 architecture and assembly language

Chapter 1 introduction

Compilers : computer languages

- Translate one language to another
- Source language(input) to target language (output)
- Source language : high-level language c or c++
- Target language : object code, machine code (machine instruction)

1.1 A brief history of compiler

- In the late1940s, programs were written in machine language， c7 06 0000 0002

- Assembly language ： numeric codes were replaced symbolic forms. Mov x, 2
  - asssembler ： translate the symbolic codes and memory location of assembly language into the corresponding numeric codes.
  - Defects of the assembly language： difficult to read 、write and understanding 、dependent on the particular machine

1.1 A brief history of compiler

- FORTRAN language and its compiler: between 1954 and 1957 , developed by the team at IBM , John Backus.
  - The structure of natural language studied by Noam Chomsky,
  - The classification of languages according to the *complexity of their grammars* and *the power of the algorithms* needed to recognize them.
  - Four levels of grammars: type 0 、type 1、type2 and type3 grammars

1.1 A brief history of compiler

- Parsing problem : studied in 1960s and 1970s
- Code improvement techniques(optimization techniques): improve compilers efficiency
- Compiler-compilers(parser generator): only in one part of the compiler process.
- YACC written in 1975 by Steve Johnson for the UNIX system. Lex written in 1975 by Mike Lest.
- Recent advances in compiler design:

1.2 Programs related to compilers

- Interpreters

  A language translator. It executes the source program immediately.

- Assemblers

  A translator translates assembly language into object code

- Linkers

  - Collects code separately compiled or assembled in different object files into a file.
  - Connects the code for standard library functions.
  - Connects resources supplied by the operating system of the computer.

1.2 Programs related to compilers

- Loaders

  Loaders resolve all relocatable address relative to the starting address.

- Preprocessors

  Delete comments, include other files, perform macro substitutions.

- Editors

  Produce a standard file (structure based editors)

- Debuggers

  Determine execution errors in a compiled program.

1.2 Programs related to compilers

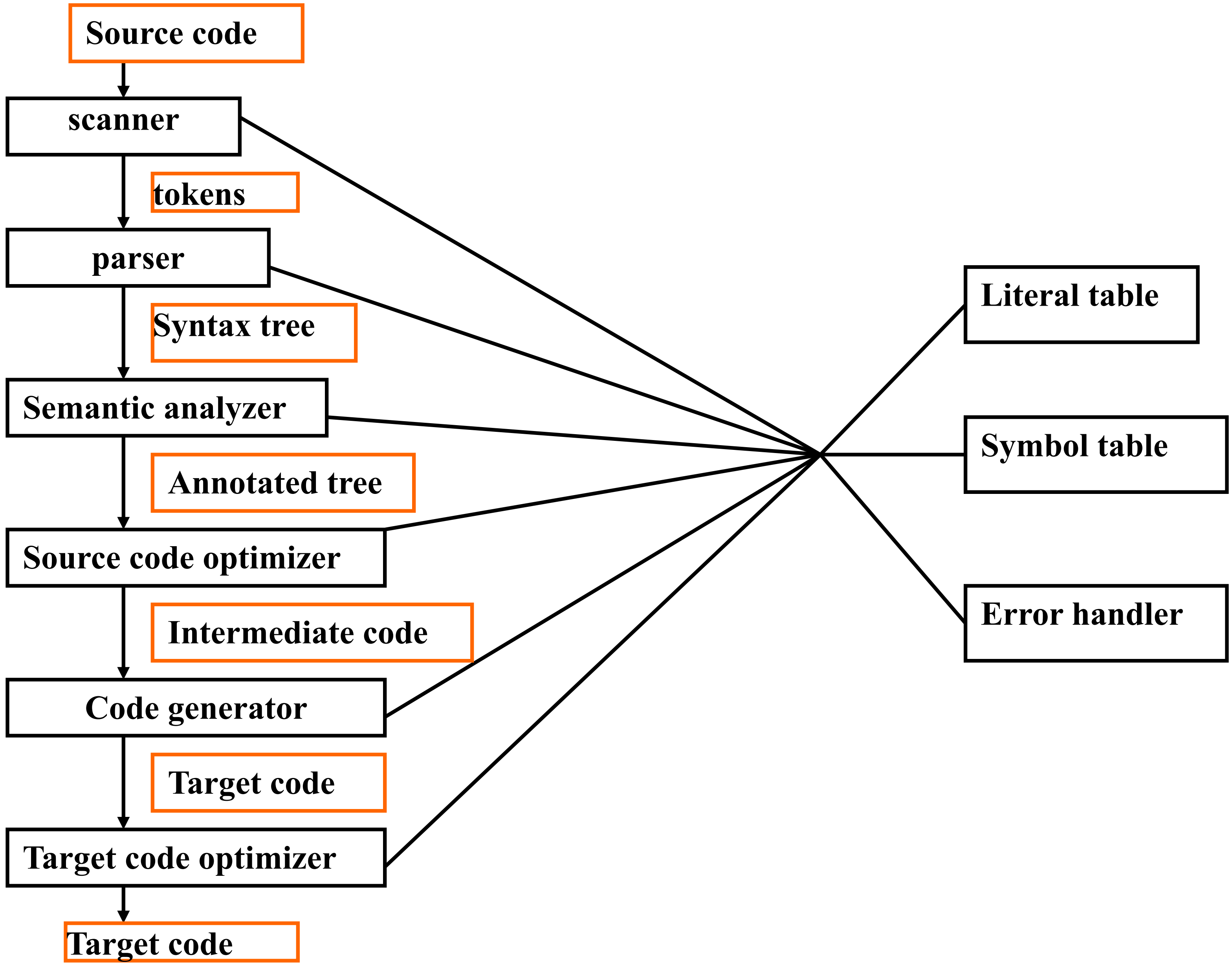
- Profilers

  Collect statistics on the behavior of an object program during execution.

- Project managers
  - Coordinate the files being worked on by different people.
  - SCCS (source code control system)
  - RCS (revision control system)

  project manager programs on Unix systems.

1.3 The translation process



Source code

scanner

tokens

parser

Syntax tree

Semantic analyzer

Annotated tree

Source code optimizer

Intermediate code

Code generator

Target code

Target code optimizer

Target code

Literal table

Symbol table

Error handler

1.3 The translation process

1. The scanner

Lexical analysis: input a stream of characters, output tokens

Example: a [index] = 4+2

Output:

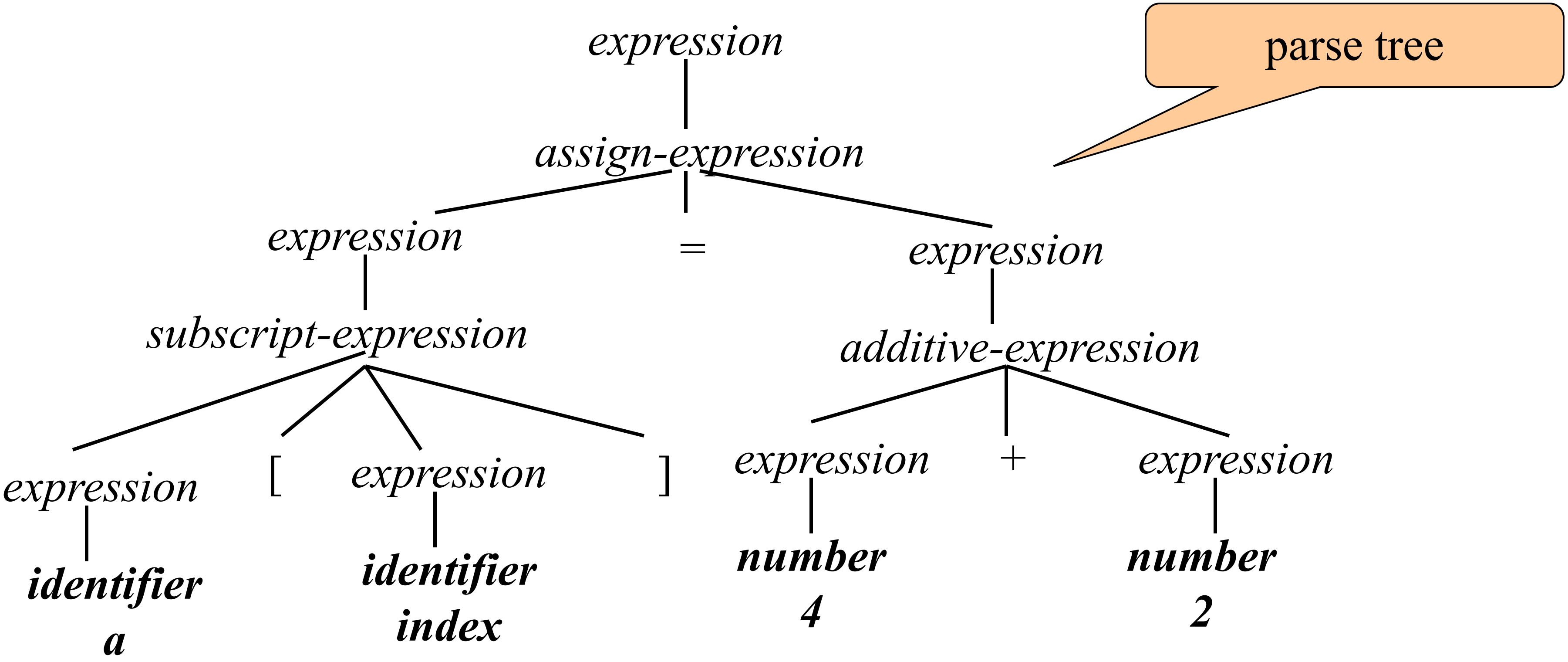
a	identifier
[	left bracket
index	identifier
]	right bracket
=	assignment
4	number
+	plus sign
2	number

other operations: enter identifiers into the symbol table
enter literals into the literal table

1.3 The translation process

2. The parser

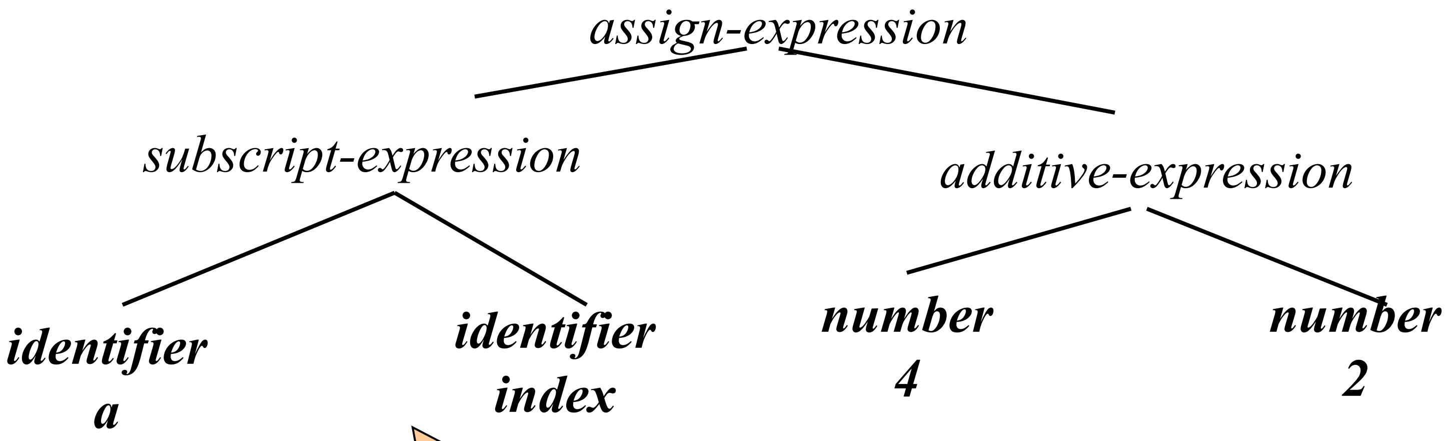
- Determine the structure of the program
- Input : the forms of tokens
- Output : a parse tree or a syntax tree



parse tree

1.3 The translation process

2. The parser



assign-expression

subscript-expression

additive-expression

identifier
a

identifier
index

number
4

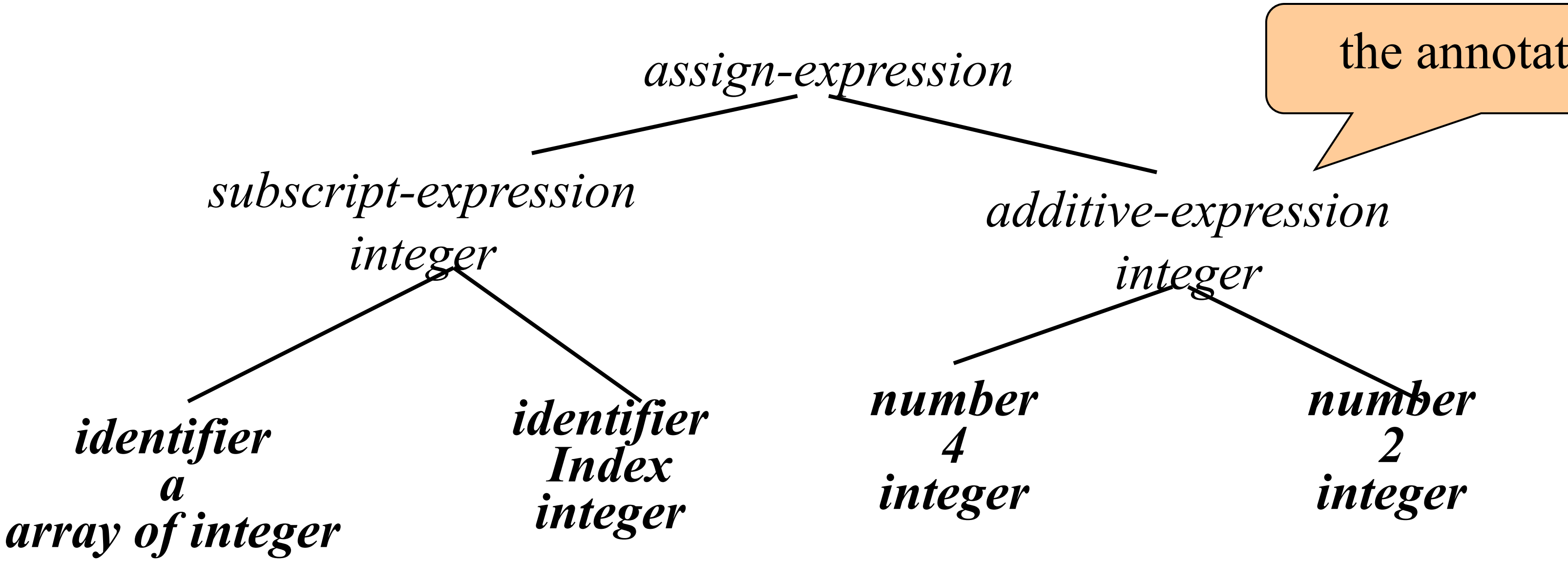
number
2

syntax tree (abstract syntax tree): a condensation of the information contained in the parse tree.

1.3 The translation process

3. The semantic analyzer

- Static semantics: including declarations and type checking
- Dynamic semantics
- attribute: computed by the semantic analyzer



the annotated tree

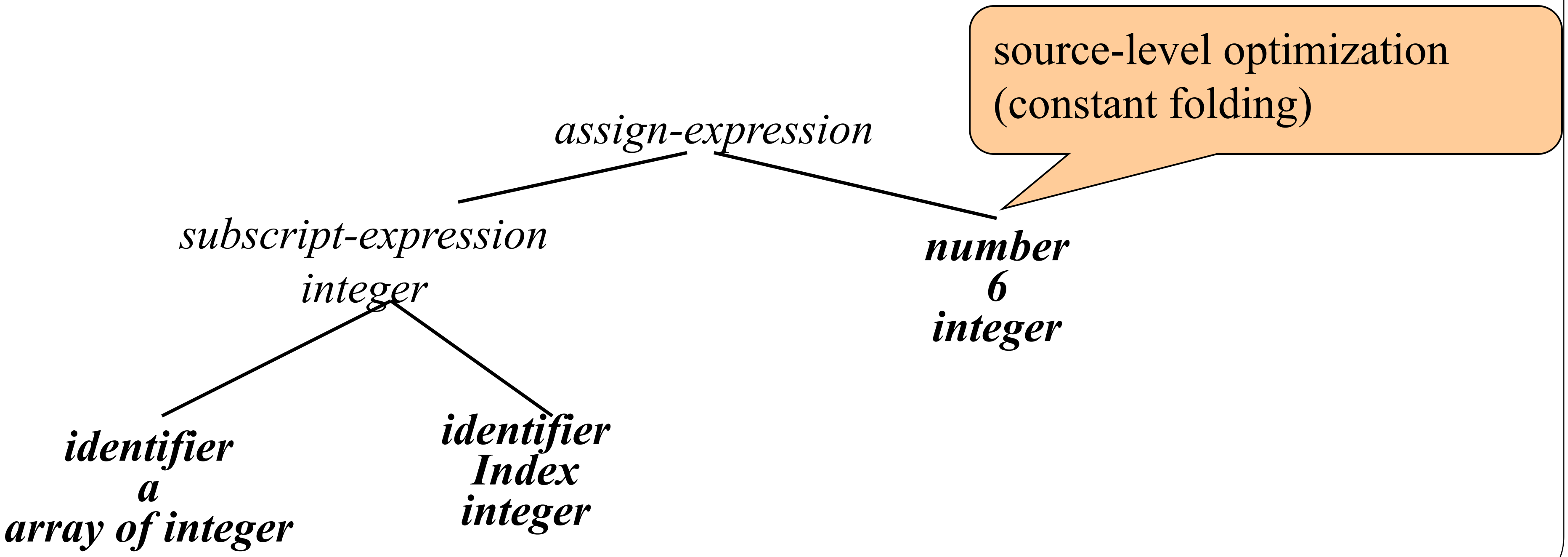
1.3 The translation process

4.The source code optimizer

Source-level optimization

intermediate code: Three–address code

Individual compilers exhibit a wide variation not only in the kinds of optimizations performed but also in the placement of the optimization phases.



source-level optimization (constant folding)

assign-expression

subscript-expression
integer

number
6
integer

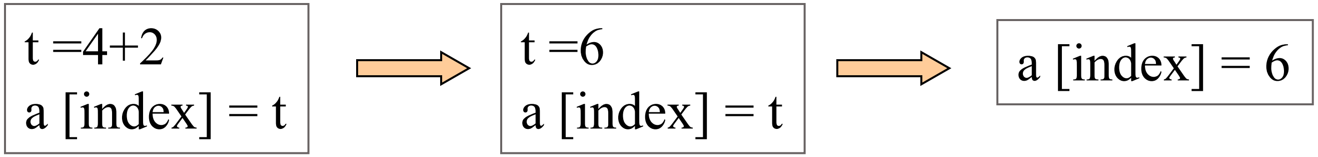
identifier
a
array of integer

identifier
Index
integer

1.3 The translation process

4.The source code optimizer

intermediate code: Three–address code or P-code



t =4+2
a [index] = t

t =6
a [index] = t

a [index] = 6

intermediate code or IR: a form of code representation intermediate between source code and object code.

syntax tree
three address code
p-code

1.3 The translation process

5. The code generator

Input : intermediate code or IR

Output: machine code, code for the target machine

```
MOV  R0,  index     ;  ;  value of index → R0
MUL   R0,  2        ;  ;  double value in R0
MOV  R1,  &a        ;  ;  address of a → R1
ADD   R1, R0        ;  ;  add R0 to R1
MOV  *R1, 6         ;  ;  constant 6 → address in R1
```

The machine performs byte addressing and that integers occupy two bytes of memory.

1.3 The translation process

6. The target code optimizer

Improve the target code generated by the code generator

choosing addressing modes to improve performance

replacing slow instructions by faster ones

eliminating redundant or unnecessary operations

```
MOV  R0,  index    ;  ;  value of index → R0
MUL   R0,  2       ;  ;  double value in R0
MOV  R1,  &a       ;  ;  address of a → R1
ADD   R1, R0       ;  ;  add R0 to R1
MOV  *R1, 6        ;  ;  constant 6 → address in R1
```

↓

```
MOV  R0,  index    ;  ;  value of index → R0
SHL   R0           ;  ;  double value in R0
MOV  &a[R0], 6     ;  ;  constant 6 → address a + R0
```

1.4 Major data structures in a compiler

1. **Tokens**

a value of an enumerated data type   the sets of tokens

2. **Syntax tree**

each node is a record whose fields represent the information collected by the parser and semantic analyzer

3. **Symbol table**

information associated with identifiers: functions, variables, constants, and data types.

4. Literal table

store constants and strings

5. Intermediate code

this code kept as an array of text strings, a temporary text file, or as a linked list of structures.

6. **Temporary files**

using temporary files to hold the products of intermediate steps

1.5 Other issues in compiler structure

- Different angles

  analysis and synthesis

  - ➢ analysis: lexical analysis 、syntax analysis、semantic analysis (optimization)
  - ➢ synthesis: code generation (optimization)

- Front end and back end

  separation depend on the source language or the target language

  - ➢ the front end: the scanner、parser、semantic analyzer, intermediate code synthesis
  - ➢ the back end: the code generator, some optimization

1.5 Other issues in compiler structure

- Passes
  - process the entire source program several times
  - a pass consist of several phases
- Language definition and compilers
  - relation between the language definition and compiler
- Compiler options and interfaces
  - interfaces with the operating system
  - provide options to the user for various purposes
- Error handling: static error, execution error

1.7 The tiny sample language and compiler

- Language TINY : as a running example (as a source language)
- Target language: assembly language (TM machine, tiny machine)