Introduction

Sung-Dong Kim

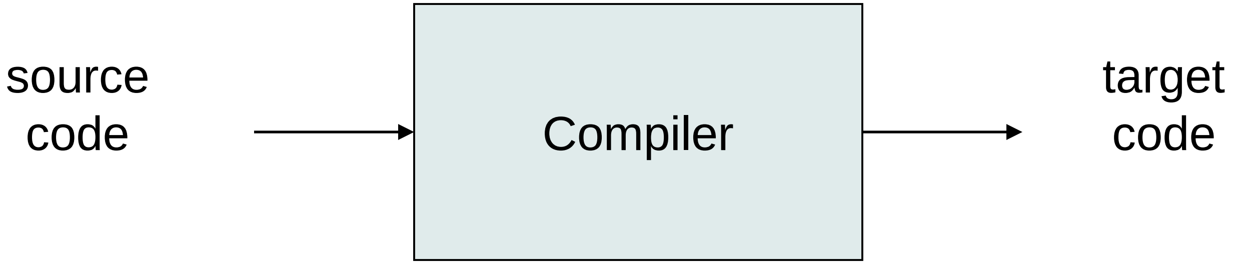
Dept. of Computer Engineering,

Hansung University

Edited by Mohammad Rawashdeh

Definition

- Programs that translate one language to another
  - Source language, source code: C, C++, …
  - Target language, target code: machine code, object code



Significance

- Compiler is a fairly complex
- Command interpreter, interface programs
  - Smaller than compilers
  - Same techniques

Objects

- Basic knowledge
- Necessary tools
- Practical experience

Requirements (1)

- Theoretical techniques
  - Automata theory: 2.2, 2.3, 2.4, 3.2
  - Data structures
  - Discrete mathematics
  - Machine architecture
  - Assembly language

Requirements (2)

- Practical coding
  - Planning
- Interaction between
  - Structure of a compiler
  - Design of the programming language

Sample languages

- TINY
- C-Minus

1.1 Why Compiler? A Brief History (1)

- Machine language
  - Numeric code
  - C7 06 0000 0002 (8x86 processors)
- Assembly language
  - Symbolic forms
  - MOV X, 2
  - Assembler

XOR CL, [12H] = 00110010 00001110 00010010 00000000 = 32H 0EH 12H 00H

1.1 Why Compiler? A Brief History (2)

- High-level programming language
  - Machine-independent
  - Mathematical notation, natural language
  - X = 2

1.1 Why Compiler? A Brief History (3)

- Chomsky
  - Chomsky hierarchy: type 0 ~ 3
  - Grammar and algorithms needed to recognize
  - Context-free grammar: type 2 (Chapter 3, 4, 5)
  - Finite automata, regular expression: type 3 (Ch. 2)
- Code improvement techniques
  - Generating efficient object code
  - Optimization techniques → wrong

1.1 Why Compiler? A Brief History (4)

- Parser generator
  - YACC (Ch. 5)
- Scanner generator
  - LEX (Ch. 2)
- Recent advances
  - Development of more sophisticated PL
  - Part of interactive development environment (IDE)

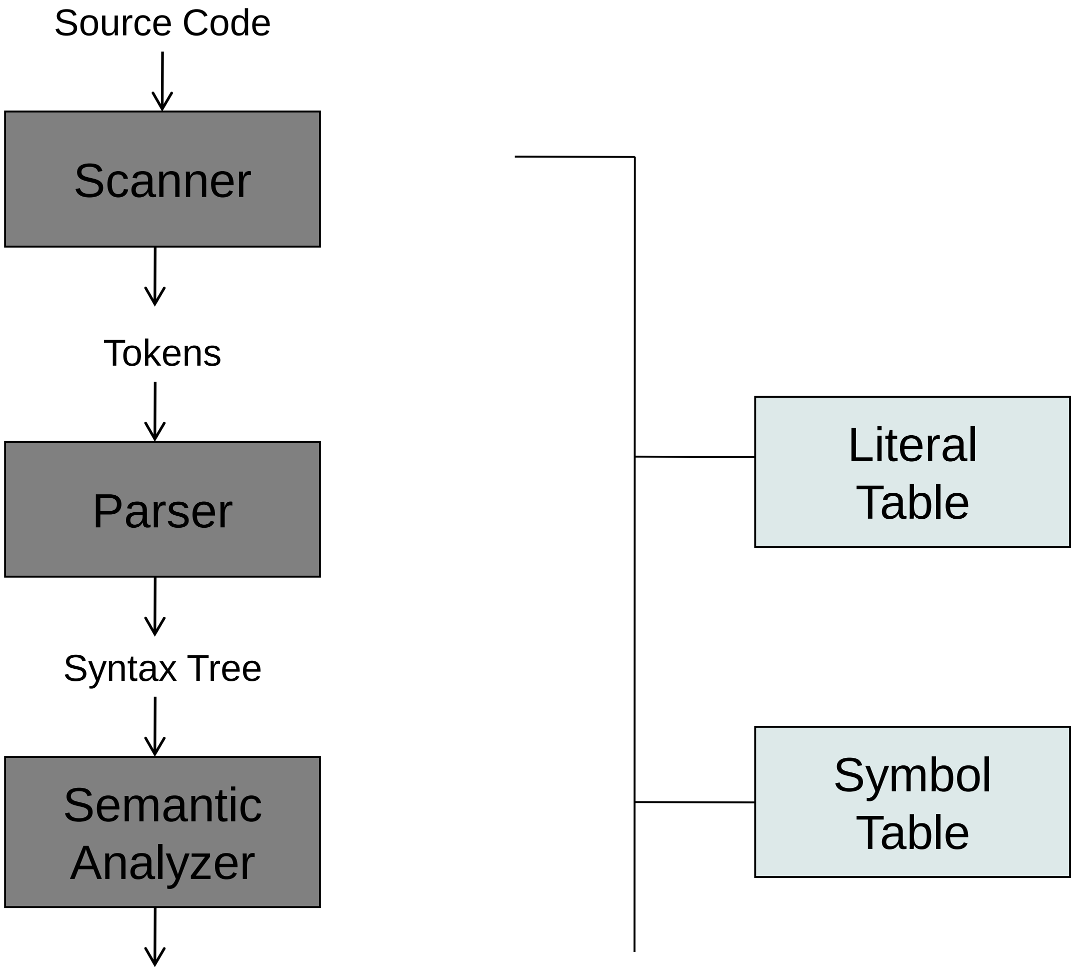
1.2 Programs related to Compilers (1)

- Interpreters
- Assemblers
- Linkers
- Loaders
- Preprocessors
- Editors

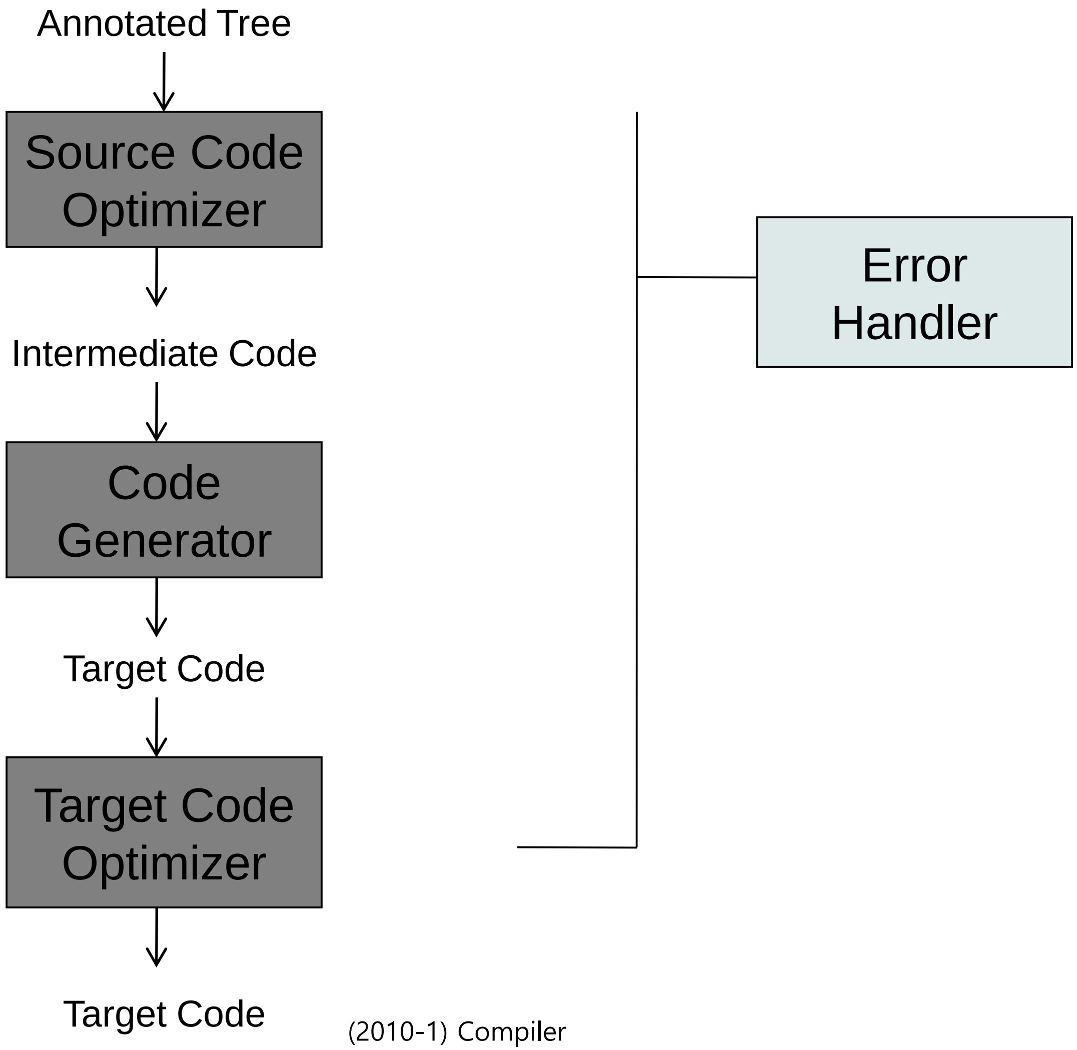
1.2 Programs related to Compilers (2)

- Debuggers
- Profilers
- Project managers

1.3 Translation Process (1)



1.3 Translation Process (2)



Annotated Tree

Source Code Optimizer

Intermediate Code

Code Generator

Target Code

Target Code Optimizer

Target Code

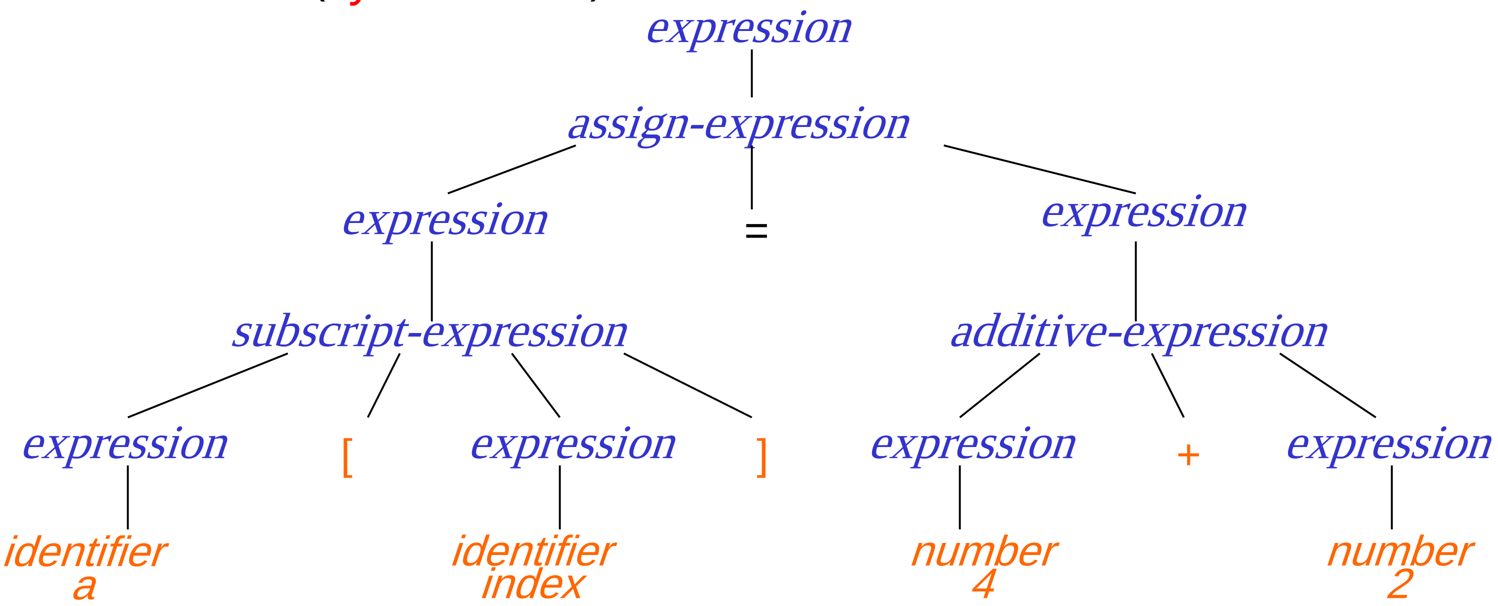
Error Handler

1.3 Translation Process (3)

- Scanner
  - Lexical analysis
  - Stream of character → token
  - Example: a[index] = 4 + 2
    - a: identifier → symbol table
    - [: left bracket
    - index: identifier
    -]: right bracket
    - =: assignment
    - 4: number → literal table (ex: "ksd")
    - +: plus sign
    - 2: number

1.3 Translation Process (4)

- Parser
  - Syntax analysis
  - Structure of the program ← grammatical analysis
  - Parse tree (syntax tree)



expression

assign-expression

expression = expression

subscript-expression

additive-expression

expression [expression]

expression + expression

identifier a

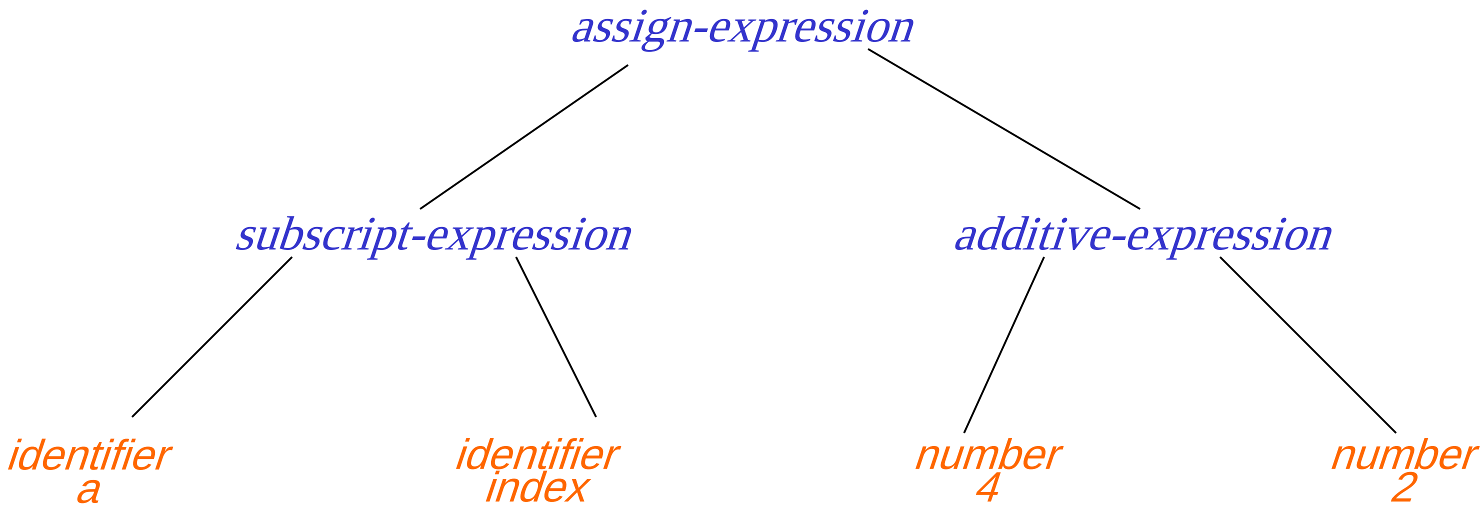
identifier index

number 4

number 2

1.3 Translation Process (5)

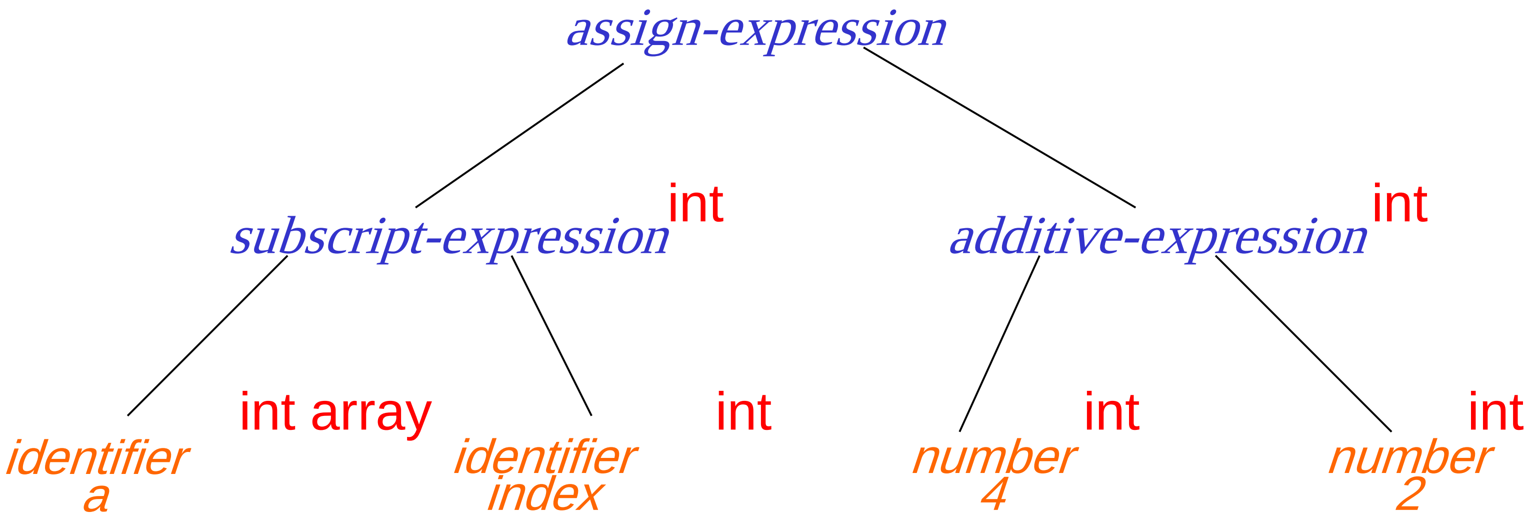
- Abstract syntax tree



1.3 Translation Process (6)

- Semantic analyzer
  - Analysis of the static semantics
    - Declarations
    - Type checking
  - Dynamic semantics → only in execution
  - Attributes: extra pieces of information

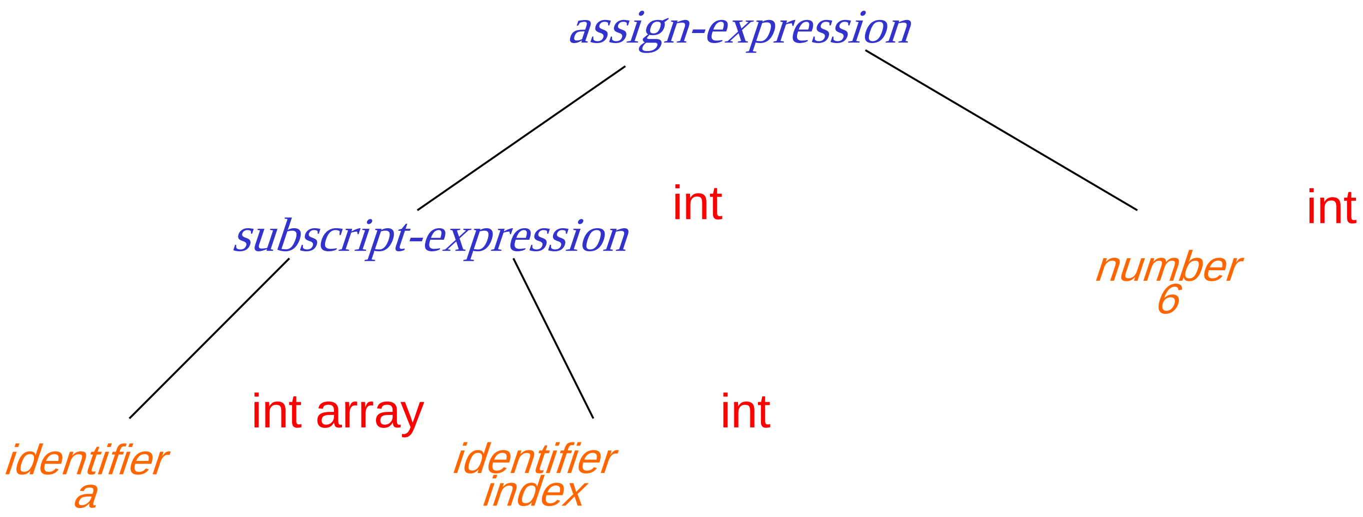
1.3 Translation Process (7)



Annotated tree

1.3 Translation Process (8)

- Source code optimizer
  - Wide variation in the kinds of optimization and the placement of the optimization phase
  - Constant folding: 4 + 2 → 6



1.3 Translation Process (9)

- Intermediate code, intermediate representation
  - Three-address code
    - t = 4 + 2, a[index] = t
    - t = 6, a[index] = t
    - a[index] = 6
  - P-code
  - Syntax tree

1.3 Translation Process (10)

- Code generator
  - Target machine’s instruction
  - Data representation
  - Ex: assembly code

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

1.3 Translation Process (11)

- Target code optimizer
  - Addressing mode
  - Faster instruction
  - Redundant code, unnecessary operations

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

1.4 Major Data Structures (1)

- Role
  - Necessary for the phase as part of operation
  - Serve to communicate information among the phases
- Tokens
  - Generated by scanner
  - Collection of characters → token
  - other information
    - String of characters: name of an identifier token
    - Value of number token

1.4 Major Data Structures (2)

- Syntax tree
  - Pointer-based structure dynamically allocated
  - Single variable → root node
  - Information collected by parser and semantic analyzer
- Symbol table
  - Information associated with identifier
  - Frequently accessed
  - Hash table, list, stack, …

1.4 Major Data Structures (3)

- Literal table
  - Constants, strings
  - Quick insertion and lookup
- Intermediate code
  - Array of text strings
  - Temporary text file
  - Linked list of structures

1.4 Major Data Structures (4)

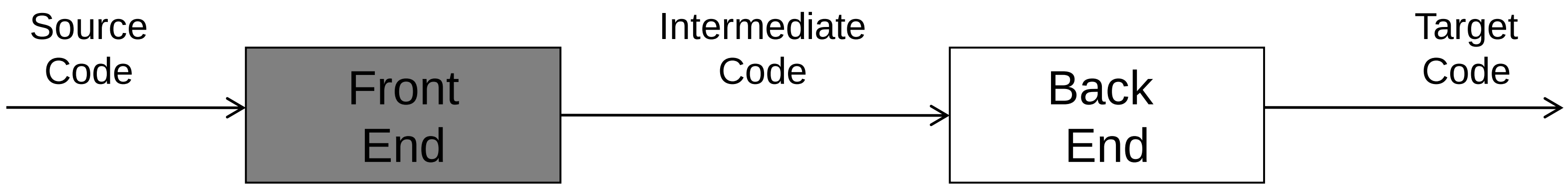
- Temporary files
  - Due to not enough memory
  - For backpatching address during code generation

1.5 Other issues (1)

- Structure of compiler is very important
  - Reliability
  - Efficiency
  - Usefulness
  - Maintainability
- Analysis and synthesis
  - Analysis: operations that analyze the source program
  - Synthesis: operations involved in producing translated code

1.5 Other issues (2)

- Front end and back end
    - Front end: operations depending on the source language
    - Back end: operations depending on the target language



1.5 Other issues (3)

- Passes
  - One pass
    - Efficient compilation
    - Less efficient target code
  - Multiple passes
    - Scanning, parsing
    - Semantic analysis, source-level optimization
    - Code generation, target-level optimization
  - More passes

1.5 Other issues (4)

- Language definition and compilers
  - Language reference manual (language definition)
    - Formal lexical and syntactic structure
      - Regular expression
      - Context-free grammar
    - Semantics of programming languages
      - English
  - Language standard
    - ANSI C, PASCAL, FORTRAN
    - Test suite: standard test programs

1.5 Other issues (5)

- TYNY
  - Lexical structure: 2.5
  - Syntactic structure: 3.7
  - Semantic structure: 6.5
- C-Minus
  - Appendix A
- Denotational semantics
  - Formal definition for semantics by mathematical terms
  - Functional programming community

1.5 Other issues (6)

- Runtime environment
  - Structure of data allowed
  - Kinds of function calls and returned values allowed
  - 3 basic types of runtime environments
    - Static
    - Semi-dynamic (stack-based)
    - Fully-dynamic

1.5 Other issues (7)

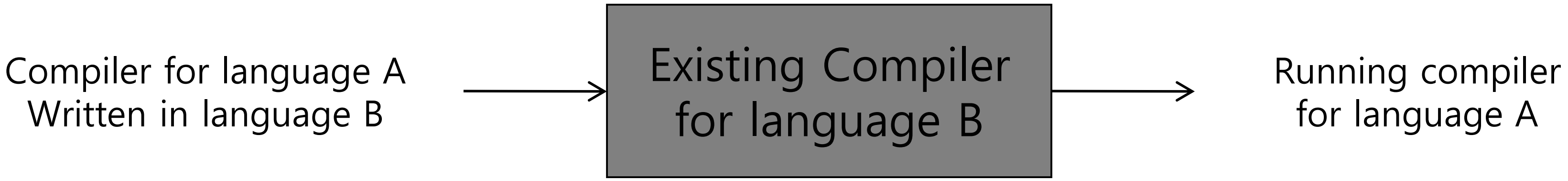
- Compiler options and interfaces = compiler pragmatics
  - Interfacing with OS
    - I/O
    - Access to file system
  - Providing options to the user
    - Listing characteristics
    - Code optimization

1.5 Other issues (8)

- Error handling
  - Static (compile-time) errors
  - Exception handling

1.6 Bootstrapping and porting (1)

- Implementation (host) language
  - Machine language
  - Execute immediately
- Cross compiler

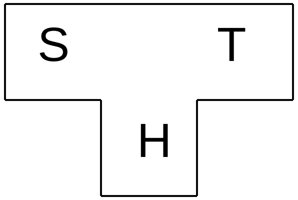


Compiler for language A
Written in language B

Existing Compiler
for language B

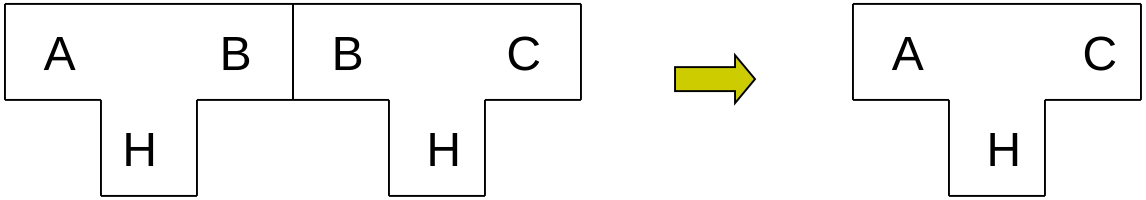
Running compiler
for language A

- T-diagram

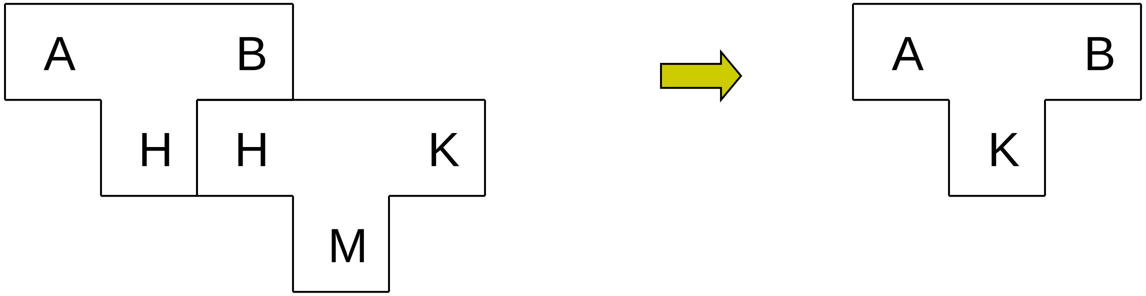


1.6 Bootstrapping and porting (2)

- Combination 1

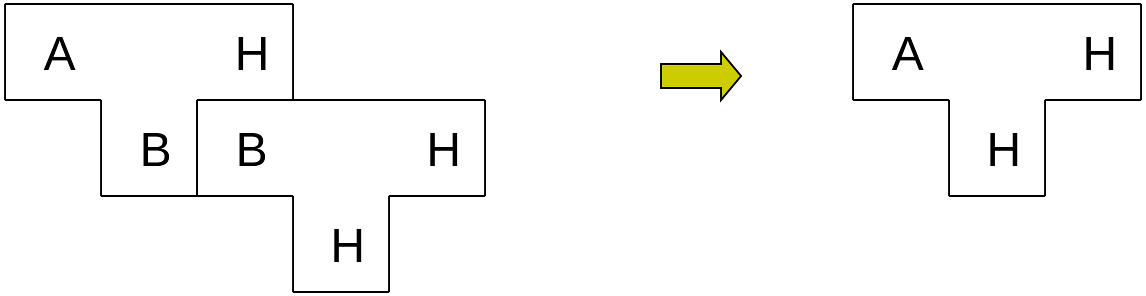


- Combination 2

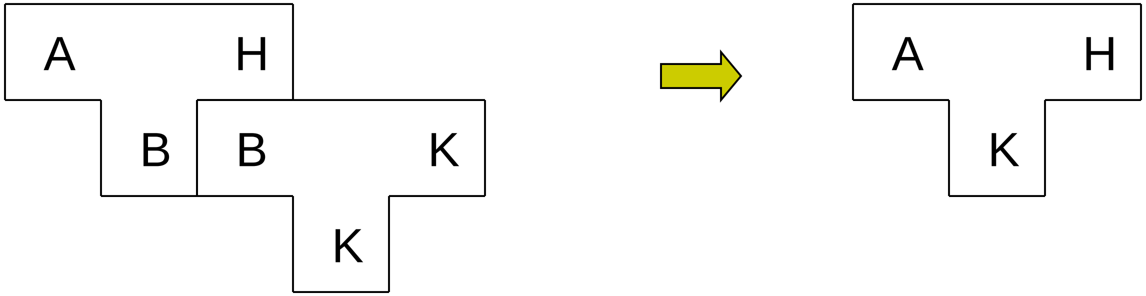


1.6 Bootstrapping and porting (3)

- Scenario 1

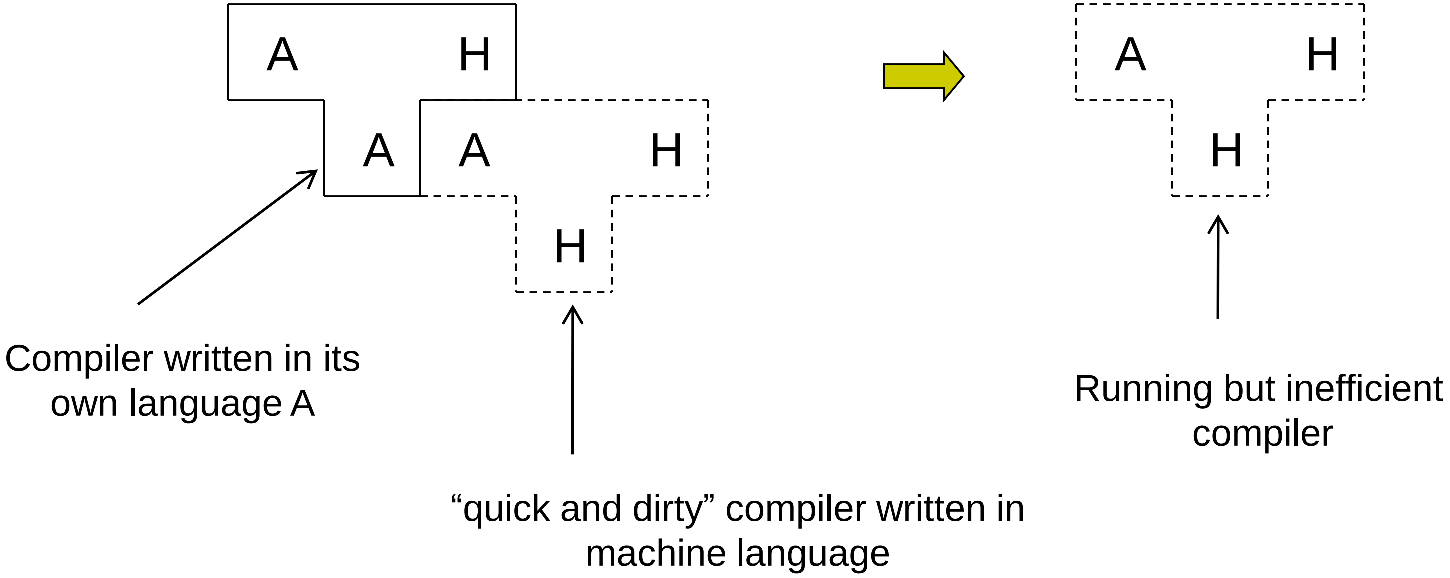


- Scenario 2: cross compiler



1.6 Bootstrapping and porting (4)

- Bootstrapping
  - "quick and dirty" compiler in assembly language
    - Inefficient and correct
    - Compile "good" compiler



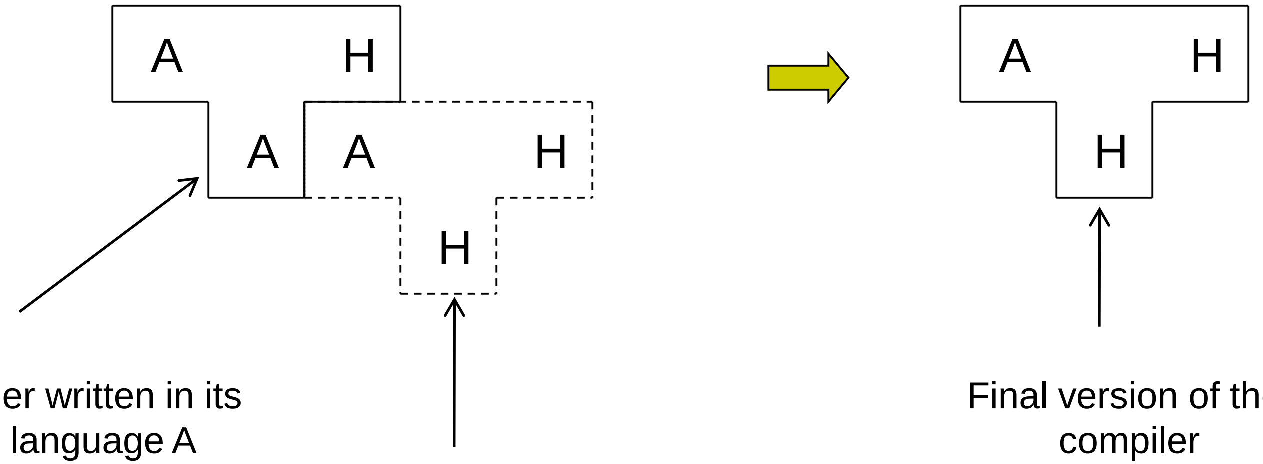
Compiler written in its own language A

"quick and dirty" compiler written in machine language

Running but inefficient compiler

1.6 Bootstrapping and porting (5)

- Recompile → final version



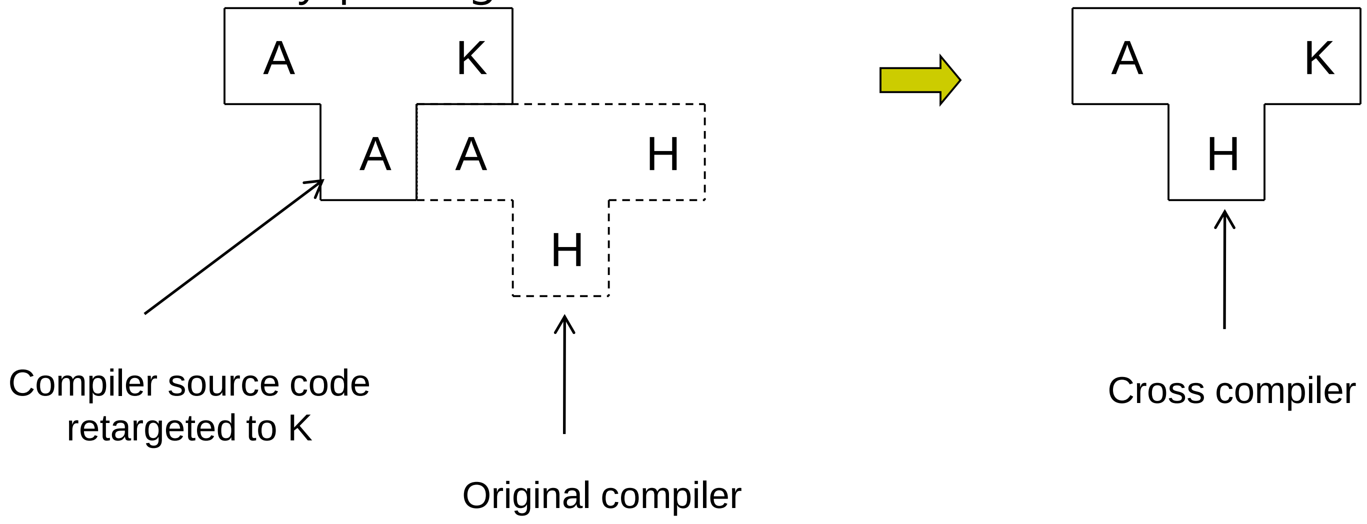
Compiler written in its own language A

Running but inefficient compiler

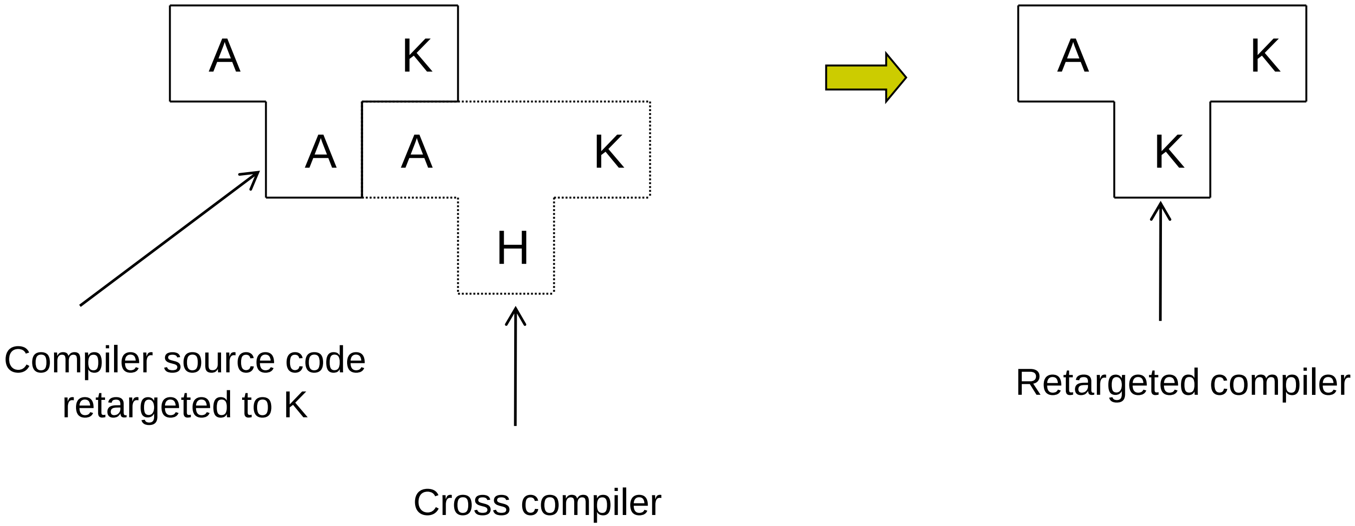
Final version of the compiler

1.6 Bootstrapping and porting (6)

- Advantages
  - Improvement to the source code of the compiler → working compiler
  - Easy porting



1.6 Bootstrapping and porting (7)



Compiler source code retargeted to K

Cross compiler

Retargeted compiler

1.7 TINY sample language (1)

- TINY language
  - Sequence of statements separated by semicolons
  - No procedures, no declarations
  - Integer variable only
  - Two control statements
    - if-statement: optional else (terminated by end)
    - repeat-statement
  - read/write statements
  - { comments }: cannot be nested

1.7 TINY sample language (2)

- Expressions
  - Boolean
    - Comparison of two arithmetic expressions: <, =
    - Only as tests in control statements: no boolean variables, assignment, I/O

1.7 TINY sample language (3)

- Integer arithmetic

```
read x;                {input an integer}
if x > 0 then          {don’t compute if x <= 0 }
  fact := 1;
  repeat
    fact := fact * x;
    x := x - 1;
  until x = 0;
  write fact           {output factorial of x }
end
```

1.7 TINY sample language (4)

- TINY compiler
  - File construction

globals.h	main.c
util.h	util.c
scan.h	scan.c
parse.h	parse.c
symtab.h	symtab.c
analyze.h	analyze.c
code.h	code.c
cgen.h	cgen.c

1.7 TINY sample language (5)

- 4 passes
  - First pass = scanner + parser
  - Second: constructing symbol table
  - Third: type checking
  - Final: code generator
- Central code
  - syntax tree = parse();
  - buildSymtab(syntaxTree);
  - typeCheck(syntaxTree);
  - codeGen(syntaxTree, codefile);

1.7 TINY sample language (6)

- Conditional compilation flags
  - NO_PARSE
  - NO_ANALYZE
  - NO_CODE
- Usage: tiny sample.tny → sample.tm

1.7 TINY sample language (7)

- Options
  - EchoSource
  - TraceScan
  - TraceParse
  - TraceAnalyze
  - TraceCode

1.7 TINY sample language (8)

- TM machine
  - Assembly language: target language
  - Translation example: a[index] =6

```
LDC 1,0(0)                load 0 into reg 1
* The following instruction assumes index is at location 10 in memory
LD  0,10(1)               load val at 10+R1 into R0
LDC 1,2(0)                load 2 into reg 1
MUL 0,1,0                 put R1*R0 into reg 1
LDC 1,0(0)                load 0 into reg 1
* The following instruction assumes a is at location 20 in memory
LDA 1,20(1)               load 20+R1 into R0
ADD 0,1,0                 put R1+R0 into R0
LDC 1,6(0)                load 6 into reg 1
ST  1,0(0)                store R1 at 0+R0
```

1.7 TINY sample language (9)

- Simulator
  - Reads the assembly code from a file
  - Execute it
  - tm sample.tm

1.8 C-Minus (1)

- More extensive language than TINY
- Considerably restricted subset of C
  - Integers
  - Integer arrays
  - Functions (procedure, void function)
  - Local, global declarations
  - Recursive functions
  - if-, while- statement

1.8 C-Minus (2)

- Sample program

```
int fact(int x)
/* recursive factorial function */
{ if (x > 1)
    return x * fact(x-1);
  else
    return 1;
}

void main(void)
{ int x;
  x = read();
  if (x > 0) write(fact(x));
}
```