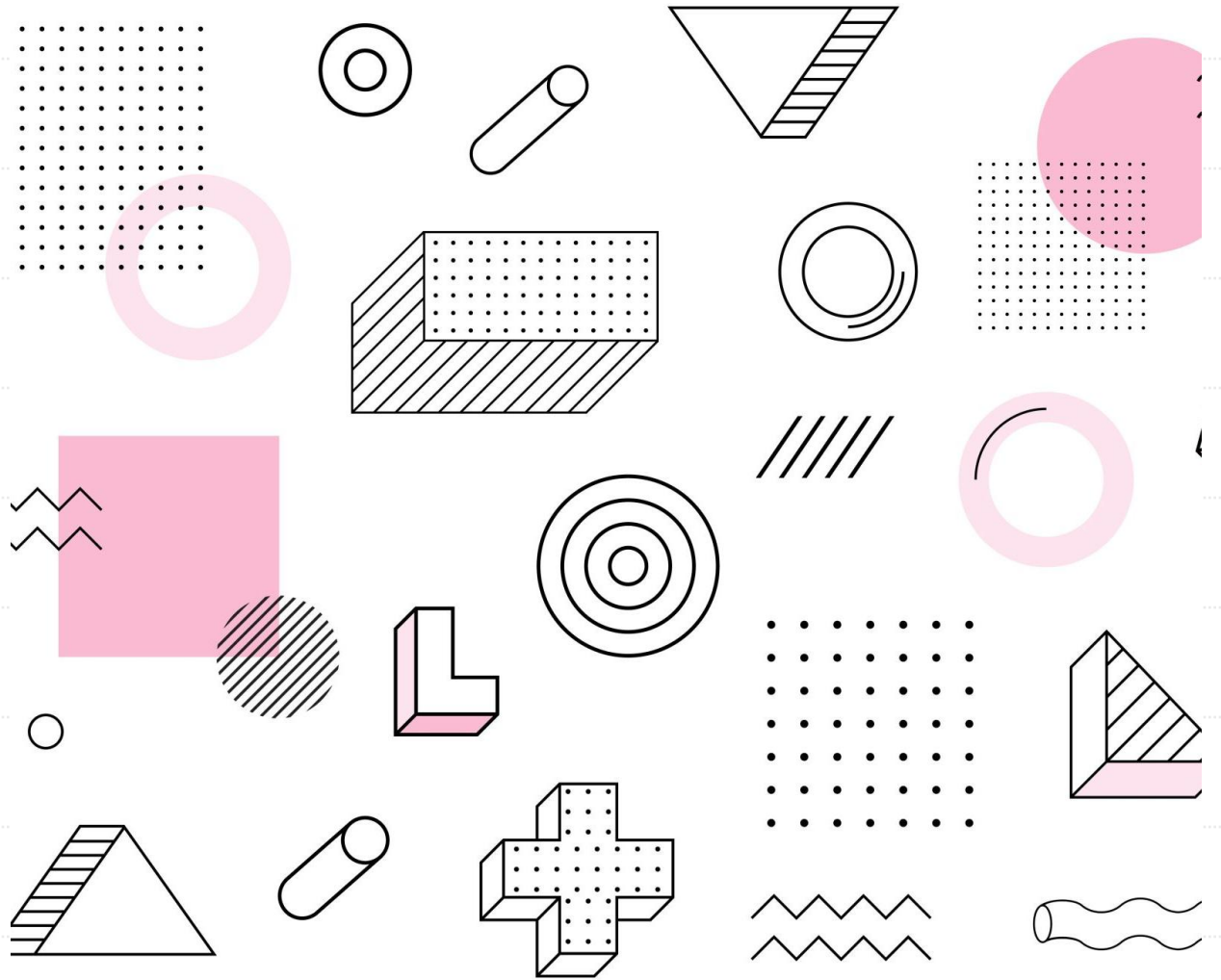


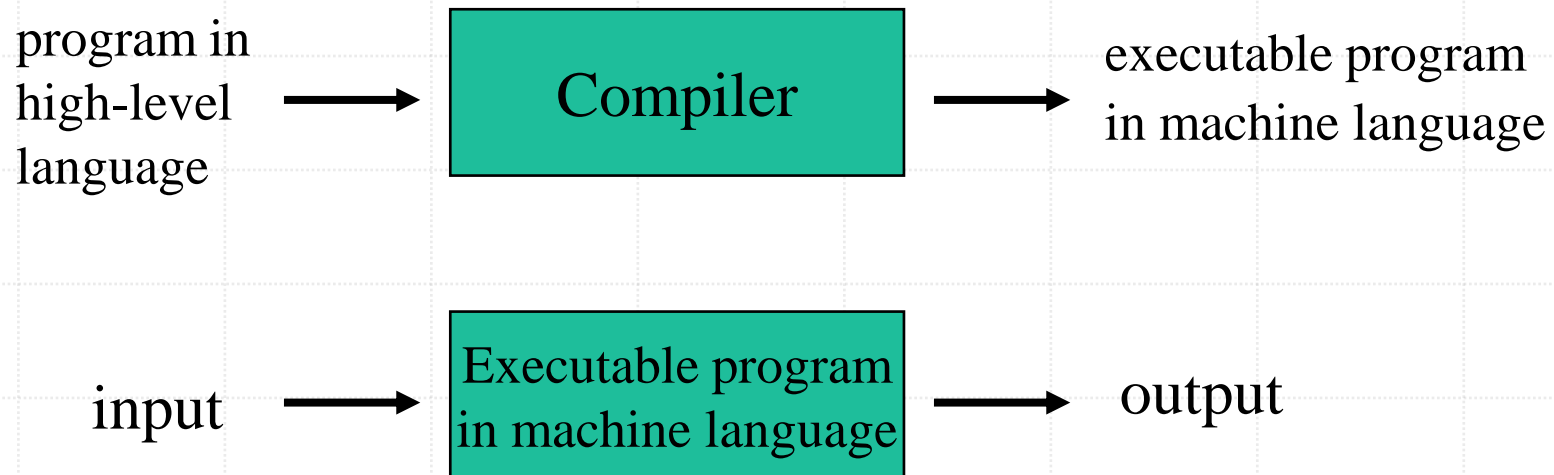
Compiler Construction 編譯系統

陳奇業 成功大學資訊工程系



Definition

- A compiler is an executable program that can read a program in one high-level language and translate it into an equivalent executable program in machine language.



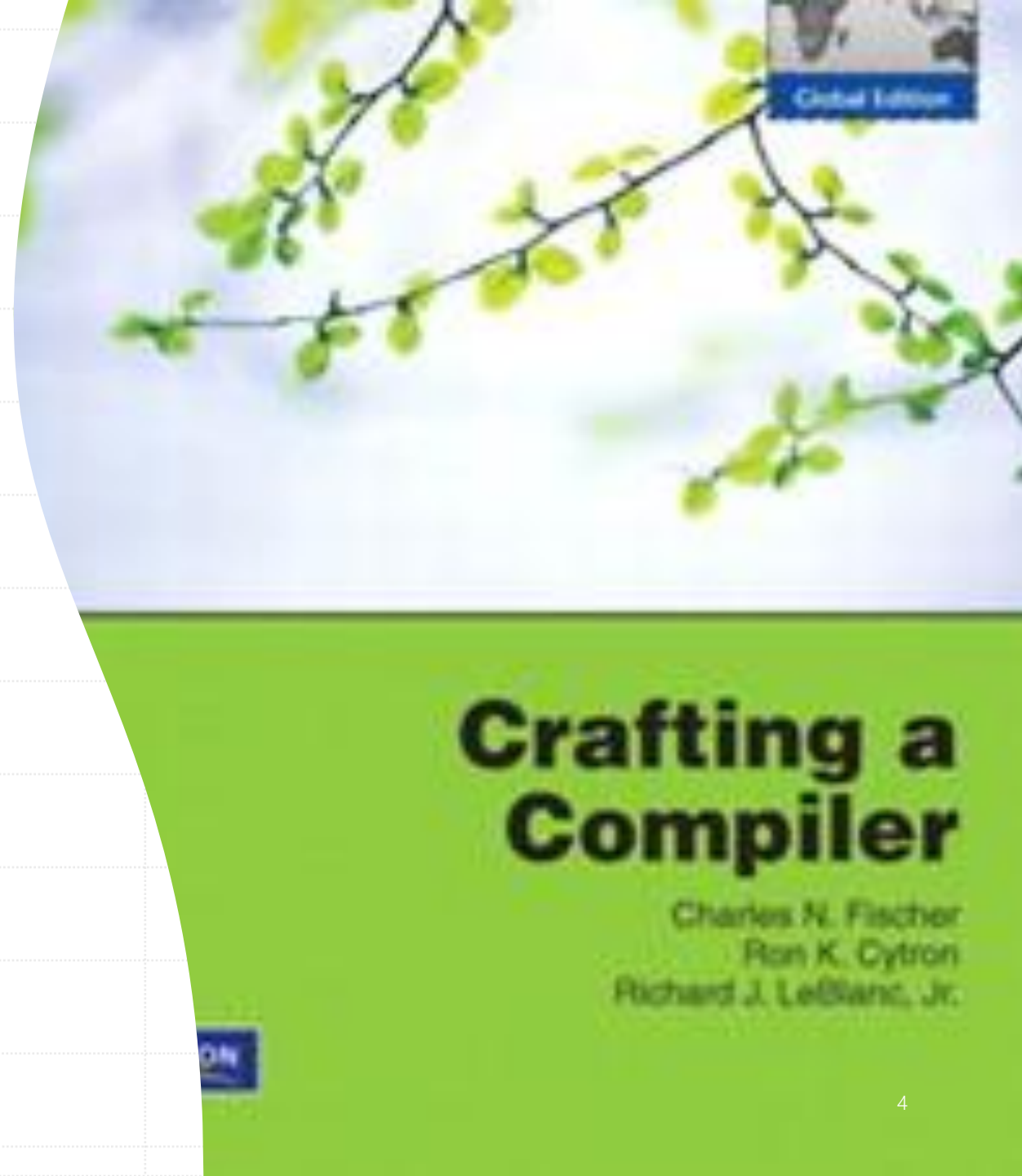


Grading

- Assignments (40%)
- Quizzes (20%)
- Mid-term Exam (20%)
- Final Exam (20%)

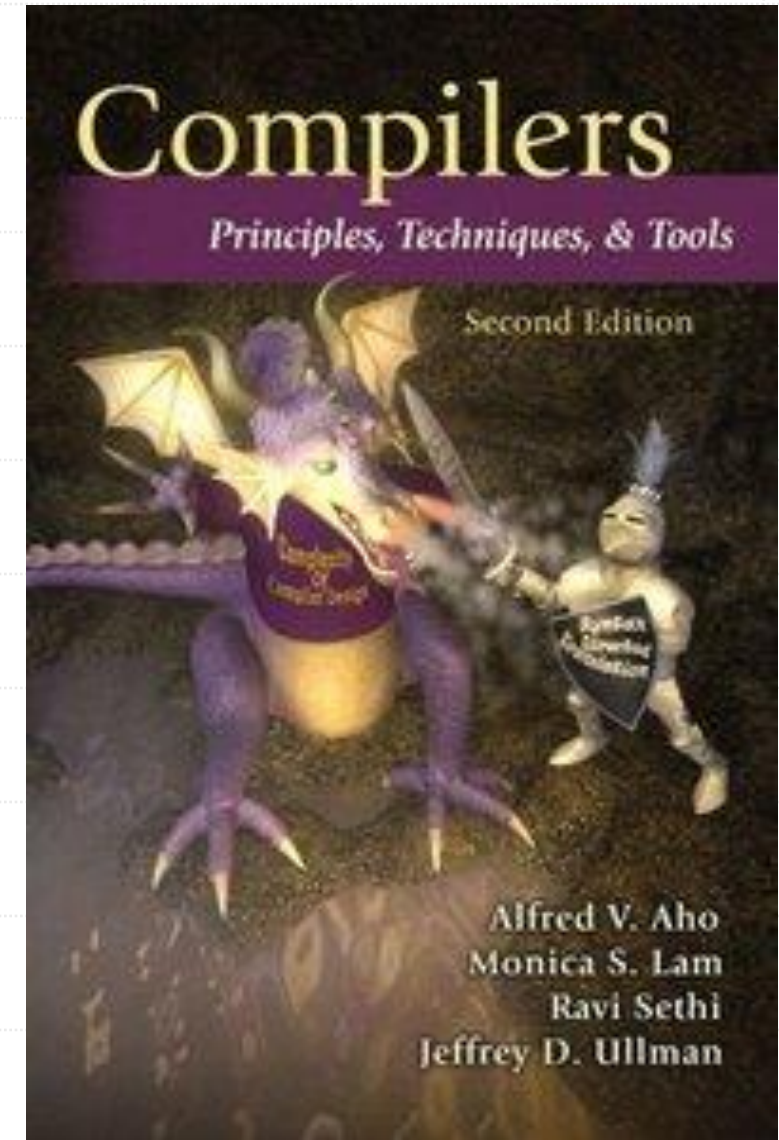
Course Material

- **Crafting a Compiler**, Fischer, Cytron, and LeBlanc, 0138017859



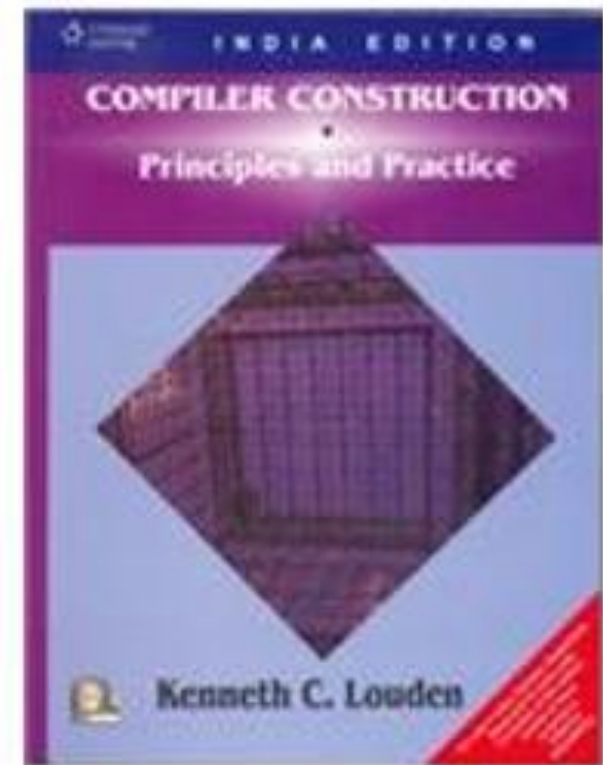
References

- **Compilers: Principles, Techniques, and Tools,**
Aho, Lam, Sethi, and Ullman



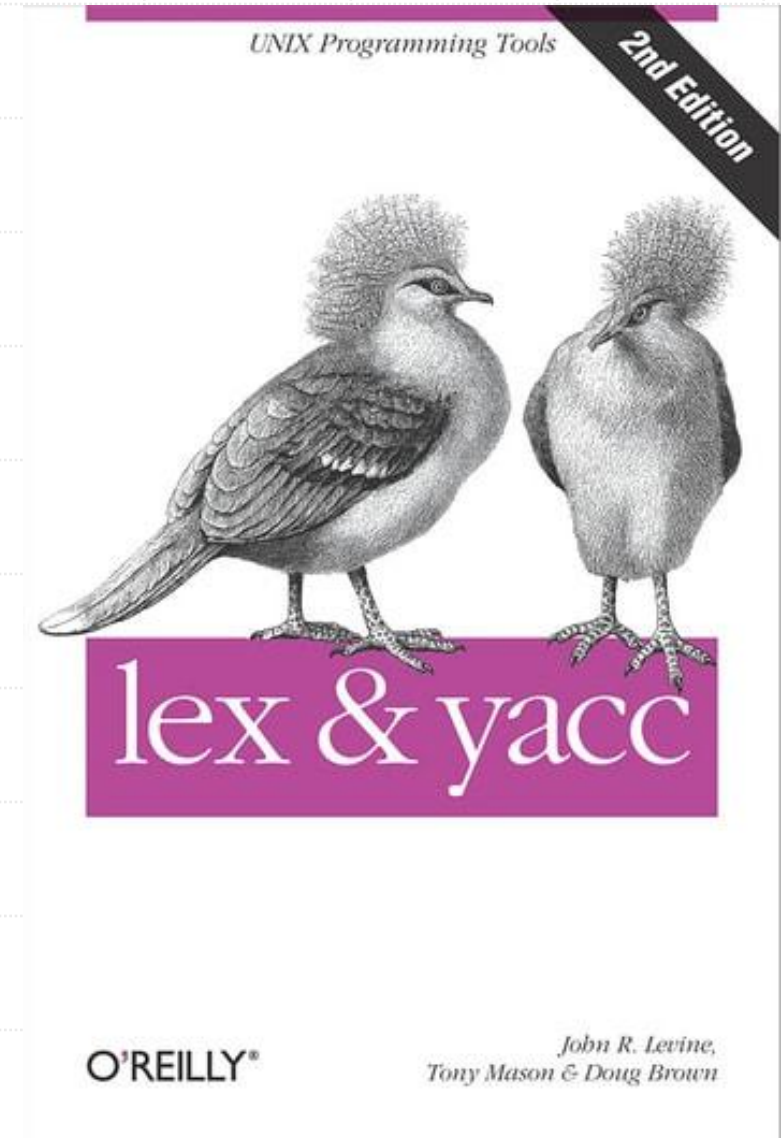
References


- **Compiler Construction - Principles and Practice**,
Kenneth C. Louden



References

- **Lex & Yacc**, Doug Brown, John Levine, and Tony Mason





日期	進度說明 Progress Description
2/17	Introduction and Overview
2/24	A Simple Compiler
3/3	Theory and Practice of Scanning
3/10	Lex (HW #1) and quiz #1
3/17	Grammars and Parsing
3/24	Top-Down Parsing I
3/31	Top-Down Parsing II
4/7	春假
4/14	Midterm
4/21	Bottom-Up Parsing I
4/28	Bottom-Up Parsing II
5/5	Yacc (HW #2) and quiz #2
5/12	Syntax-Directed Translation
5/19	Intermediate Representations
5/26	Code Generation for a Virtual Machine (HW #3)
6/2	Runtime Support, Target Code Generation
6/9	Final
6/16	Project demo (A simple compiler)



Chapter 1

Introduction






The progression of programming languages:

- Machine language `c7 06 0000 0002`
- Assembly language `mov x 2`
- High-level language `x = 2`

*The first compiler was developed by the team at IBM led by John Backus between 1954 and 1957.



Why do we need to learn compilers?

(1) for new platforms

(2) for new languages

- language extensions & improvement
- specification languages
- 4th generation languages (Ex: Perl, Python, Ruby, SQL, and MatLab)

(3) foundation of parallelizing compilers & related tools

(4) theories learned are applicable to other fields
e.g., silicon compiler, prototyping tools, database languages, text formatter, FSM (Finite State Machine) translator, query interpreter, command interpreter, interface programs, etc.

(5) for improving capabilities of existing compiler/interpreter

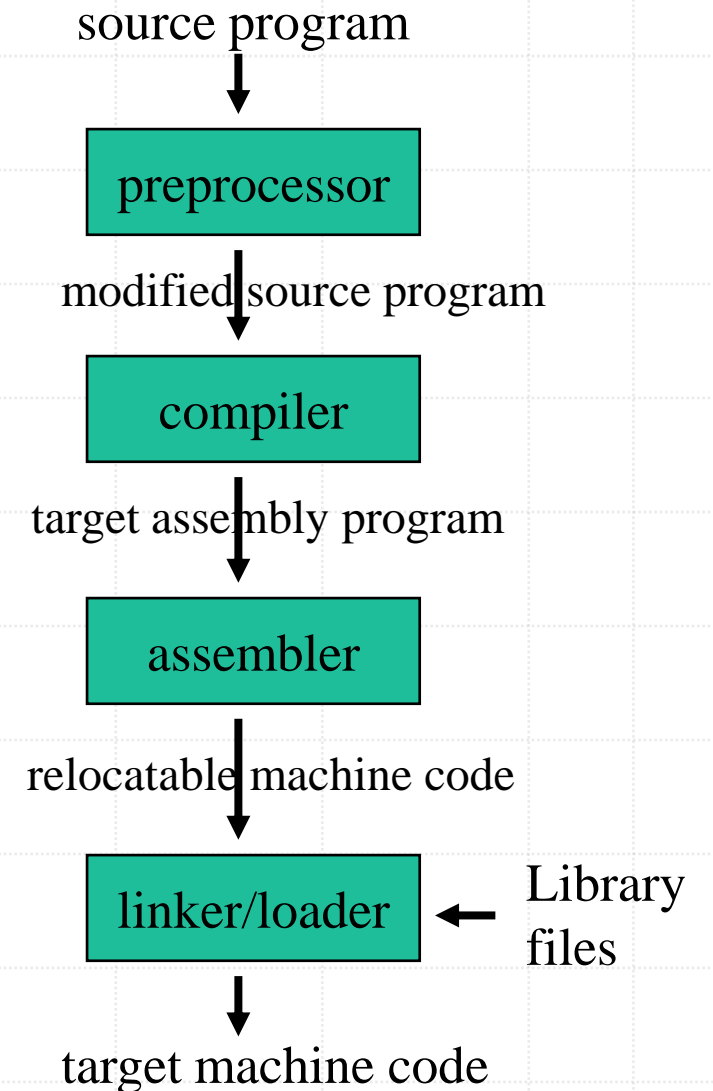


Silicon compiler

- Source language: conventional programming language
Variables represents not the location but logical signals (0 or 1) or groups of signals in a switching circuit.
- Output : circuit design in an appropriate language

Programs Related to Compilers

- Interpreters
- Assemblers
- Linkers
- Loaders
- Preprocessors
- Editors
- Debuggers
- Profilers
- Project Managers





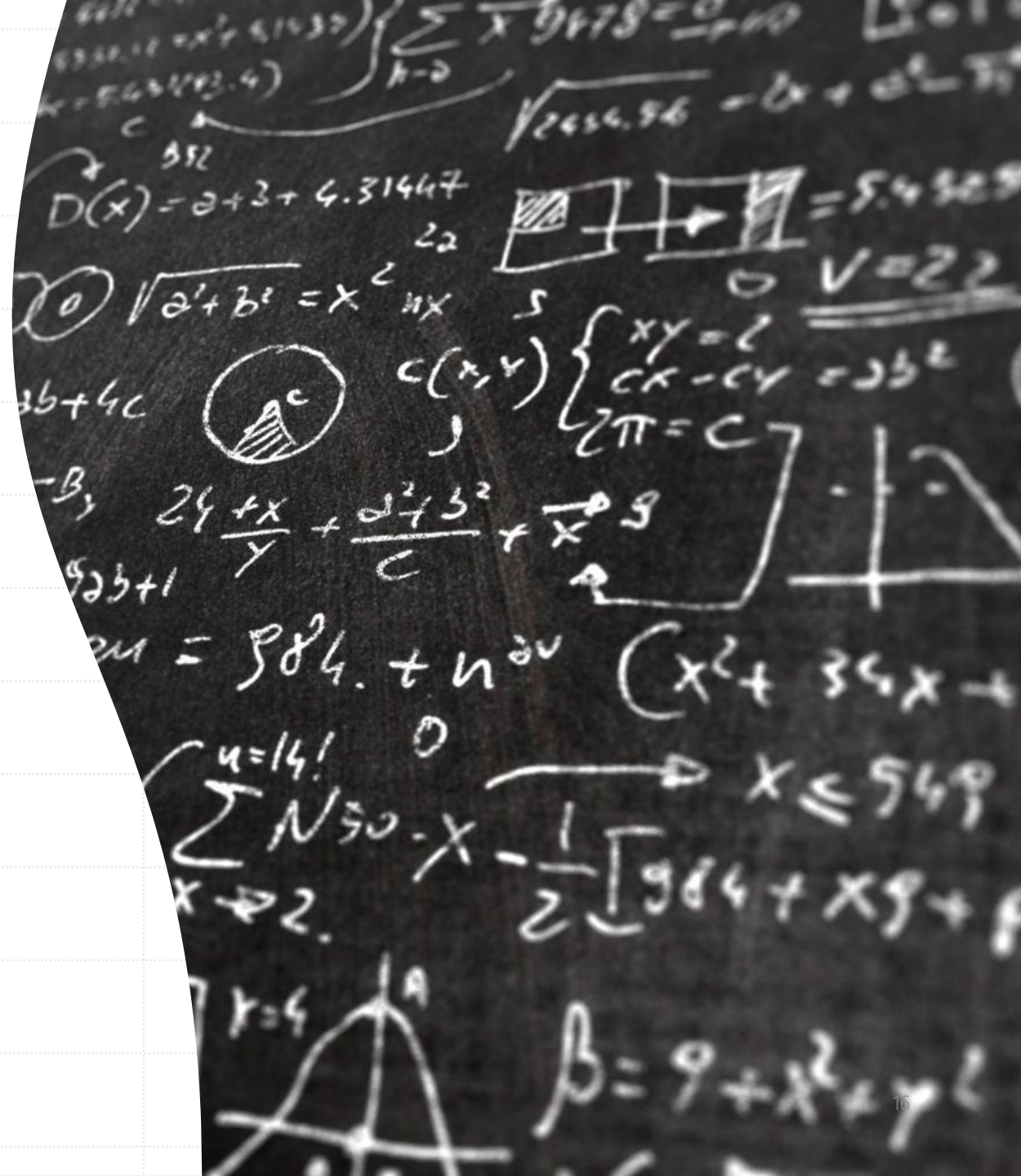
Definitions of Languages

- Source language
- Target language
- Implementation language

Translator

- A program, written in the implementation language, that takes sentences (strings) in the source language and outputs equivalent sentences (strings) in the target language.

e.g. - preprocessor, pretty printer,
fortran2c, pascal2c (high to high),
assembler (low to lower), disassembler
(lower to low), compiler (high to low)



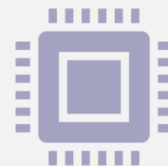
Category of compilers



1. Self-compiling Compiler
Source and implementation languages are the same.



2. Self-resident Compiler
Implementation and object languages are the same.

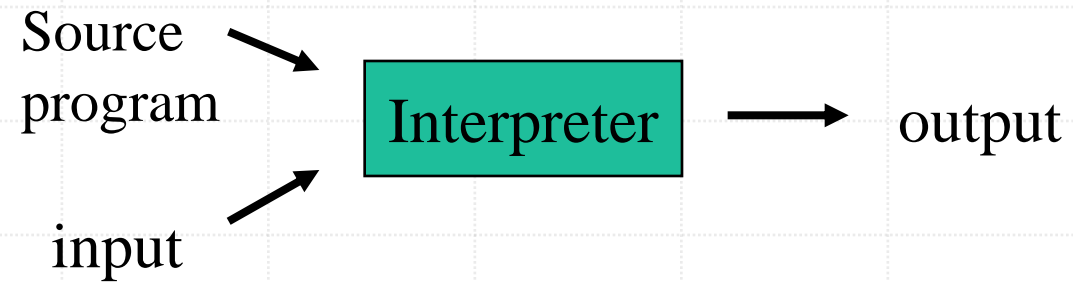


3. Cross compiler
A compiler that runs on one machine and produces object code for another machine.

Interpreter

- Def.

An interpreter performs the operations implied by the source program.



A hybrid compiler

Source program



Translator



Intermediate program



Input



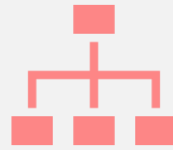
Virtual
Machine

→ Output

The Analysis-Synthesis Model of Compilation



There are two parts to compilation: analysis & synthesis.



During analysis, the operations implied by the source program are determined and recorded in a hierarchical structure called a tree.

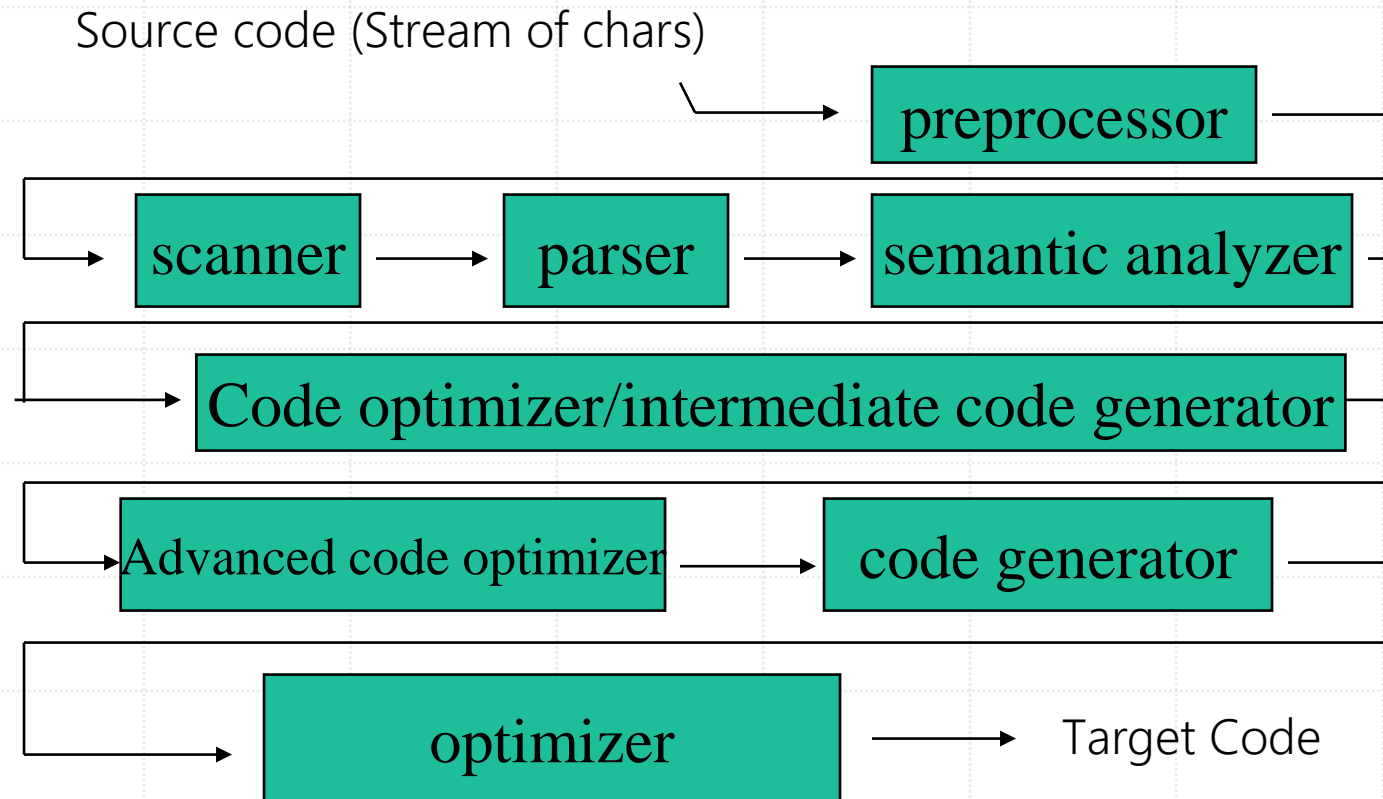


During synthesis, the operations involved in producing translated code.

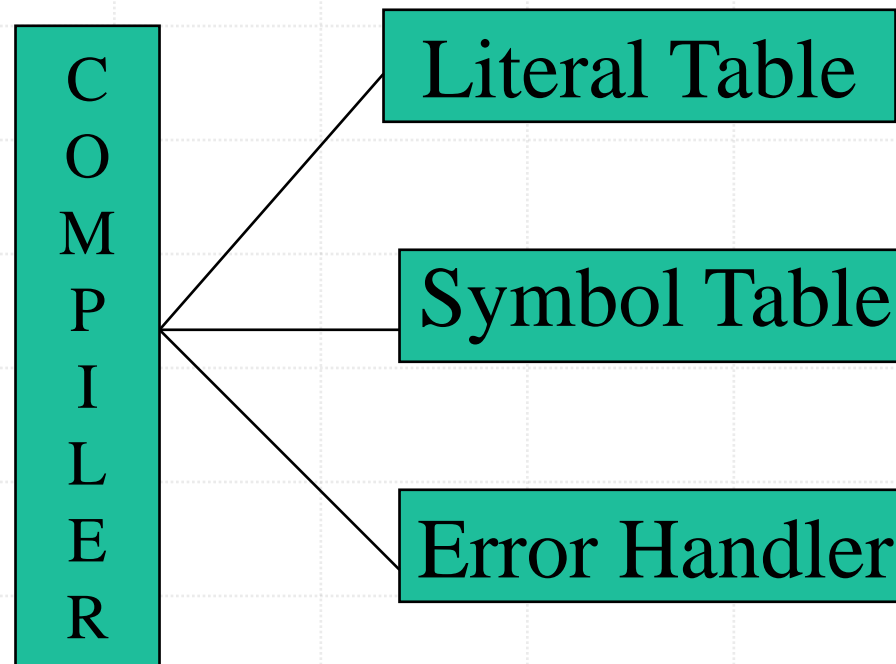
The Front-end and Back-end Model of Compilation



Compiling Process & Compiler Structure



Compiler Structure (continued)



Preprocessor (or Character handler)

- throw away the comments
- compress multiple blank characters
- include files (include nested files)
- perform macro expansions (nested macro expansion)
 - a macro facility is a text replacement capability (two aspects: definition & use).
 - a macro statement will be expanded into a set of programming language statements or other macro.
- compiler option (conditional compilation)
(These jobs may be conducted by lexical analyzer.)

```
mirror_mod = modifier_ob.  
# Add mirror object to mirror_mod  
mirror_mod.mirror_object =  
operation = "MIRROR_X":  
mirror_mod.use_x = True  
mirror_mod.use_y = False  
mirror_mod.use_z = False  
operation = "MIRROR_Y":  
mirror_mod.use_x = False  
mirror_mod.use_y = True  
mirror_mod.use_z = False  
operation = "MIRROR_Z":  
mirror_mod.use_x = False  
mirror_mod.use_y = False  
mirror_mod.use_z = True  
  
# Selection at the end -add  
mirror_ob.select= 1  
modifier_ob.select=1  
context.scene.objects.active  
("Selected" + str(modifier_ob.name))  
mirror_ob.select = 0  
= bpy.context.selected_object  
data.objects[one.name].select  
  
print("please select exactly  
-- OPERATOR CLASSES ----  
  
types.Operator):  
on X mirror to the selected  
object.mirror_mirror_x"  
mirror X"  
  
context):  
context.active_object is not
```



Scanner (Lexical Analyzer)

- To identify lexical (語彙) structure
- Input: a stream of chars;
- Output: a stream of tokens.
- A scanner may also enter identifiers into the symbol table and enter literals into literal table. (literals include numeric constants such as 3.1415926535 and quoted strings such as "Hello, world!").

An Example: $a[\text{index}] = 4 + 2 ;$

- (1) Output of the Scanner :

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

How tokens
(string of
chars) are
formed from
underlying
character set?



Usually specified (described) by
sequence of regular expression.



Lexical structures are analyzed via finite
state automata.



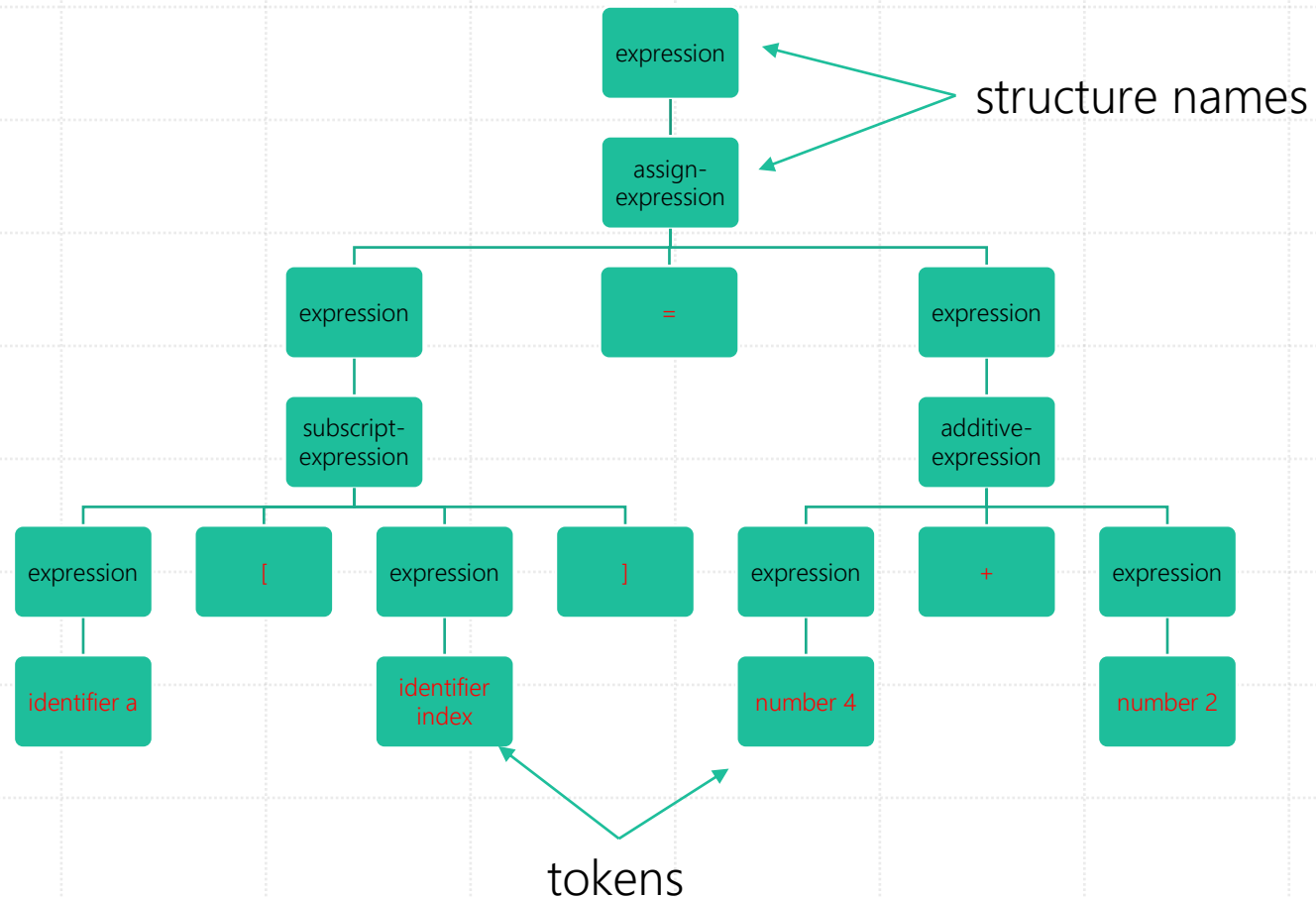
But it has the look-ahead requirement.
(To recognize a token the scanner may
need to look more characters ahead of
the token.)



Parser (Syntax Analyzer)

- To identify syntax structure
 - Input: a stream of tokens
 - Output: On a logical level, some representation of a parse tree.
 - Determine how do the tokens fit together to make up the various syntax entity of a program.
 - ** Most compilers do not generate a parse tree explicitly but rather go to intermediate code directly as syntax analysis takes place.
- Usually specified via context free grammar.

(2) Output of the parser – parse tree (logical level)

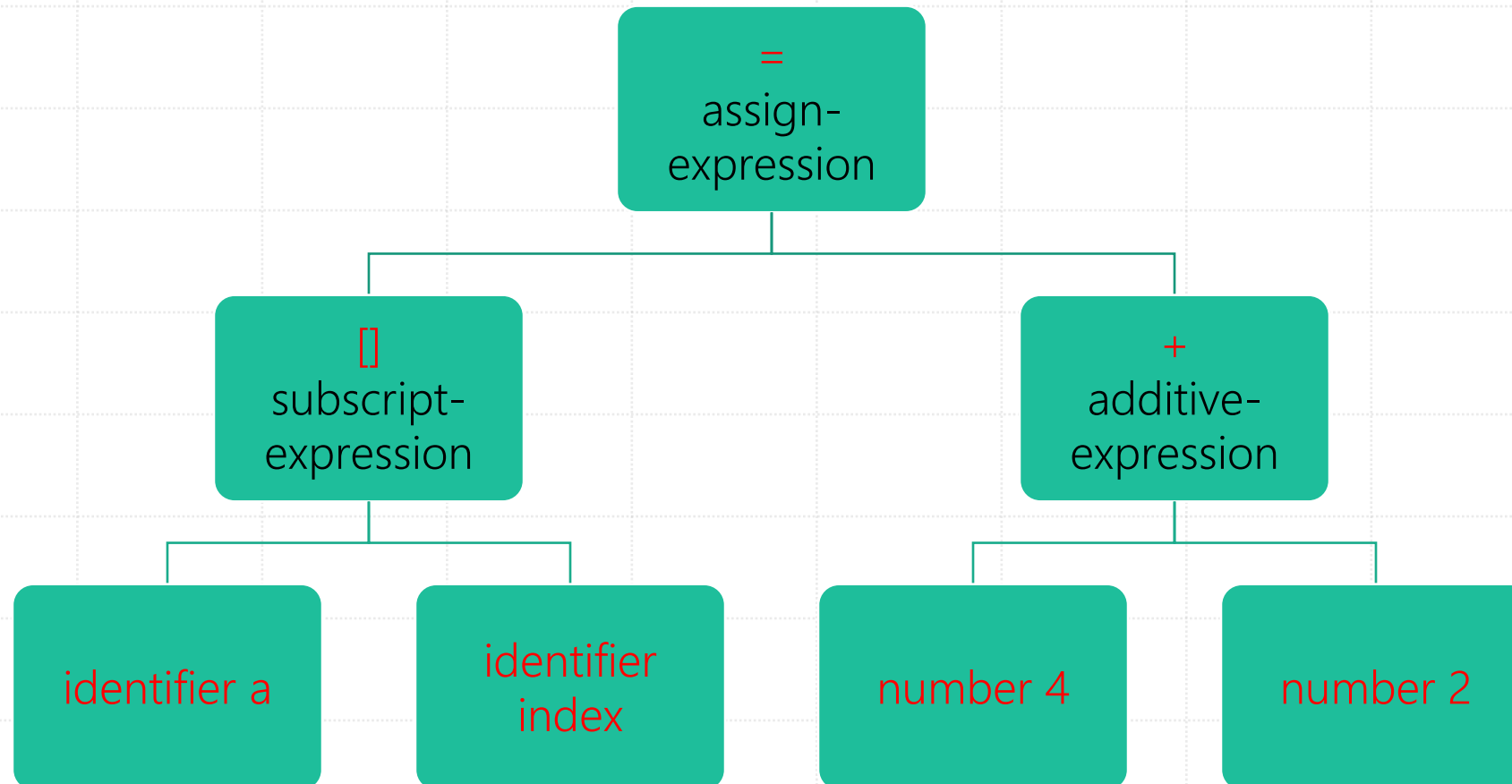




Predefined context-free grammar

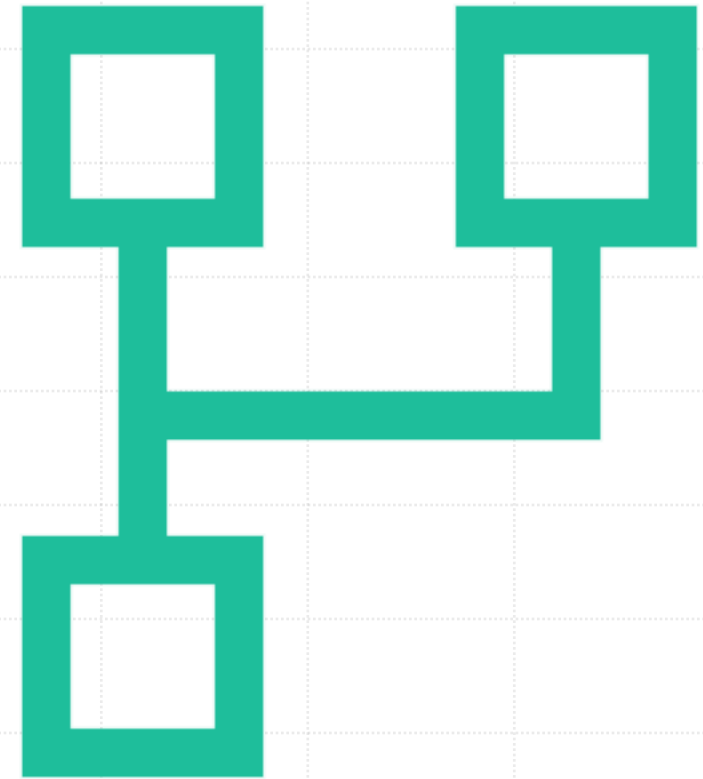
- $\text{expression} \rightarrow \text{assign-expression} \mid \text{subscript-expression} \mid \text{additive-expression} \mid \text{identifier} \mid \text{number}$
- $\text{assign-expression} \rightarrow \text{expression} = \text{expression}$
- $\text{subscript-expression} \rightarrow \text{expression} [\text{expression}]$
- $\text{additive-expression} \rightarrow \text{expression} + \text{expression}$

(2)' Output of the parser – Abstract Syntax Tree (condensed parse tree)

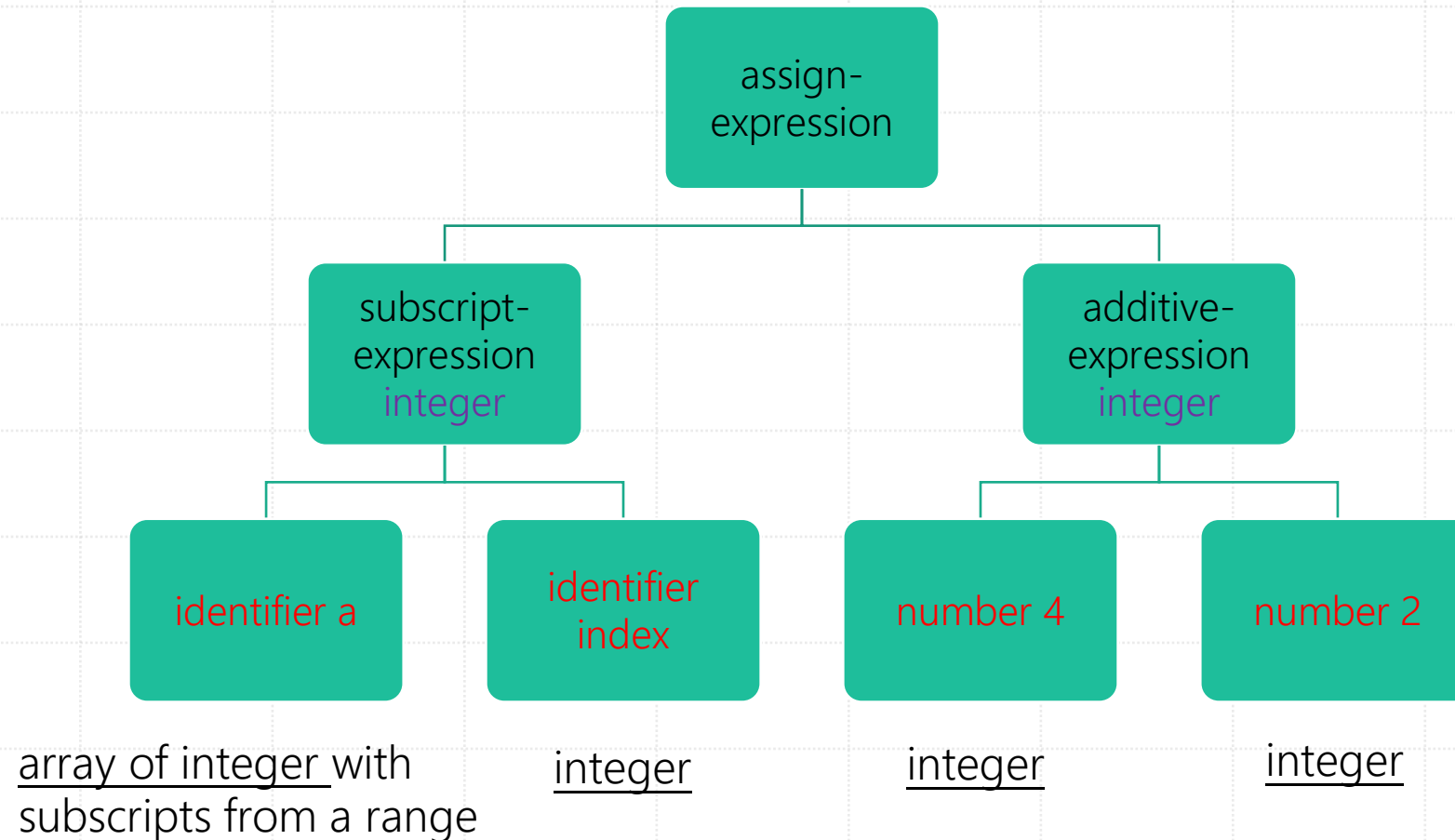


Semantic Analyzer

- Semantic Structure
 - What is the program supposed to do?
 - Semantics analysis can be done during syntax analysis phase or intermediate code generator phase or the final code generator.
 - typical static semantic features include declarations and type checking.
 - information (attributes) gathered can be either added to the tree as annotations or entered into the symbol table.



(3) Output of the semantic analyzer – annotated AST



(3) Output of the semantic analyzer (cont'd)



finds the consistence of data type among 'a', 'index', and $2 + 4$, or



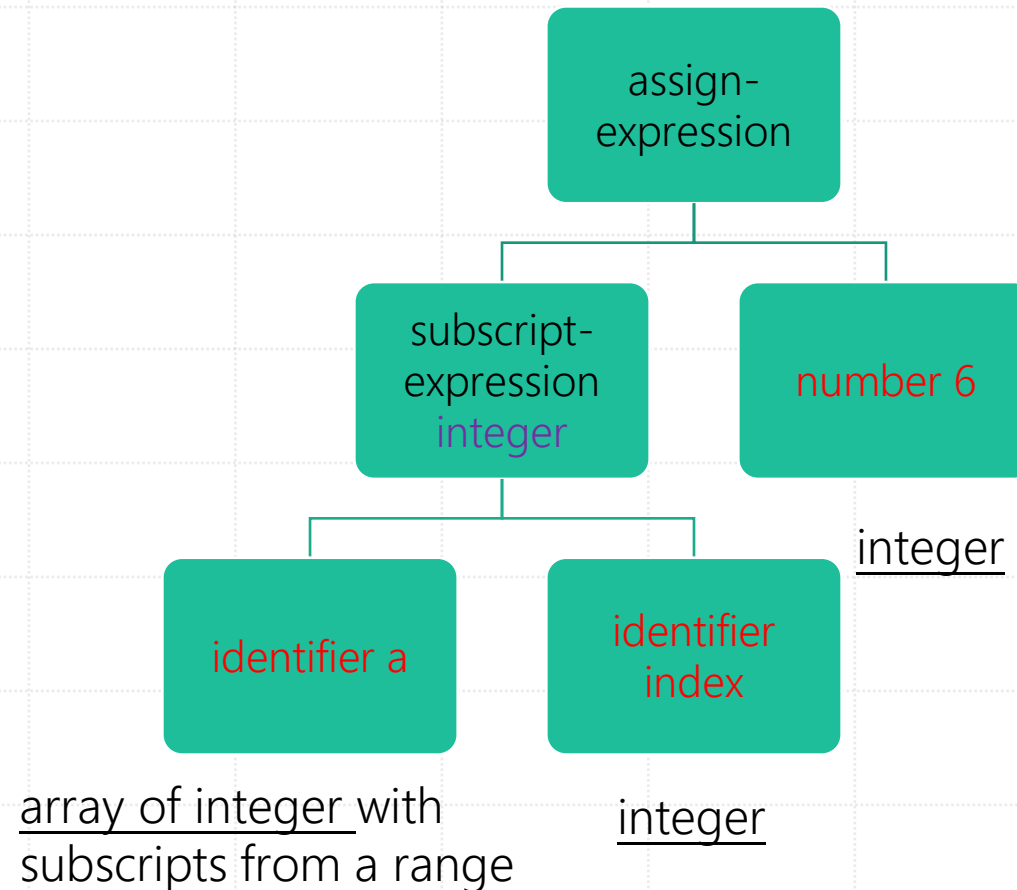
declares a type mismatch error if not.

The time ratio for
scanning, parsing,
and semantic
processing is
30:25:45.



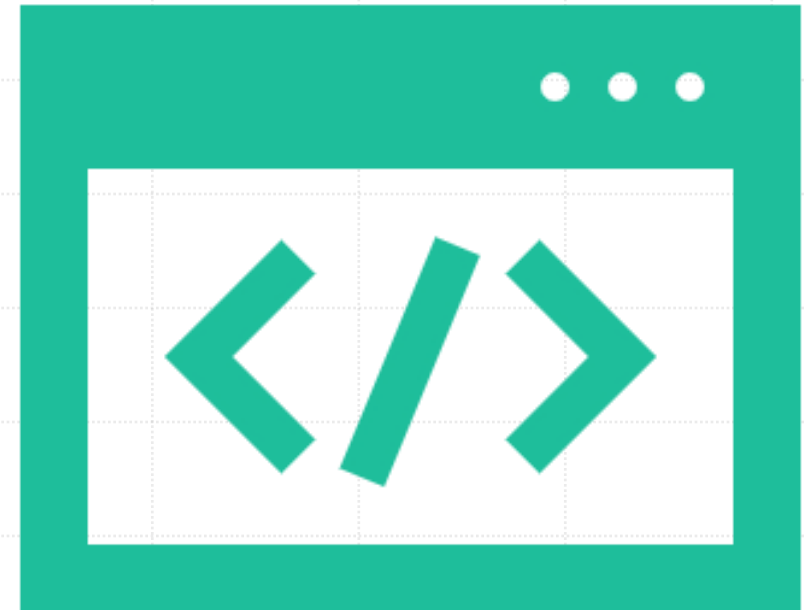
Source Code Optimizer

(4)' Output of the Source Code Optimizer



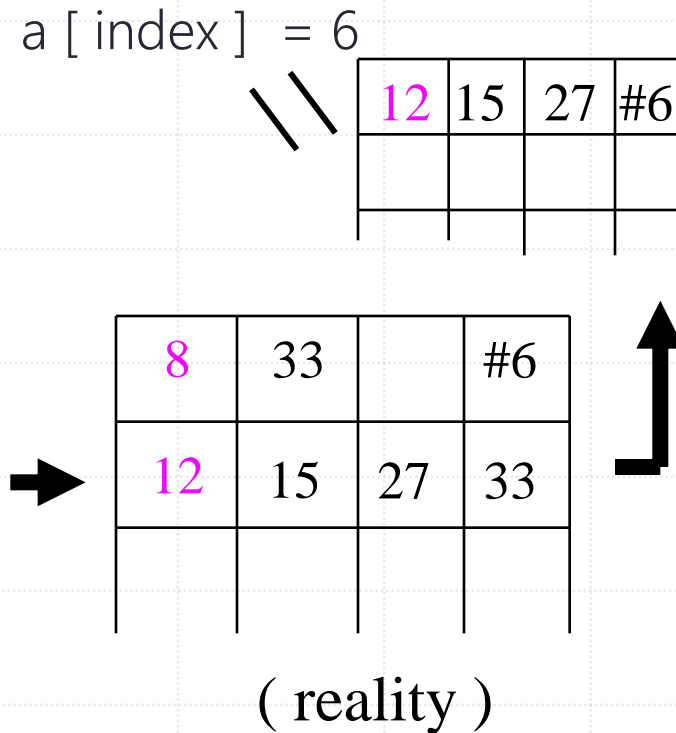
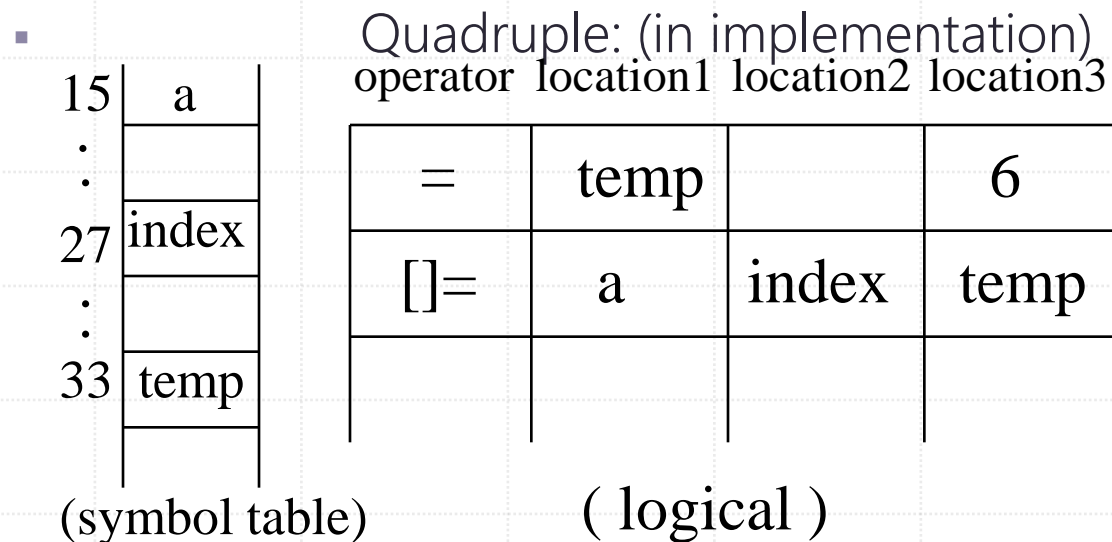
Intermediate Code Generator


- Transform the parse tree (logical level) into an intermediate language representation, e.g., three address code: $A = B \text{ op } C$ (op is a binary operator)
- Difference between intermediate code and assembly code
 - Specify the registers to be used for each operation in assembly code
 - Actually intermediate code can be represented as any internal representation such as the syntax tree.



(4) Output of the intermediate code generator

- intermediate code (three address code, two address code, P-code, etc.)
- Three address code
- temp = 6
- a [index] = temp





Advanced Code Optimizer

Detection of undefined variables

Detection of loop invariant
computation

Constant folding

Removal of induction variables

Elimination of common expression

Induction Variable Elimination

- When there are two or more induction variables in a loop we have opportunity to get rid of all but one.

.....

I = 1

Repeat

T = 4 * I ==>

X = Y [T]

Prod = Prod + X

I = I + 1

Until I > 20

.....

T = 0

Repeat

T = T + 4

X = Y [T]

Prod = Prod + X

Until T > 76

* Suppose I is not needed after the loop terminates



Elimination of common expression

$$A = B + C + D$$

$$E = B + C + F$$

might be

$$T = B + C$$

$$A = T + D$$

$$E = T + F$$



Code Generator



(5) Output of the code generator

```
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
```




(Machine-dependent) Peephole Optimizer

- A simple but effective technique for locally improving the target code.
- Examine a short sequence of target instruction (called peephole) and replacing these instruction by a shorter or faster sequence whenever possible.

e.g. redundant instruction elimination
flow-of-control optimization
algebraic simplification
use of machine idioms



(6) Output of the peephole optimizer

```
Mov  R0, index    // value of index -> R0
Shl   R0           // double value in R0
Mov  &a[R0], 6     // constant 6 -> address a + R0
```

Error Handling (Detection & Reporting)

- An important function of the compiler.
- Errors can be encountered by all of the phases of a compiler.
- The error messages should be reported to allow the programmer to determine where the errors have occurred.
- Once the error has been noted the compiler must modify the input to allow the latter phases can continue processing.

```
mirror_mod = modifier_ob.  
# Add mirror object to mirror  
mirror_mod.mirror_object =  
operation = "MIRROR_X":  
mirror_mod.use_x = True  
mirror_mod.use_y = False  
mirror_mod.use_z = False  
operation = "MIRROR_Y":  
mirror_mod.use_x = False  
mirror_mod.use_y = True  
mirror_mod.use_z = False  
operation = "MIRROR_Z":  
mirror_mod.use_x = False  
mirror_mod.use_y = False  
mirror_mod.use_z = True  
  
# Selection at the end - add  
mirror_ob.select = 1  
modifier_ob.select = 1  
context.scene.objects.active  
("Selected" + str(modifier_ob.name))  
mirror_ob.select = 0  
= bpy.context.selected_object  
data.objects[one.name].select  
  
print("please select exactly  
-- OPERATOR CLASSES --  
  
types.Operator):  
on X mirror to the selected  
object.mirror_mirror_x"  
error X"  
  
context):  
context.active_object is not
```

Phase	Example
Lexical Analyzer	A token is misspelled.
Syntax Analyzer	A syntax entity is unable to be inferred.
Semantic analyzer/Intermediate Code Generator	An operator whose operands have incompatible types.
Code Optimizer	Certain statements can never be reached.
Code Generator	A compiler-created constant is too large to fit in a word of the target machine
Symbol Table Management	An identifier that has been multiply declared with contradictory attribute.



Major Data Structures in a Compiler

Token	=> a value
-------	------------

The Syntax Tree	=> pointer-based structure
-----------------	----------------------------

The Symbol Table	=> hash table/an array of struct/...
------------------	--------------------------------------

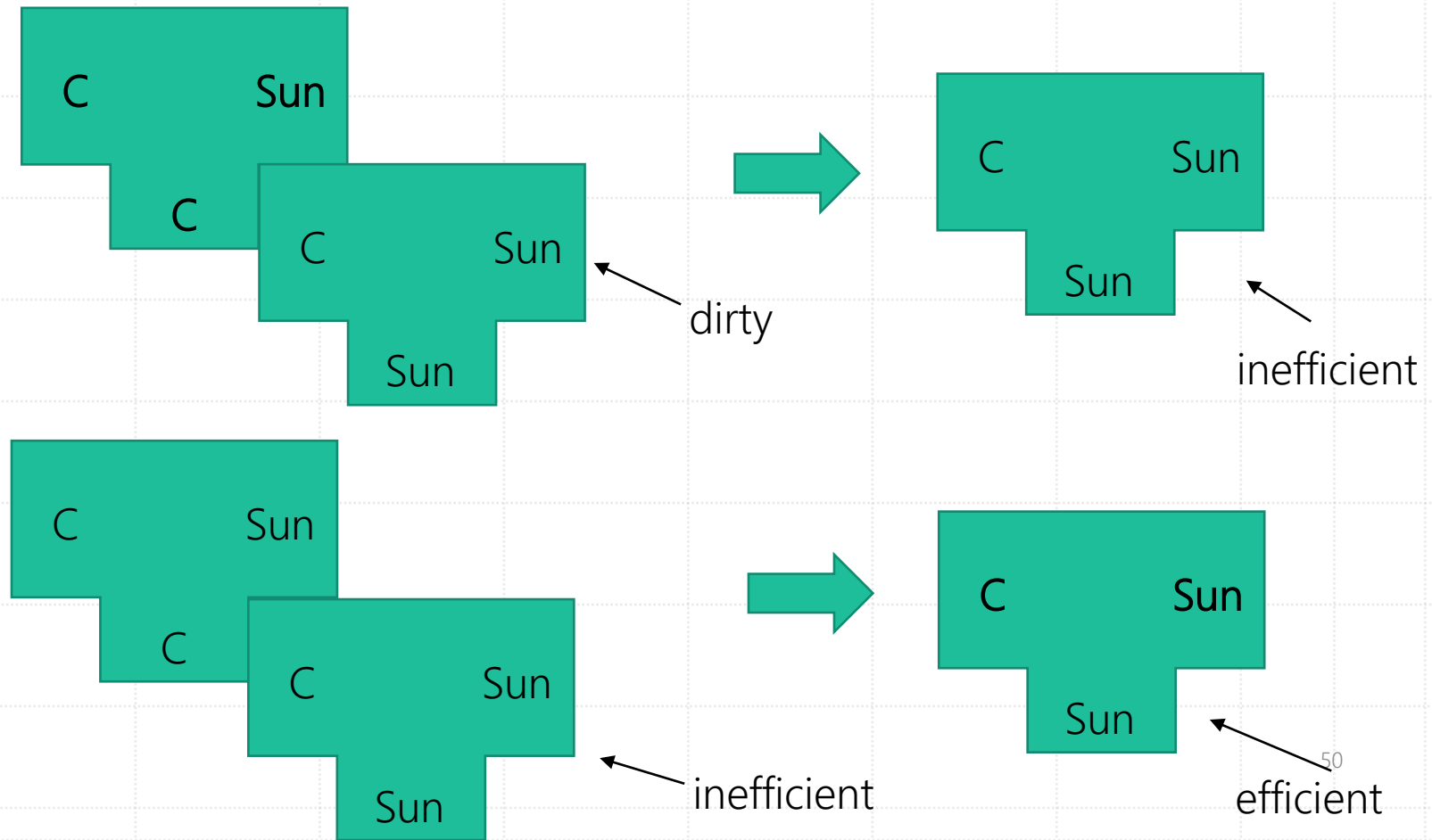
The Literal Table	=> an array of struct
-------------------	-----------------------

Intermediate Code	=> Quadruple (an array of struct)
-------------------	-----------------------------------

Temporary Files	
-----------------	--

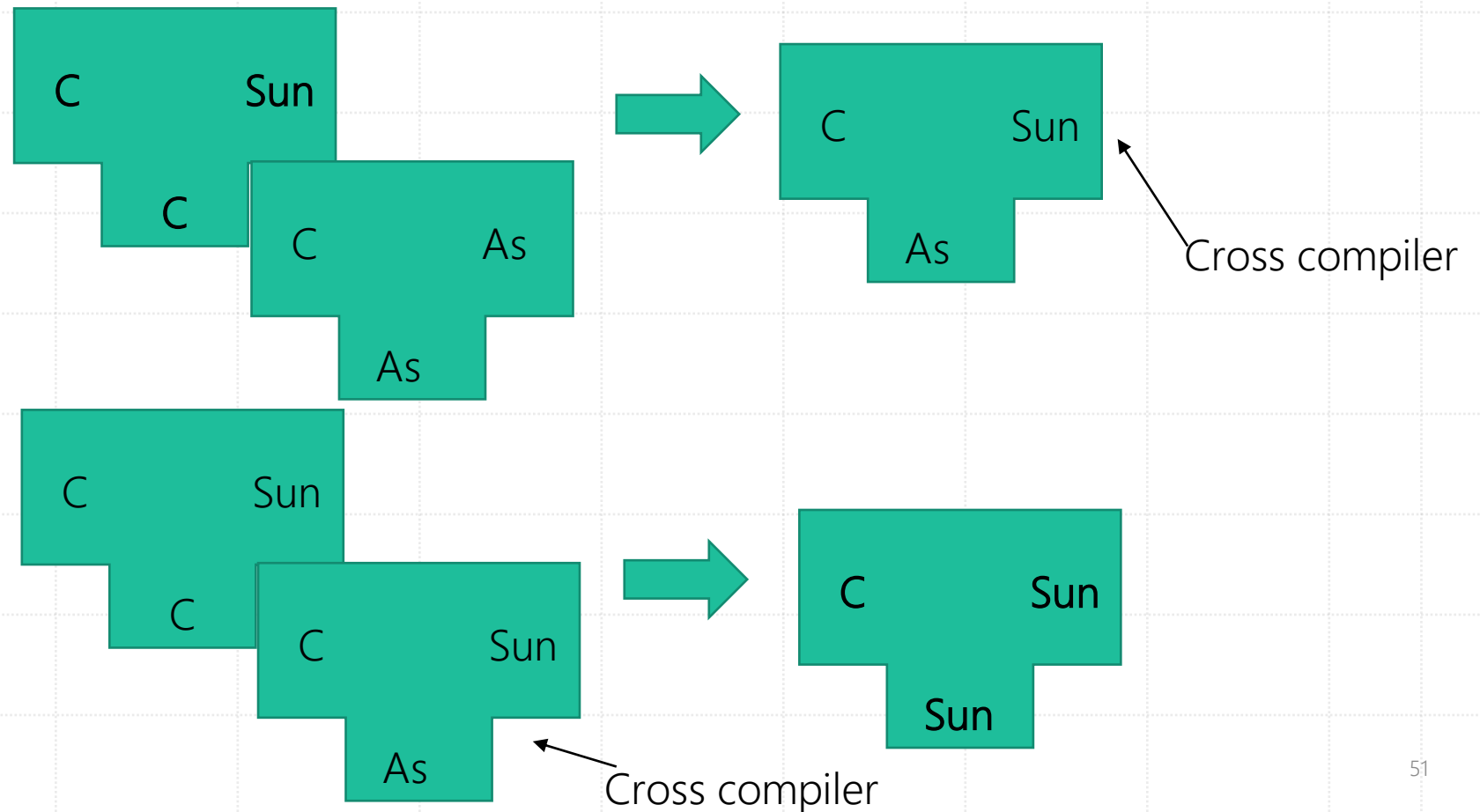
Developing the first compiler

- Suppose that we have a self-compiling C compiler for Sun Sparc 2. Suppose we also have an inefficient self-resident C compiler for Sun Sparc 2. How can we get an efficient self-resident C compiler for Sun Sparc 2?



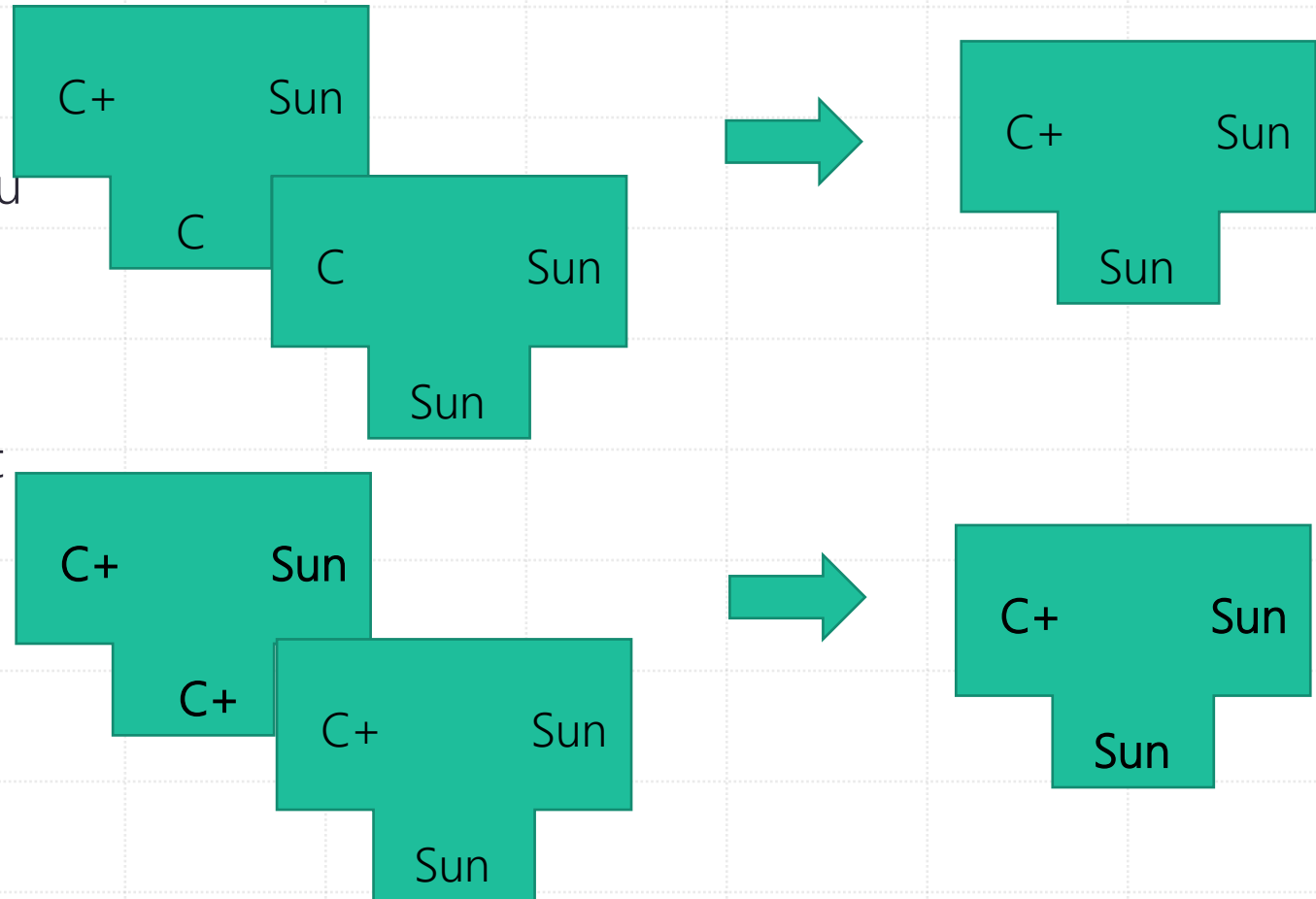
Porting a compiler for a new machine

- Suppose that you have a self-compiling C compiler for Sun Sparc 2. Suppose you also have a self-resident C compiler for IBM AS400. How can we get a self-resident C compiler for Sun Sparc 2?



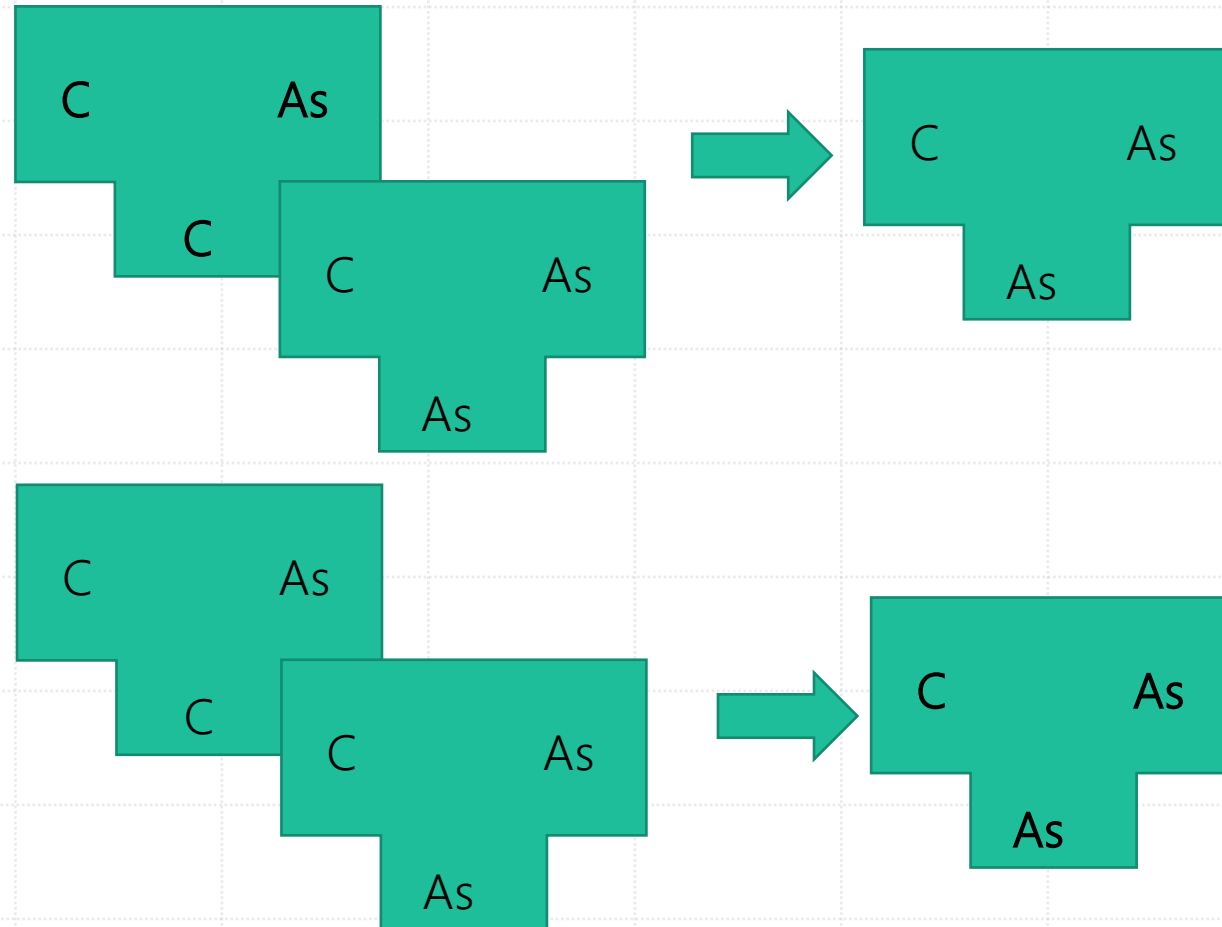
Extending a language and developing its corresponding compiler

- Suppose you have both self-compiling and self-resident C compilers for Sun Sparc 2. If you want to extend the C language to become C+ with some new features. How do you get the self-compiling and self-resident C+ compilers for Sun Sparc 2?



Improving an existing compiler

- Suppose you have a good self-resident C compiler for IBM AS400. Now you want to develop an enhanced version of C compiler with excellent optimizing capabilities for IBM AS400. How do you do it?



```
main() {  
    int a = 1;  
    int b = 1;  
    {  
        int b = 2;  
        {  
            int a = 3;  
            cout << a << b;  
        }  
        {  
            int b = 4;  
            cout << a << b;  
        }  
        cout << a << b;  
    }  
    cout << a << b;  
}
```

B₁

B₂

B₃

B₄

Blocks in a C++ program