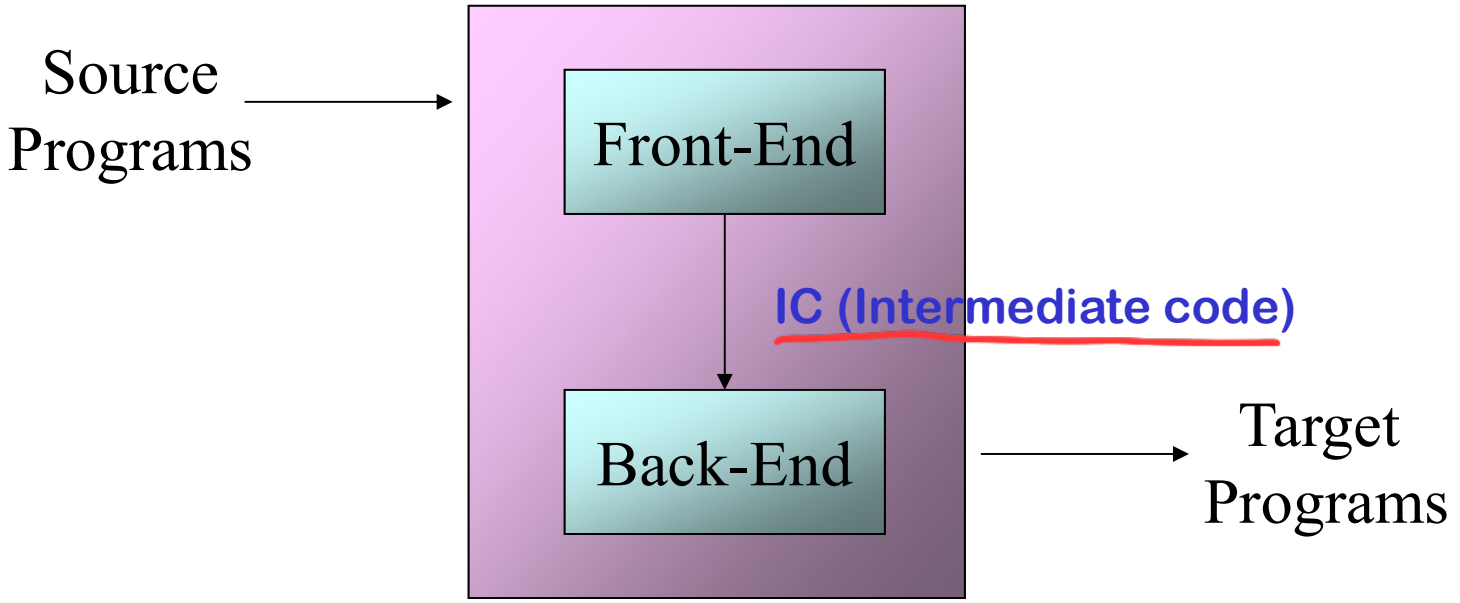
Chapter I
Introduction : part II

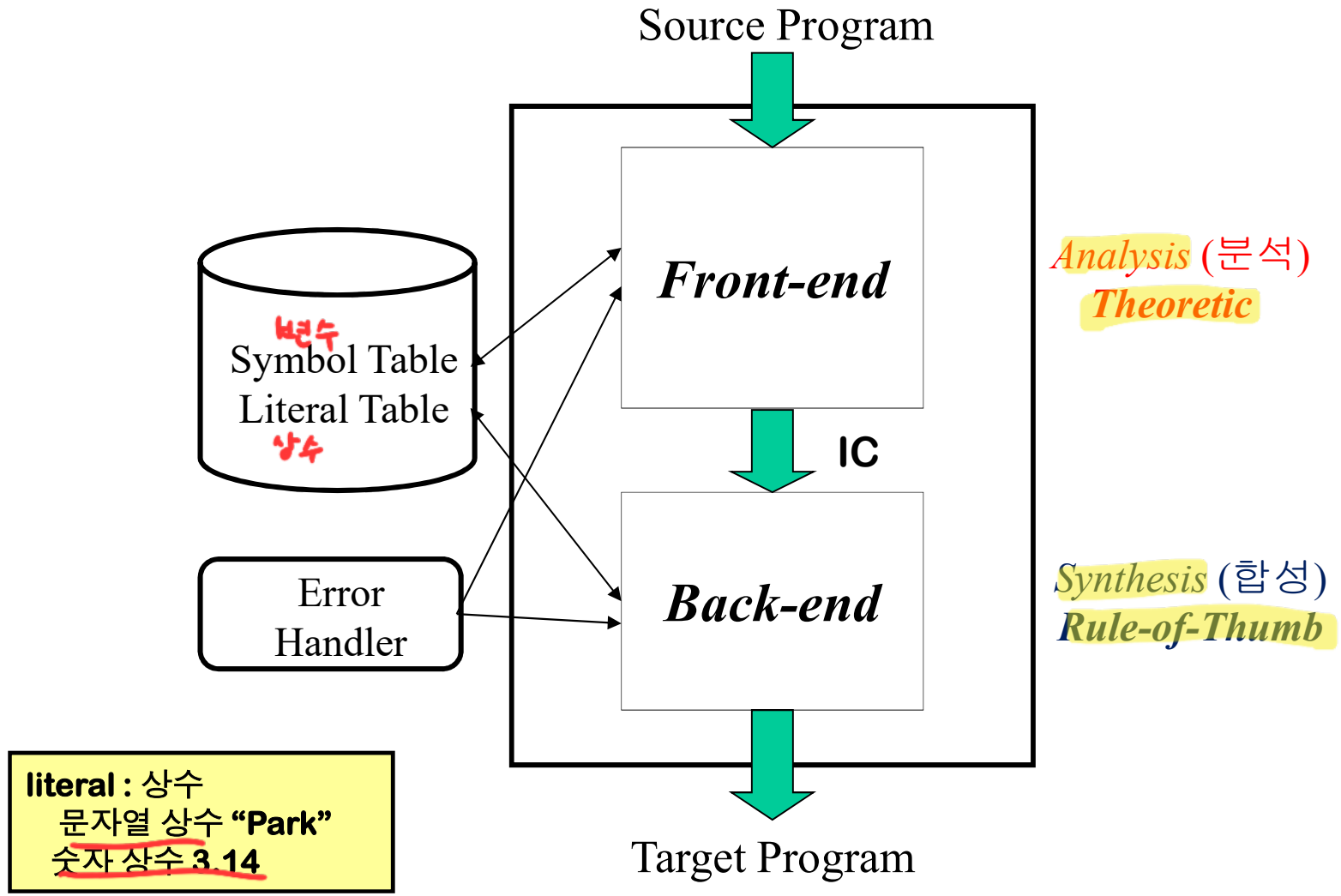
컴퓨터공학과
교수 홍 윤 식
yshong @ *inu.ac.kr*

컴파일러 구조(*an abstract view*)



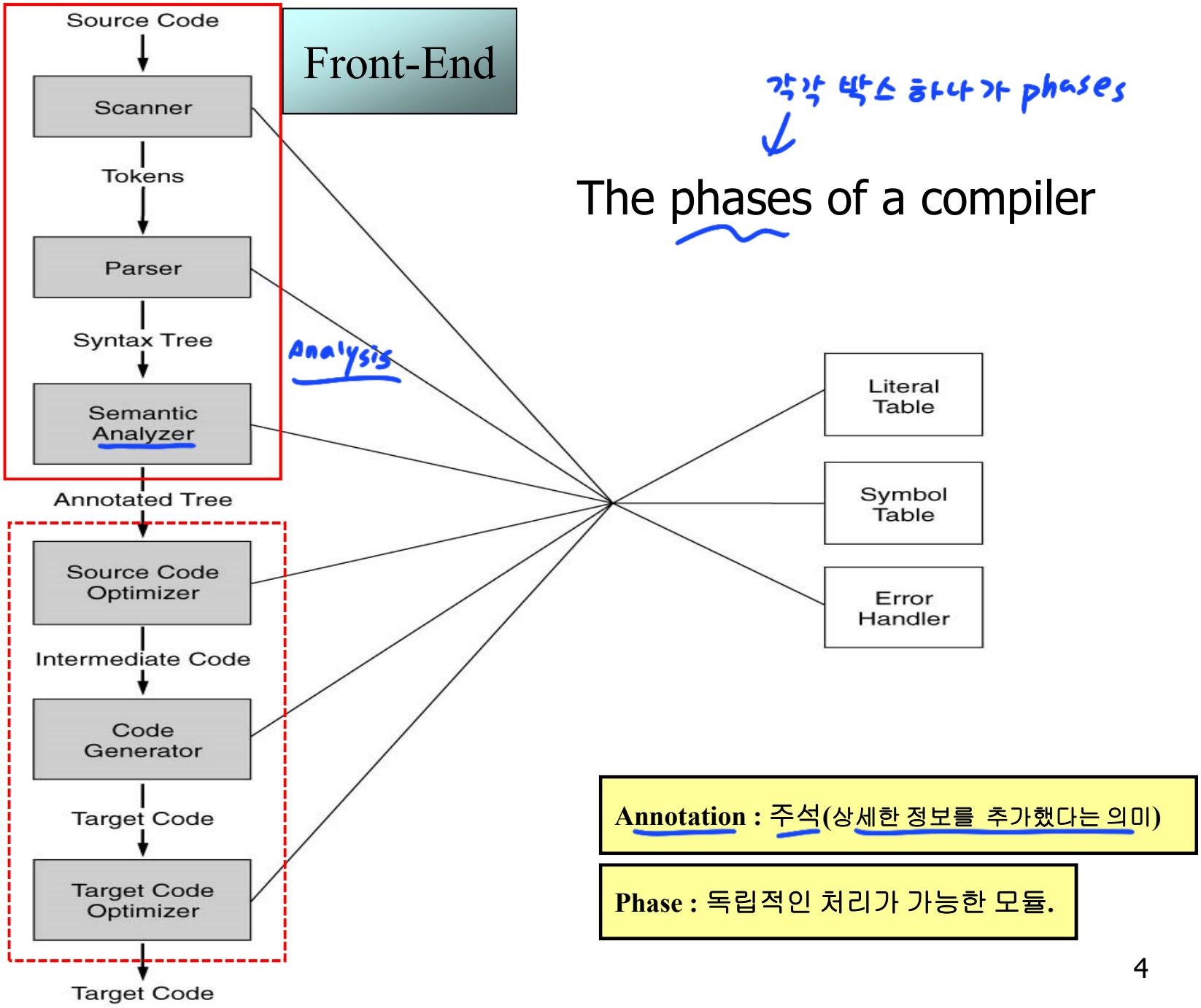
Front-end(전반부) : **language-dependent**
Back-end(후반부) : **machine-dependent**

컴파일러 구조(*a detailed view*)



The phases of a compiler

각각 박스 하나가 phases



Annotation : 주석(상세한 정보를 추가했다는 의미)

Phase : 독립적인 처리가 가능한 모듈.

어휘 분석 (*Scanner*, *Lexical Analyzer*)

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

Token
= a sequence of characters

`a`	identifier (변수)
`[`	left bracket
`index`	identifier
`]`	right bracket
`=`	assignment (연산 기호)
4	number (literal, 숫자 상수)
`+`	plus sign (연산 기호)
`2`	number

구문 분석(**Parser, Syntax Analyzer**)

assignment statement

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

- 어떤 종류의 문장(statement)인가?

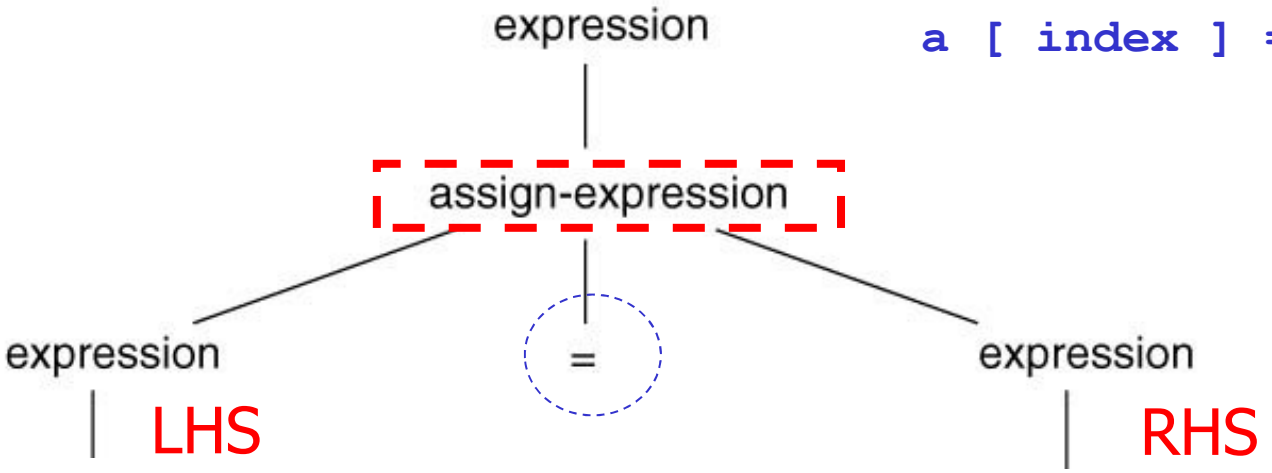
  … = ….

  – **LHS** variable = **RHS** expression
    - 할당 문장 (assignment statement) 이고,
    - 오른쪽 수식(expression)은 덧셈 연산이며,
    - 왼쪽 변수는 배열의 원소를 가리킴
  – 이런 생각을 구체적으로 나타내면?

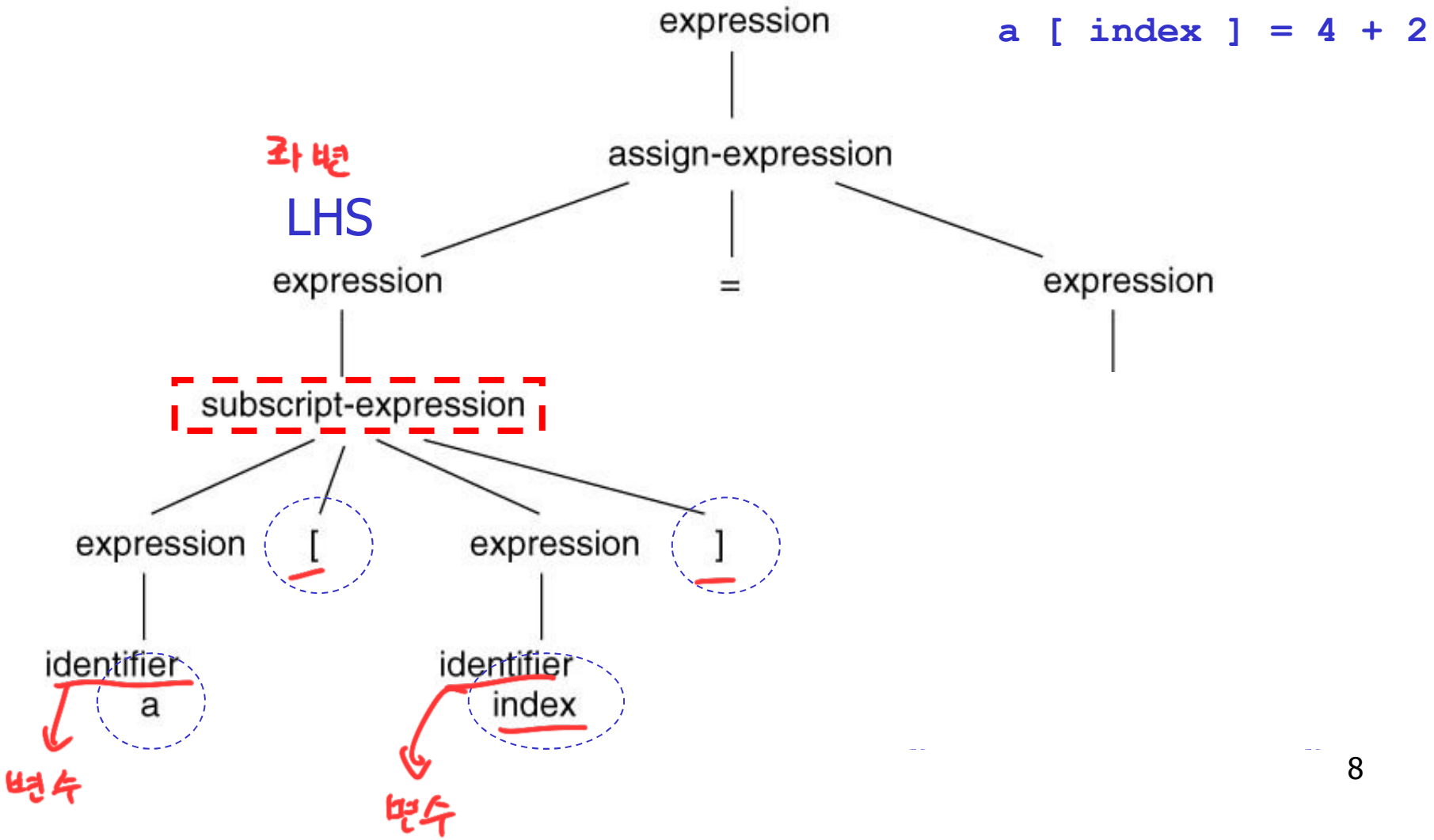
LHS : Left Hand Side
RHS : Right Hand Side

Statement 와 Expression은 어떻게 다른가?

구문 분석(Parser, Syntax Analyzer) (1/3)

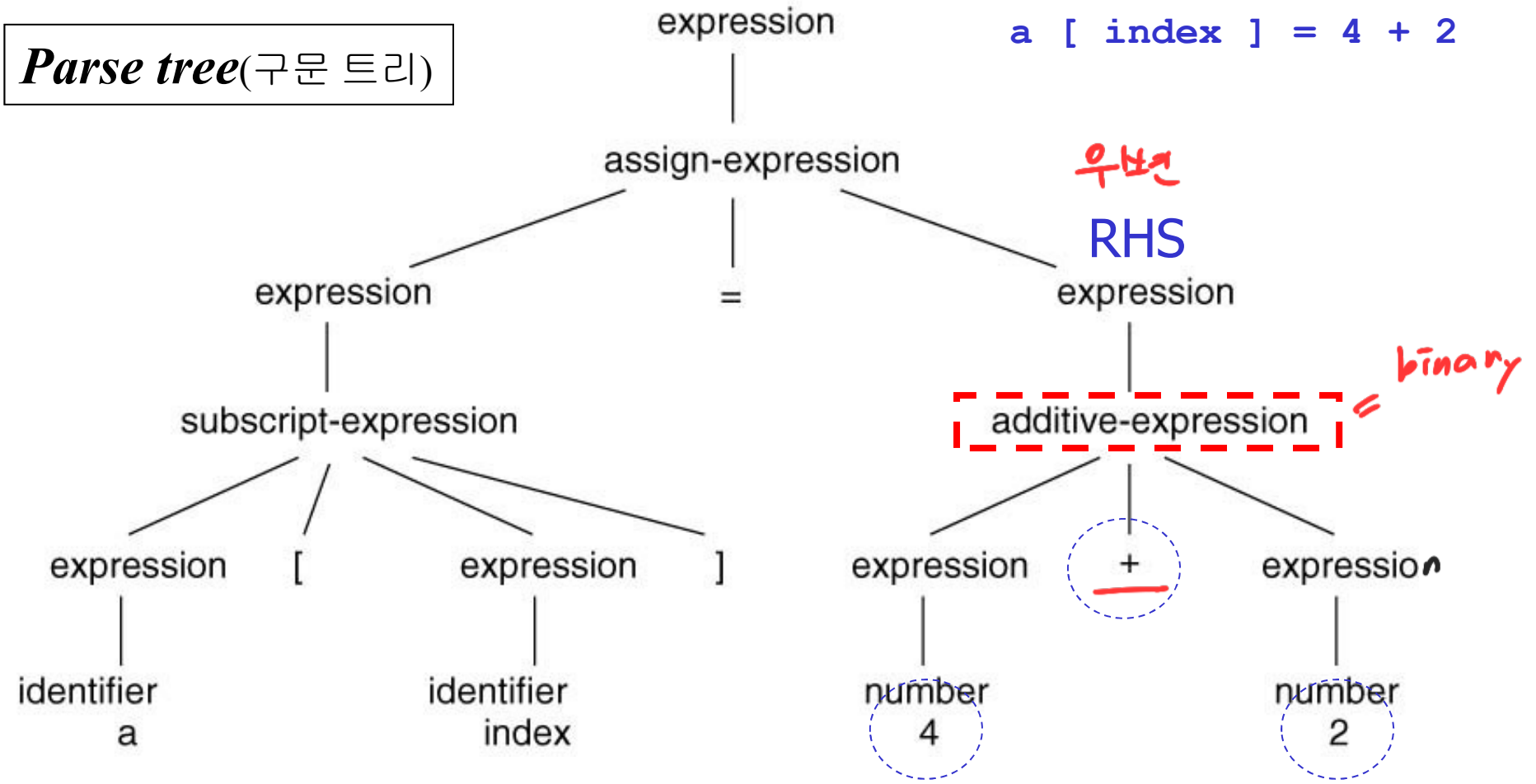


구문 분석(**Parser**, **Syntax Analyzer**) (2/3)



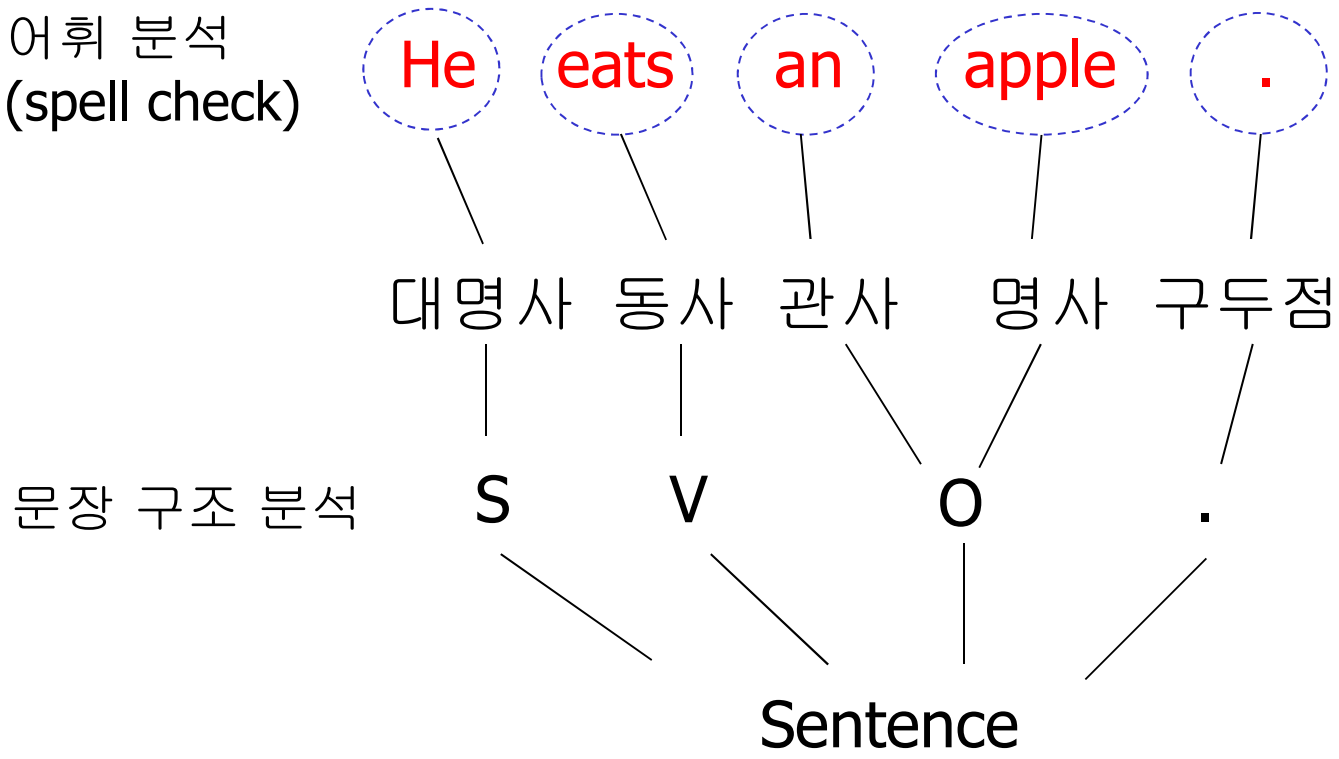
구문 분석(**Parser**, **Syntax Analyzer**) (3/3)

Parse tree(구문 트리)



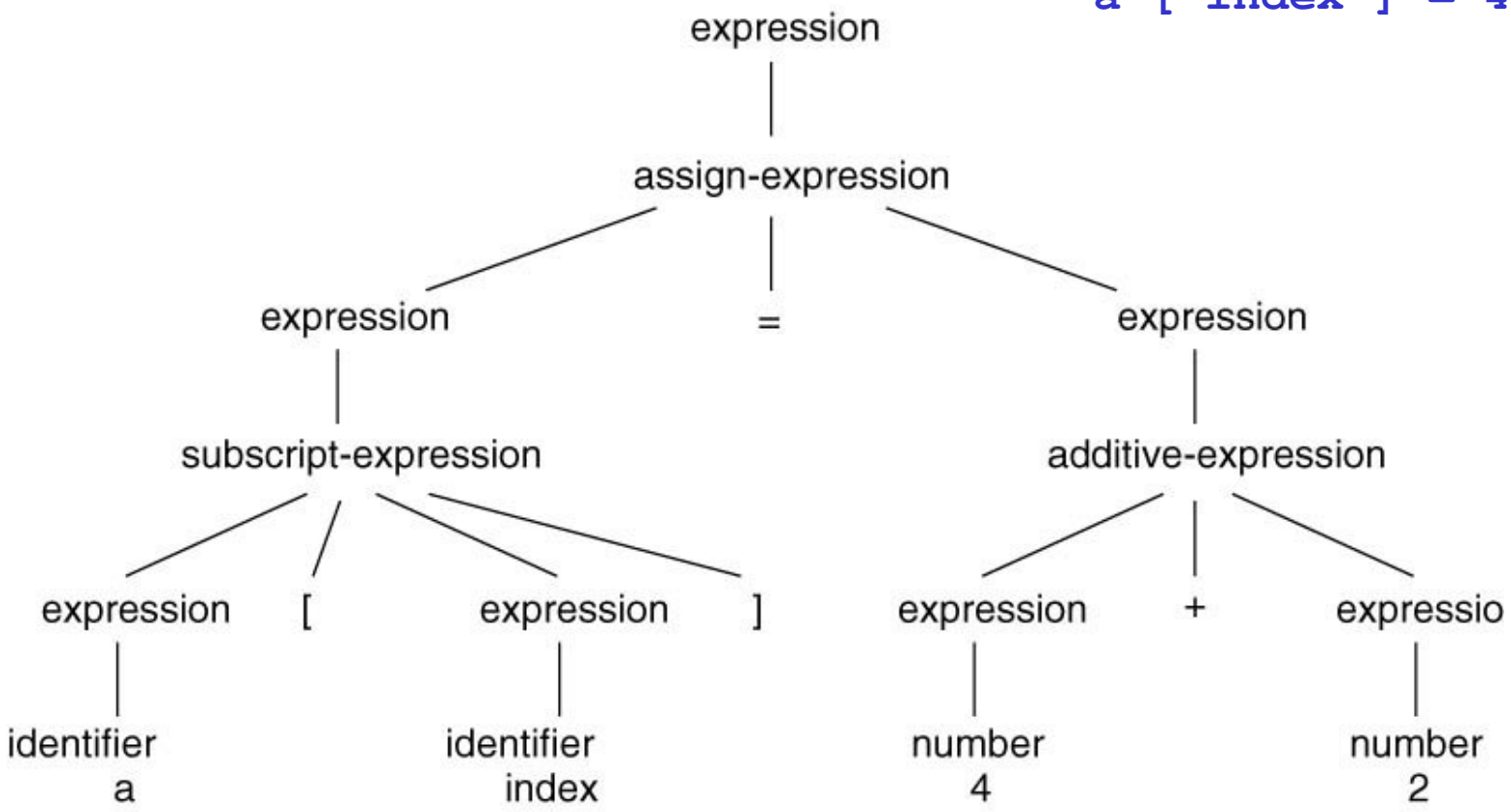
An illustrative example

He eats an apple.



구문 분석 : Concrete Syntax Tree

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

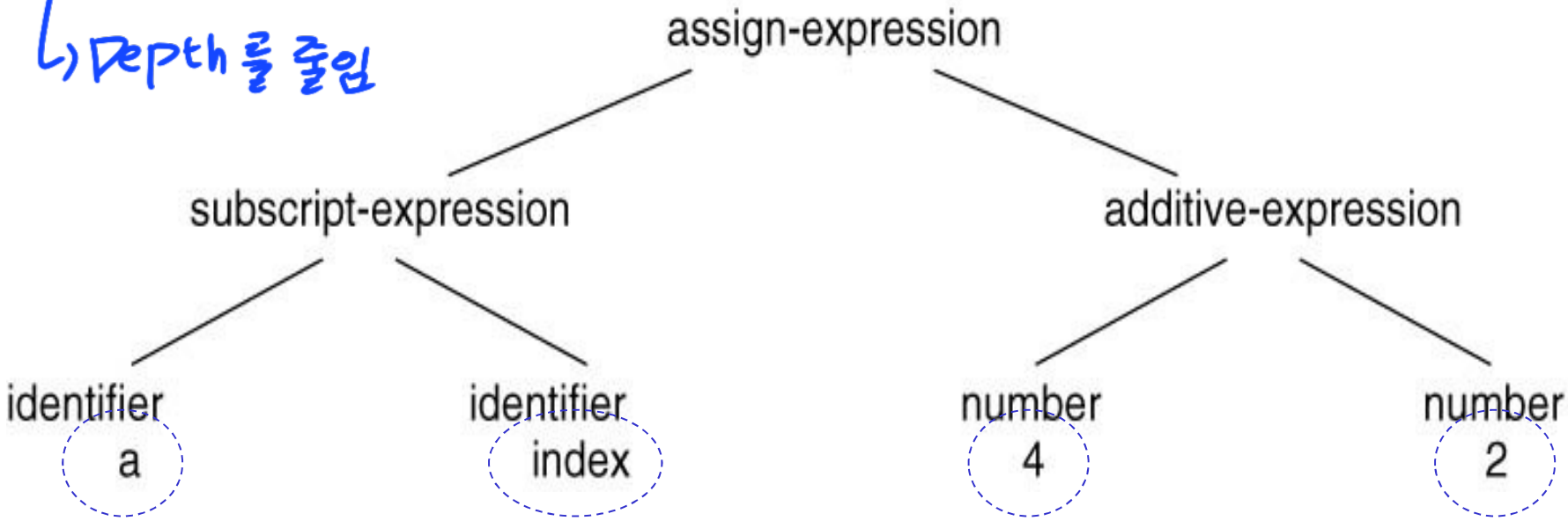


구문 분석 : Abstract Syntax Tree(AST)

AST : 추상 구문 트리

↳ Depth를 줄임

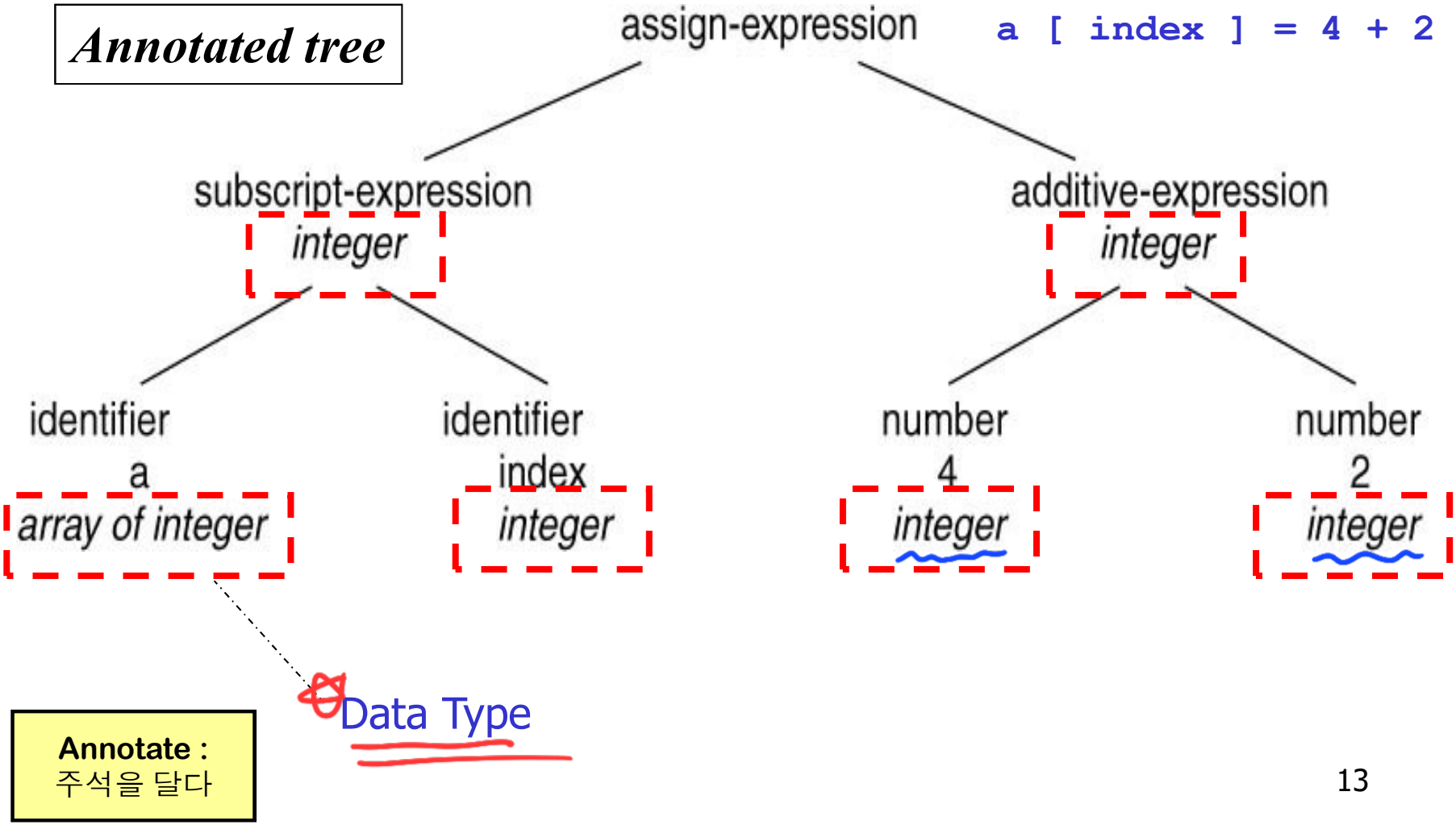
```
a [ index ] = 4 + 2
```



왜 **abstract**라고 부를까?
(코드 생성에)필요한 부분만 남기고 나머지 문장 구성 요소를 없앴기 때문

Type 정의

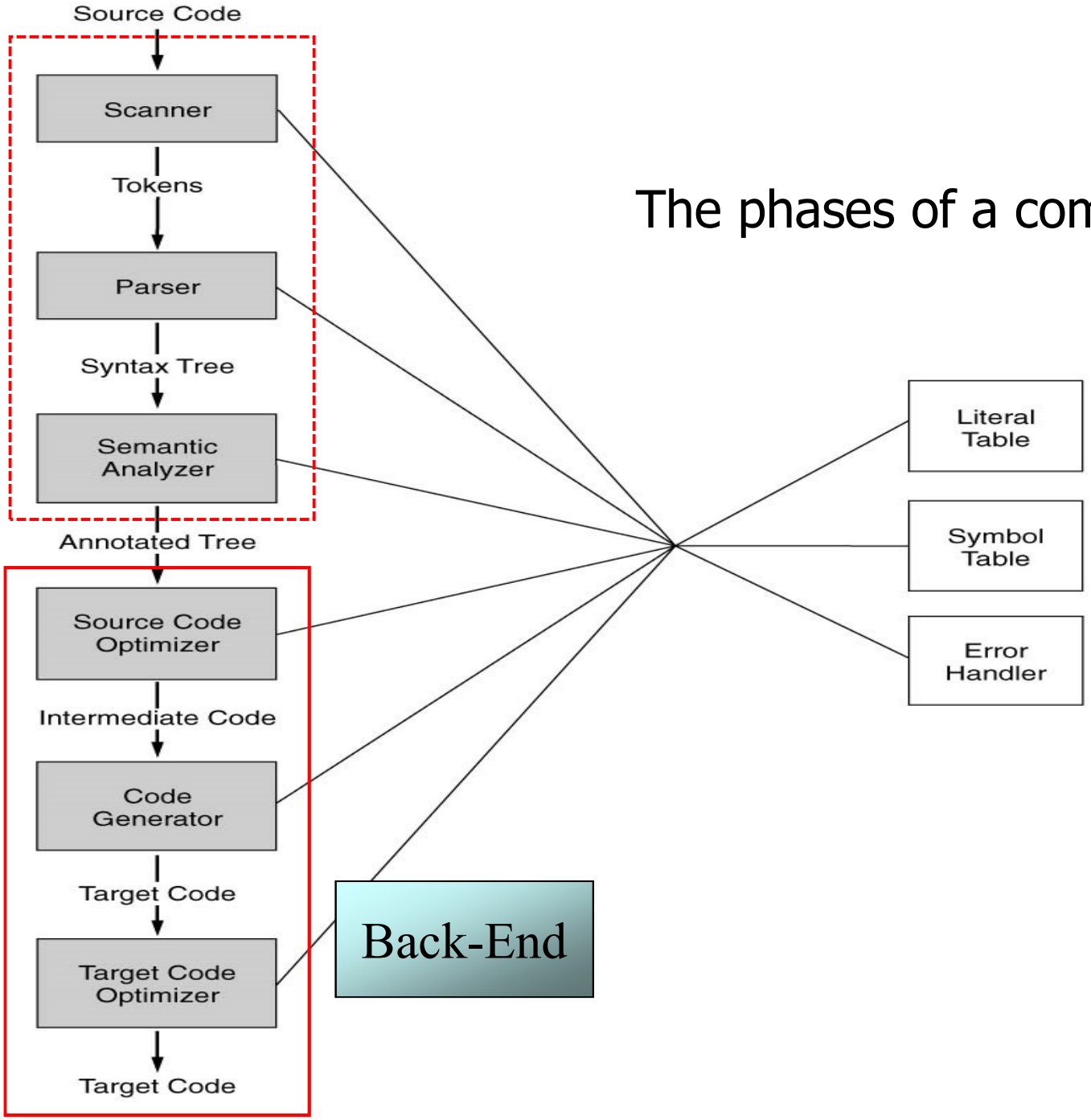
의미 분석 (**Semantic** Analyzer)



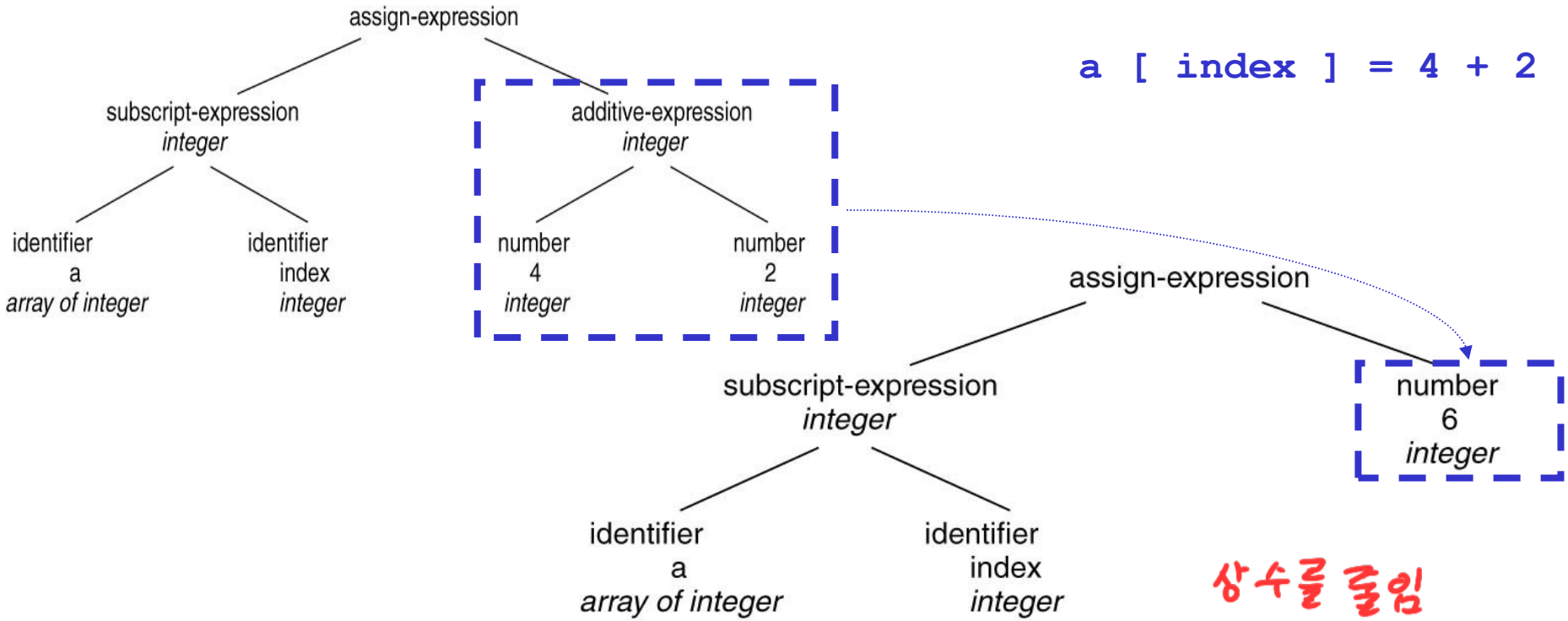
Front-end에서 나온 용어 정리

- Lexical : spelling or word (어휘, vocabulary)
  - Lexical analysis ➔ scanner (*a pet name*)
- Syntactic : what does it show?
  - 구문 (structure of a sentence) : 문장 구조
    - Syntactic analysis ➔ parser
    - Parse : 분석(analysis)
- Semantic : what does it mean? (의미, meaning)
- Sentence(문장)와 expression(수식)
  - 수식은 문장의 일부
  - 예: if (*conditional-expr*) *stmt-1* else *stmt-2*

The phases of a compiler



코드 최적화(Source Code ***Optimizer***)



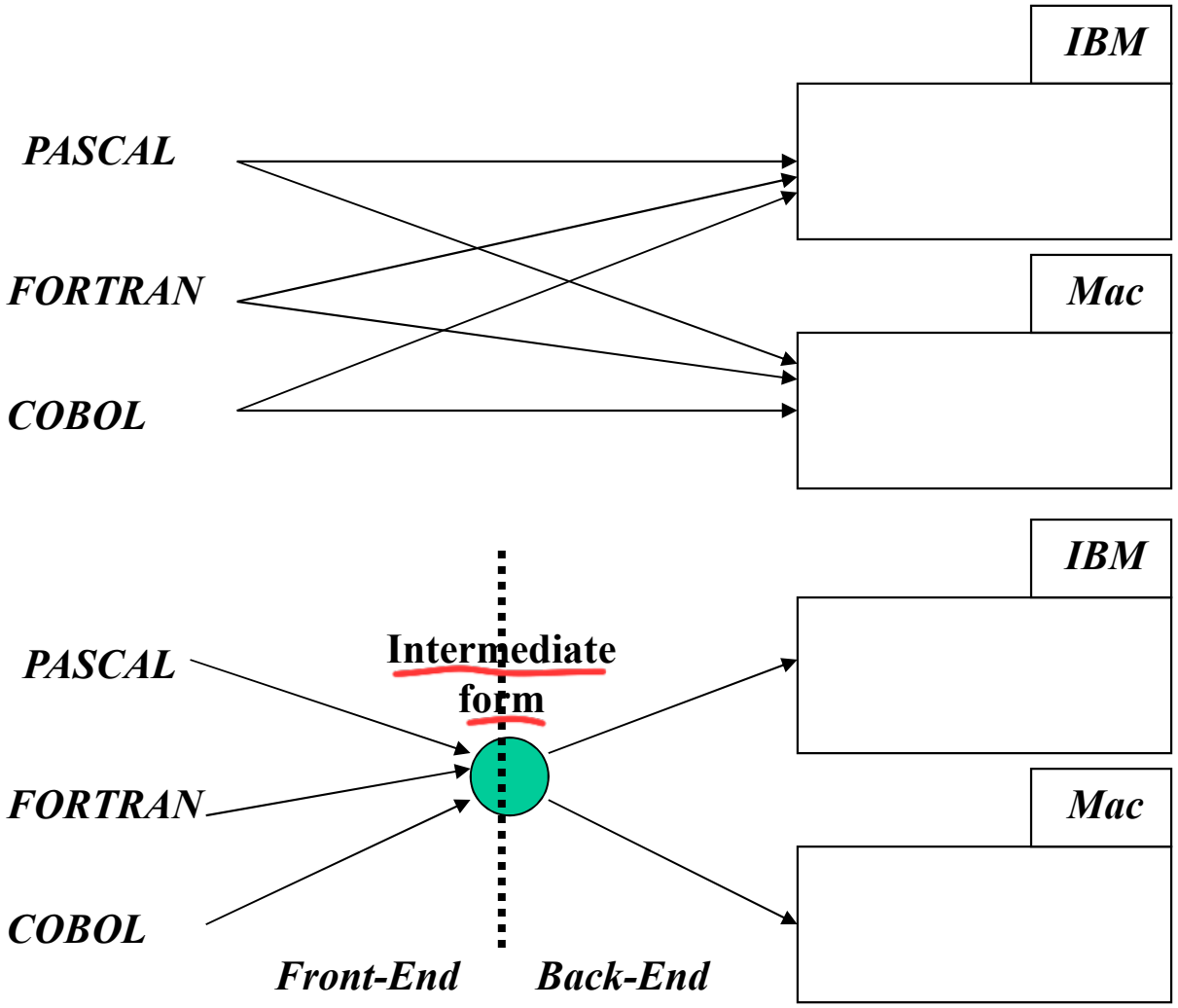
상수를 줄임

Constant Folding

4 + 2 can be pre-computed by the compiler to the result **6**

전역 최적화(*global optimization*) 라고도 함

Compiling to Intermediate Form



중간 코드 (*intermediate* code)

- 구문 트리를 직접 목적 코드로 번역하는 대신 기계 독립적인 중간 코드로 번역
  - 중간 코드 생성 ➔ 코드 최적화 ➔ 기계 코드 생성
  - 예 : 3-address code, P-code 등
  - *3-address code* for `a[index] = 4 + 2`

```
t = 4 + 2    // t : 임시 변수
a[index] = t
```

  - 덧셈 결과를 미리 계산(constant folding)

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

  - 임시 변수 t 대신 상수 6을 직접 사용

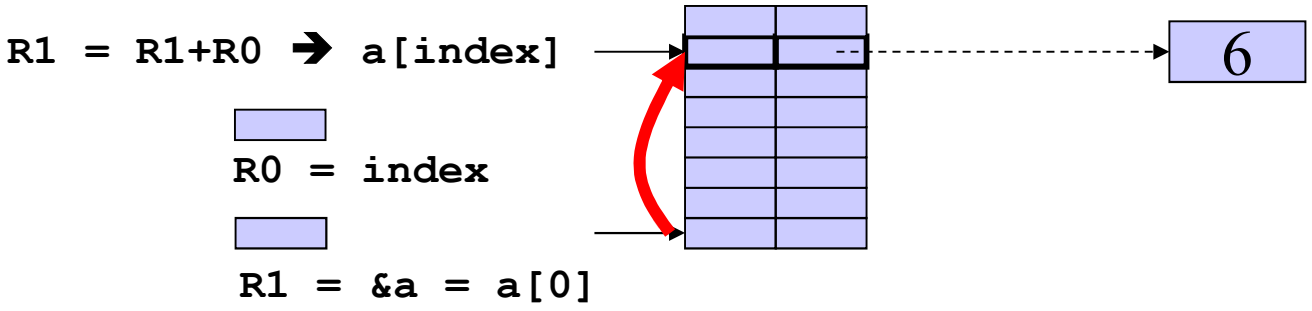
```
a [index] = 6
```

코드 생성(code generator)

- Use instructions as they exist on the target machine
  - Target machine의 속성에 따라 크게 좌우됨
    - 정수형 변수 및 실수형 변수는 몇 바이트를 차지하는가?
    - 배열의 인덱싱을 위한 addressing 방식은 무엇인가?
    - CPU 내부에 register는 몇 개나 있는가?
      - <참고> register allocation

코드 생성 예

```
MOV R0, index   ;; value of index -> R0
MUL R0, 2       ;; double value in R0(정수가 2bytes를 차지)
MOV R1, &a      ;; address of a -> R1 ↝ 기준위치
ADD R1, R0      ;; add R0 to R1 → 더해서 이동
MOV *R1, 6      ;; constant 6 -> address in R1
```



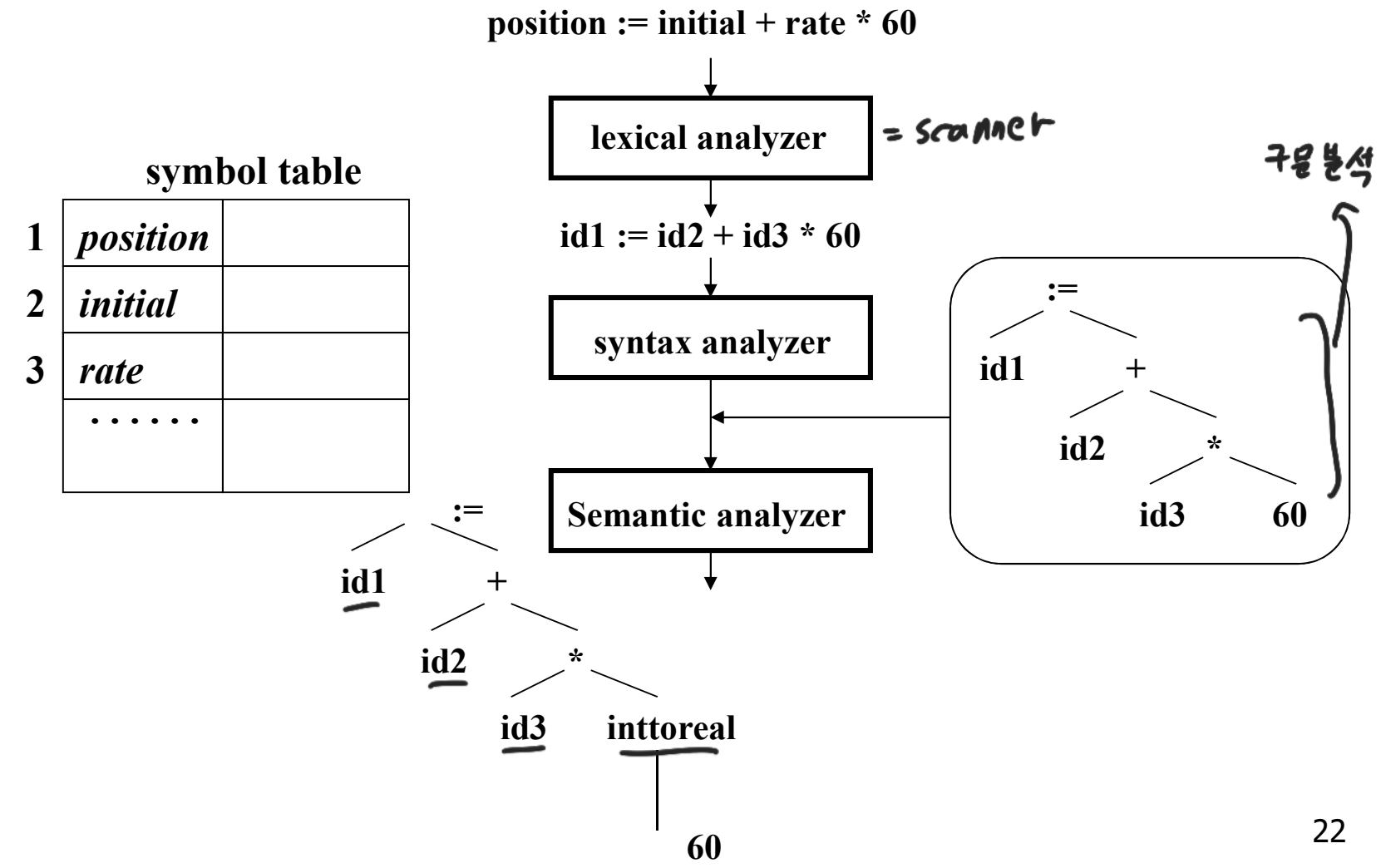
a [index] = 4 + 2

목적 코드 최적화(target code optimizer)

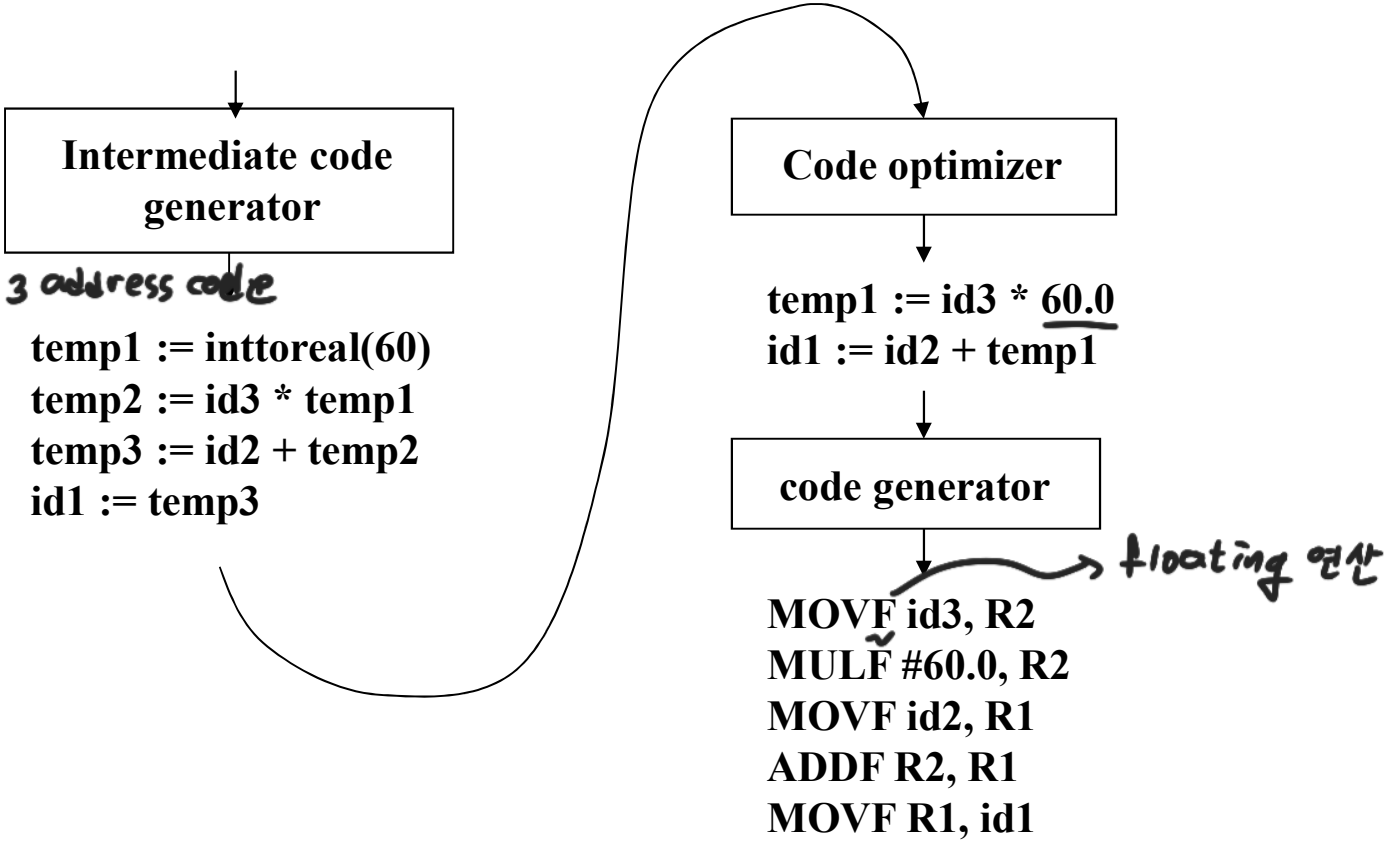
- 성능 향상 방법
  - 성능 향상을 위한 addressing 모드 선택
  - 실행 속도가 빠른 명령어로 대체
  - 중복 또는 불필요한 연산 제거
- 예:
  - 곱셈 명령어(**MUL**) → shift 명령어 (**SHL**)
  - 배열에서 index addressing 모드 사용

```
MOV R0, index       ;; value of index -> R0
SHL R0, 2           ;; double value in R0
MOV &a[R0], 6       ;; constant 6 -> address a + R0
```

예 : 단계별 컴파일 과정(1)


position := initial + rate * 60
lexical analyzer
= scanner
symbol table
1 position
2 initial
3 rate
. . . . . .
id1 := id2 + id3 * 60
syntax analyzer
Semantic analyzer
:=
id1
+
id2
*
id3
inttoreal
60
구문분석
:=
id1
+
id2
*
id3
60

예 : 단계별 컴파일 과정(2)


Intermediate code generator
3 address code
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
Code optimizer
temp1 := id3 * 60.0
id1 := id2 + temp1
code generator
floating 연산
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1