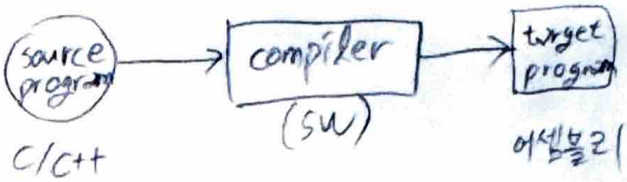
1. Course Overview

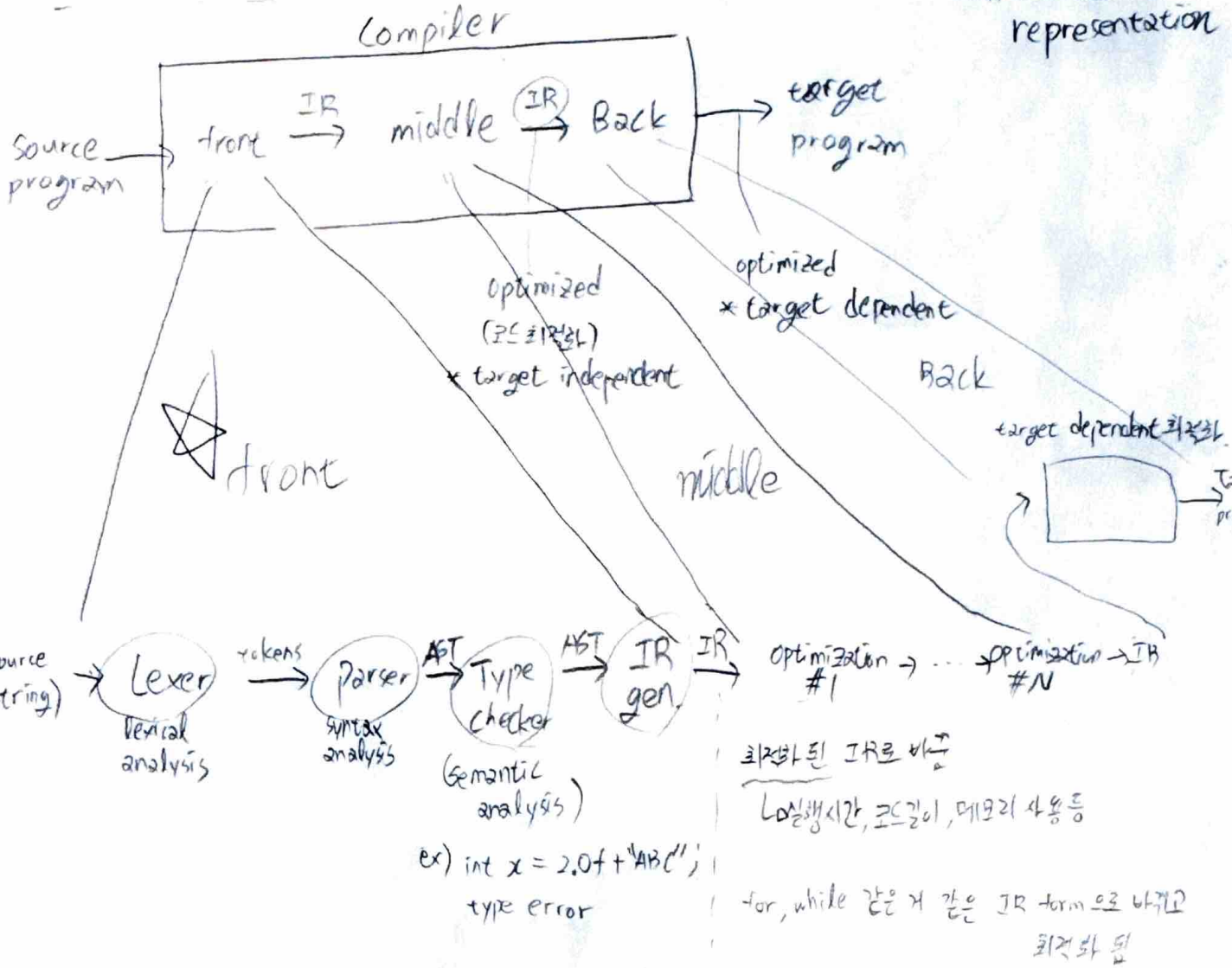


Compiler VS interpreter

translates input program into another program

directly executes the input program

IR=intermediate representation



ex) int x = 2.0f + "ABC";
type error

2. Lexical Analysis

Lexer : 주어진 문자열을 토큰으로 split.

이 과정에서 토큰의 type 과 실제값을 알아냄.

(id, 숫자, ...) (x, 10)

Specification of lexical analyzer : RegEx

명세..

Excercise) RegEx 해석?

$\Sigma = \{ 'a', 'b' \}$

① $R = (a|b)^* b$

L(R) : b로 끝나는 문자열

② $R = (a \cdot (ba)^*)(b|\varepsilon)$

$L(R) = \{ a, ab, aba, \cdots \}$

⇒ a 시작, 번갈아 나오는 문자열

이제, RegEx로 token 정의했음. ex) operator = '='|'+'|'-'| ...

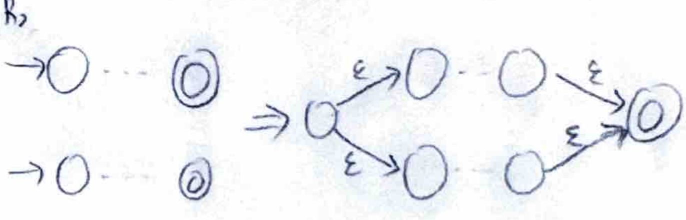
이제 확인하는 코드 짜야 함.

Automatic Generator: RegEx → NFA → DFA → Code (Lexer)

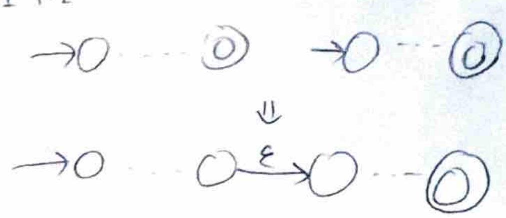
(Specification)

string ↓ Code (Lexer) ↓ token

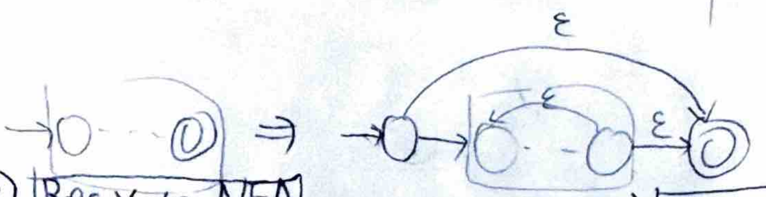
$R_1 | R_2$



$R_1 \cdot R_2$



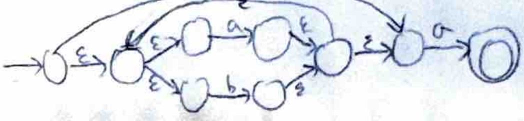
R_1^*



ex) NFA에서 Keyword, Identifier 둘 다 되면 먼저 정의된 걸로 생각

Exercise) RegEx to NFA

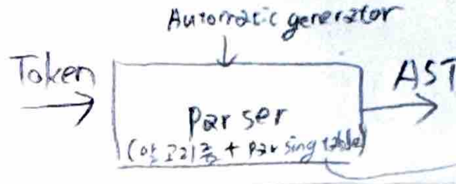
$(a|b)^* a$



Exercise) NFA to DFA

S_{12} 면 1, 2에서 a로 갈 수 있는 곳 (1,2,3) 구하고 E({1,2,3})으로 —a→ 주기.

3. Top-down Parsing



CFG에서 Automatic generator로 Parsing Table 생성

Specification of syntax analyzer(parser) : context-free grammar

③탑다운 ④바텀업

CFG. $E \to E+E \mid id \mid num$

$E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+num$

1단계씩

$E \Rightarrow^* id+num$

arbitary 단계..

용어) Terminal symbol: +, x, id, num

Non-terminal symbol: E, T, ...

또는 terminal도, nonterminal도 아닌 특수 기호!

Sentence : non terminal 없는 거. ex) id + num

$L(G) = \{ w \mid S \Rightarrow^* w$, w는 터미널로만 구성$\}$

(시작 심볼)

Leftmost derivation

Derivation 하면 parse tree 나옴 (AST는 아님)

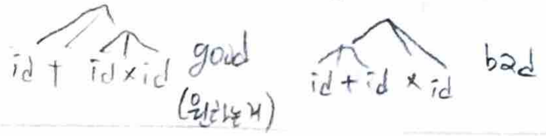
RegEx vs CFG

불가 ← {{ }} 이런거 가능

accept, reject만 + parse tree 까지

CFG 모호성

left derivation 다른 방식으로 같게 나옴



○ 다시 쓰자 목표 ① 모호성 삭제, ② *를 +보다 먼저 묶기, ③ *랑 +는 왼쪽부터 묶기

Unfortunately, 일반적인 방법은 없어. 심지어 모호한지 확인하는 일반적인 방법도 없음.

지금부터 unambiguous만 볼거임

왼쪽부터 다음토큰 1개

LL(1) Parsing

leftmost

: 각 단계에서 가장 왼쪽 nonterminal 다시 써야함.

⇒ 뭐 적용해야하지? ⇒ Parse table 만들고, 그대로..

* LL(1)은 모든 CFG에 적용할 수 있진 않음.

* Left recursion : 있으면 LL(1) 아님.
  (나중에 배운다?)

  $E \rightarrow E + T \mid T$

  $T \rightarrow T * F \mid F$

⇒ 수정

$A \rightarrow A\alpha \mid \beta$

to

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

근데, left recursive 아니어도 LL(1) 아닐 수도 있지만 강의에서는 많음... 그니까 넘어가고

stack (top)	input token
E $	num+id $
T E' $	num + id $
~~num~~ T'E'$	~~num~~+id $
$	$ 면 끝!

만약 스택 top = token이면 둘다 pop

← 기록해두면 derivation 알 수 있겠지. 그럼 parse tree 나옴.

First (α)은 easy,,, 첫글자 가능한 거 (ε포함 가능)

Follow(X) :
(X : non-terminal)

⇒ 이용해서 Parse table 만들기...

$X \rightarrow \alpha$ 있으면

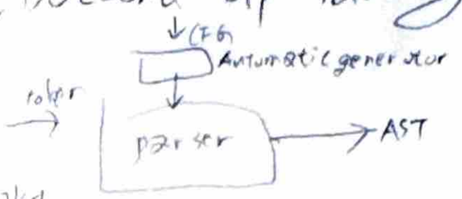
First(α)인 t에 대해 X, t에 $X \rightarrow \alpha$ 추가

⇒ t에 ε도 있으면 p∈Follow(X)에 대해 X, p에 $X \rightarrow \alpha$ 추가.

같은 칸에 여러 rule 들어가면 올바른 파싱테이블 만들기 실패한 거고, input CFG가 LL(1) grammar가 아닌거임.

≙ First(αi)끼리 교집합 있는 경우 $X \rightarrow \alpha_1 \mid \cdots \mid \alpha_n$ 에서 or First(A)랑 Follow(A)랑 교집합 있는 경우

4. Bottom-up Parsing

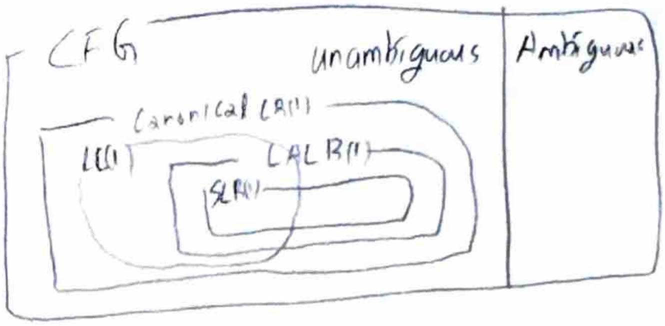


여기서 parenthesis 처리하는 CFG 사용, id만 사용 (num은 ✗), left-recursion 삭제할 필요 없음.
(괄호)

Right derive
LR(1) parsing
파가우

- SLR(1) : simple
- LALR(1) : Look Ahead
- Canonical LR(1) : 복잡

↓ powerful ↑ simple



State stack (top →)	Symbol stack (top)	Input Tokens	
0		$id_1 \times id_2$ $	s5
0 5	id_1	$\times id_2$ $	reduce (pop / pop & push)
0	F	$\times id_2$ $	g 3
0 3	F		
		$	→ success!
0 1	E		

pop 개수 동일해야함.

reduce + g 3 → 한 step

shift k : token 옮기고 상태에 k push (추가만 있음...)

parsing table 만들기...

① S→E 추가 : 이러면 parsing 성공 여부 간단하게 알 수 있음

② LR(0) items.. 진행 위치 느낌?

A→B(•)D : BC가 top이고 D가 추가되면 reduce 가능...

LR(0)들로 DFA 만들고 DFA로 파싱테이블 만들기임.
(DFA: state는 LR(0) item들의 집합)

Closure(I) : I ∪ (X→α•Xβ 가 있고, X→σ 이 있으면 X→•σ 추가 until 변화 ✗)

Next(I, E) = E 넘어 •한 칸씩 민 거의 closure ⇒ 이걸로 DFA 생성 ⇒ SLR(1) Parse table 생성

(I) —x→ (J)
- x가 terminal : sJ를 (I, x)에 추가   I에 있는 A→B• 에 대해
- x가 non-terminal : gJ를 (I, x)에 추가   x∈Follow(A)인 (I, x)에 rN 추가