제 4 장

어휘분석과 구문 분석

제 4 장 어휘분석과 구문분석

❖ 목적 : 자세한 것은 컴파일러 시간에(내용 조금 줄임)

- 컴파일러의 구조
- 어휘 분석
- 파싱 문제
- 상향식 파싱

❖ 4장의 내용을 프로그래밍 언어의 교재에 포함시킨 이유

- 구문 분석기는 3장에서 논의된 문법에 직접 기반한다.
- 어휘 분석기와 구문 분석기는 컴파일러 설계가 아닌 많은 상황에서 필요하다.
  - ✓ **간단한** interface program, text searching, website filtering, word processing 그리고 Command interpreter 등에 사용
  - ✓ 따라서 소프트웨어 개발자들이 컴파일러를 작성하지 않아도 어휘 분석과 구문 분석은 그들에게 매우 중요한 주제이다.

4.1 서론

- **컴파일러의 구조에 대한 견해 :**
  - 분석(Analysis)-합성(Synthesis) 모델 : 논리적인 구성 측면
  - 전단부(Front-end)-후단부(Back-end) : 기계 독립(machine independent) or 기계 종속(dependent), 구현측면
- **컴파일러의 논리적 구조**

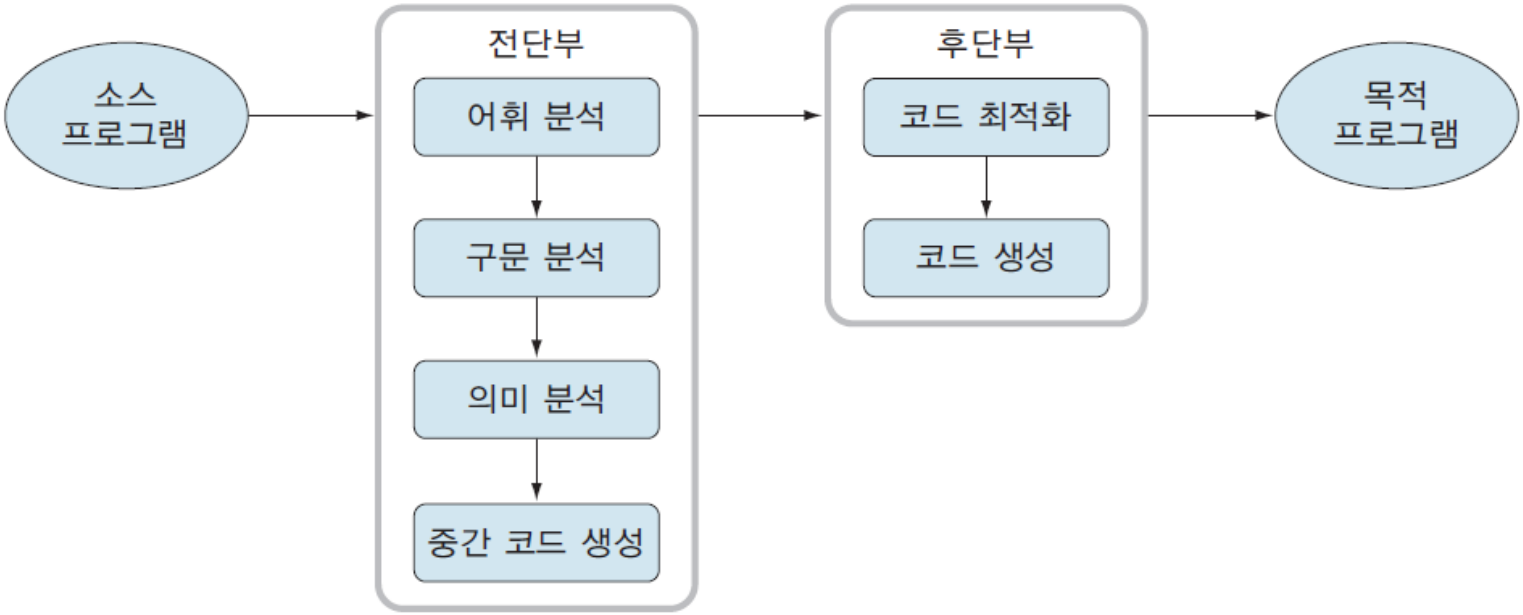


그림 2-2 컴파일러의 구체적 구조

4.1 서론

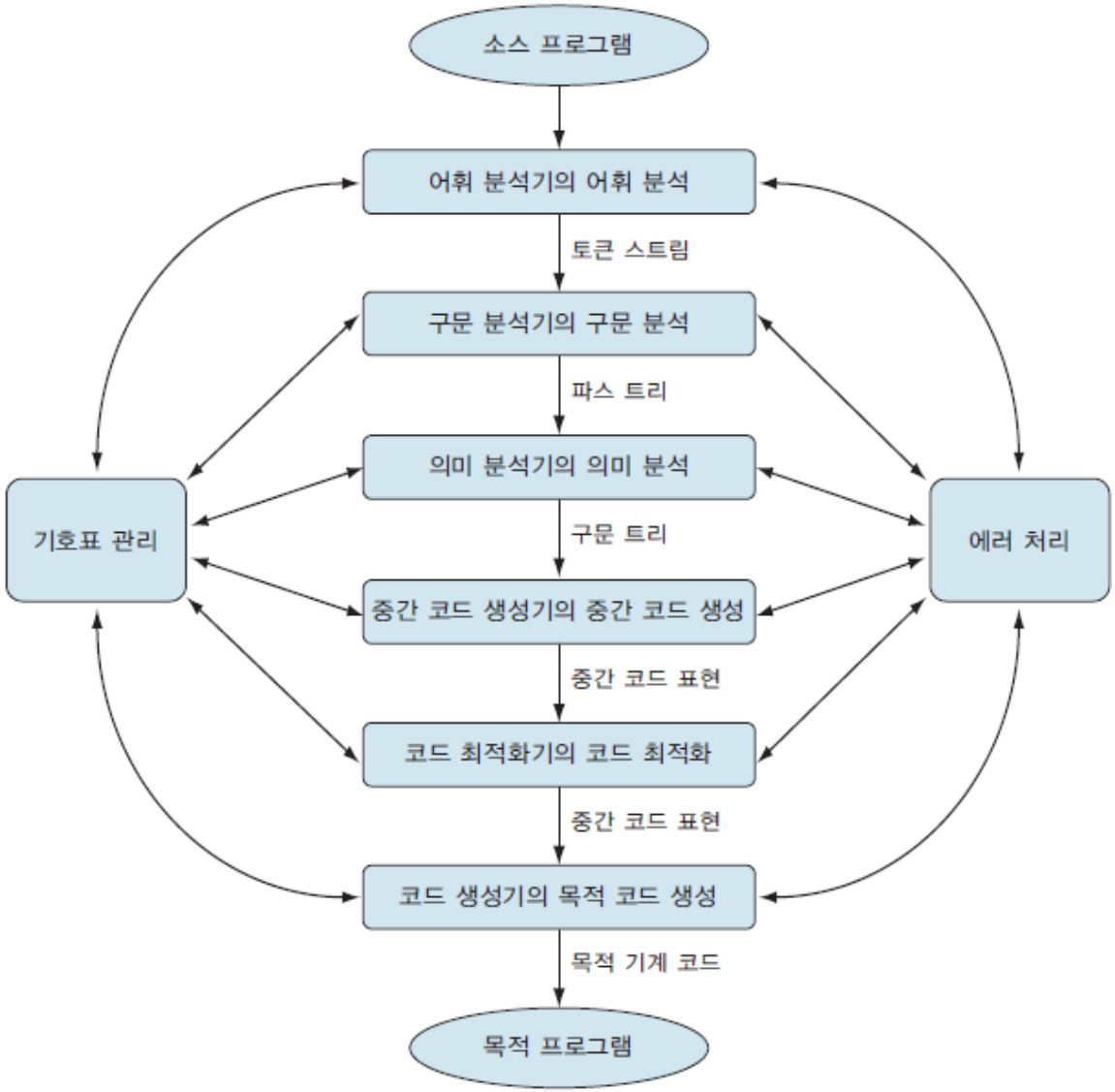


그림 2-3 컴파일러의 논리적 구조

4.1 서론

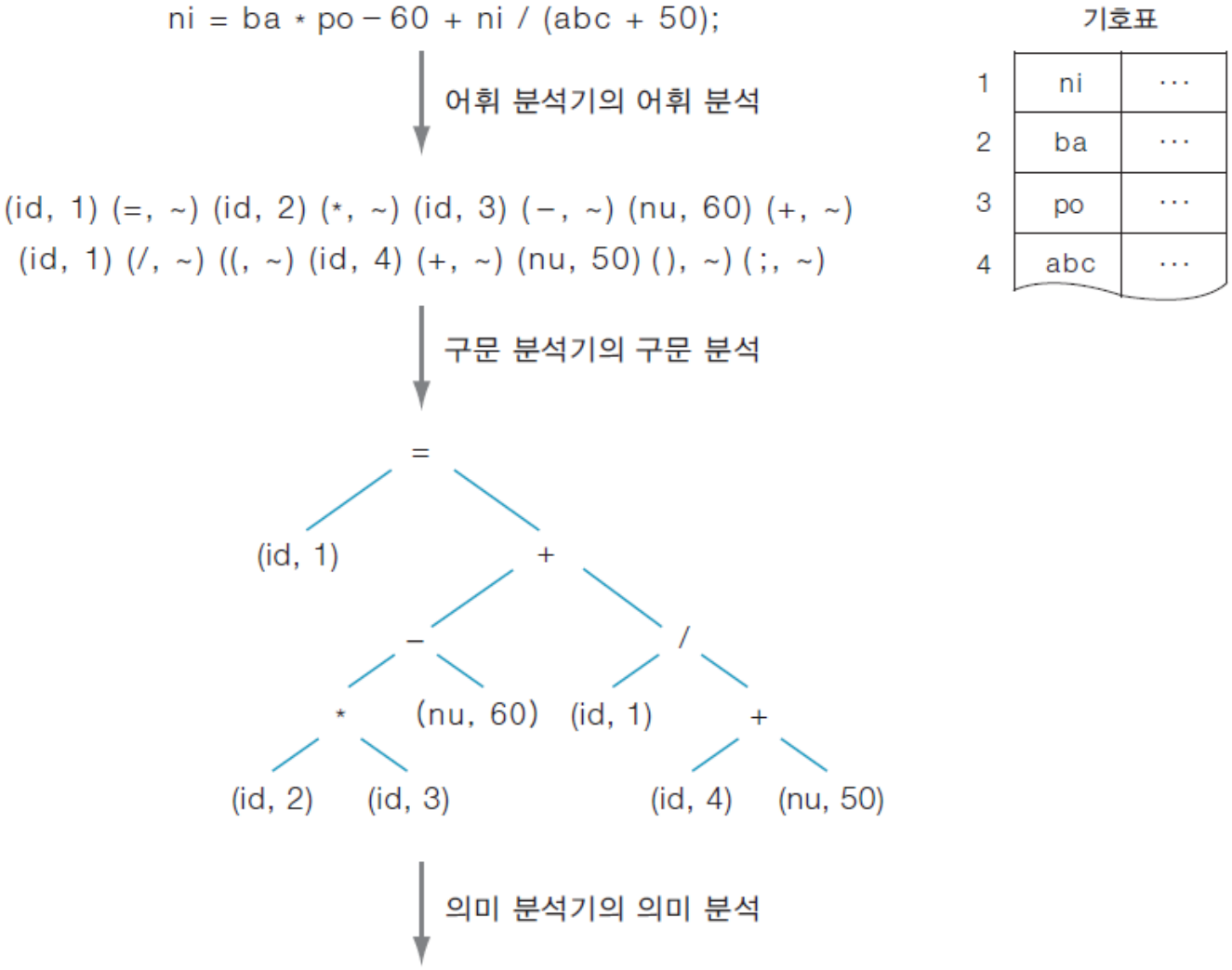
■ 아주 간단한 C 문법

❶ ⟨Sub C⟩ ::= ⟨assign–st⟩
❷ ⟨assign–st⟩ ::= ⟨lhs⟩ = ⟨exp⟩;
❸ ⟨lhs⟩ ::= ⟨variable⟩
❹ ⟨exp⟩ ::= ⟨exp⟩+⟨term⟩ | ⟨exp⟩–⟨term⟩ | ⟨term⟩
❺ ⟨term⟩ ::= ⟨term⟩*⟨factor⟩ | ⟨term⟩/⟨factor⟩ | ⟨factor⟩
❻ ⟨factor⟩ ::= ⟨variable⟩ | ⟨number⟩ | (⟨exp⟩)
❼ ⟨variable⟩ ::= ⟨ident⟩
❽ ⟨ident⟩ ::= (⟨letter⟩ | _){⟨letter⟩ | ⟨digit⟩ | _}
❾ ⟨number⟩ ::= {⟨digit⟩}
❿ ⟨letter⟩ ::= a | … | z
⓫ ⟨digit⟩ ::= 0 | 1 | … | 9

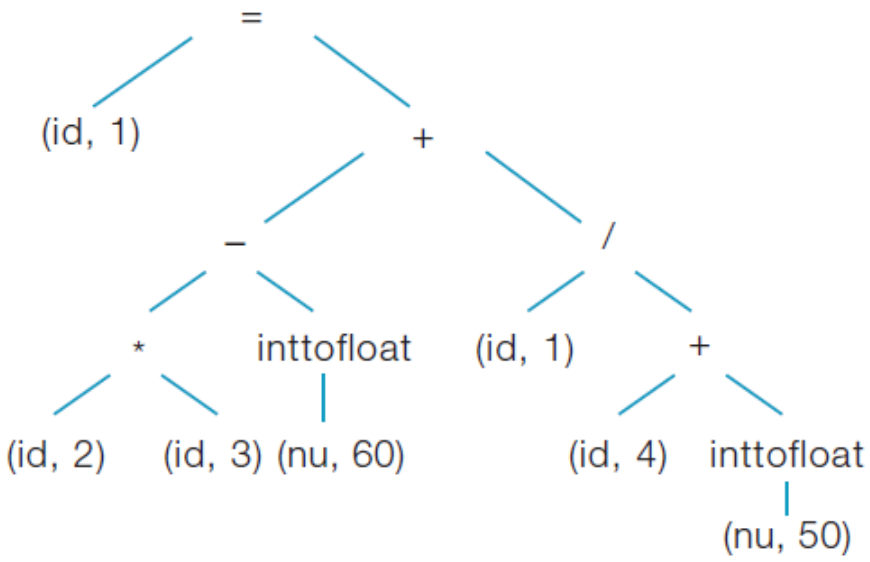
그림 2–4 아주 간단한 C 문법

4.1 서론

- 치환문 ni = ba * po - 60 + ni / (abc + 50);에 대한 컴파일러 도식화



4.1 서론



중간 코드 생성기의 중간 코드 생성

```
t1 = id2 * id3
t2 = inttofloat(60)
t3 = t1 − t2
t4 = inttofloat(50)
t5 = id4 + t4
t6 = id1 / t5
t7 = t3 + t6
id1 = t7
```

코드 최적화기의 코드 최적화

```
t1 = id2 * id3
t3 = t1 − 60.0
t5 = id4 + 50.0
t6 = id1 / t5
id1 = t3 + t6
```

코드 생성기의 목적 코드 생성

```
LOAD   R2, id2
LOAD   R1, id3
MULT   R2, R1
SUBT   R2, #60.0
LOAD   R1, id4
ADD    R1, #50.0
STORE  W1, R1
LOAD   R1, id1
DIV    R1, W1
ADD    R2, R1
STORE  id1, R2
```

그림 2-5 치환문의 컴파일링 과정

4.2 어휘분석

어휘분석(lexical analysis, scanning)

- 원시 프로그램을 읽어 들여 토큰(token)이라는 의미 있는 문법적 단위(syntactic entity)로 분리
- 어휘분석을 담당하는 도구(tool)를 어휘 분석기(lexical analyzer) 혹은 스캐너(scanner)라고 부른다.
- 토큰 : 문법적으로 의미있는 최소 단위로 문법에서 터미널 기호
  - ✓렉심, 패턴 (3장에서 이미 다룸)
  - ✓배정문 result = oldsum – value / 100; (182 참조)

어휘 분석기를 구성하는 두 가지 방법

- 정규 표현(regular expression)을 이용하여 토큰 패턴들에 대해서 기술하고 이는 lex(flex)와 unix(bison) 시스템의 입력으로 사용.
- 토큰 패턴들을 기술하는 상태 전이도를 설계하고 이 다이어그램을 구현하는 프로그램을 작성

4.2 어휘분석

- 토큰 패턴들을 기술하는 상태 전이도를 설계하고 이 다이어그램을 구현하는 프로그램을 작성

✓상태 전이도

- 유향 그래프로 상태 전이도의 노드는 상태 이름이고, 간선(edge, ark)은 상태들간의 전이를 일으키는 입력 문자들로 구성

✓정규 언어(regular language)를 정규 표현으로

✓정규 표현을 받아들이는 상태 전이도

✓이런 상태전이도를 유한 오토마타(finite automata)

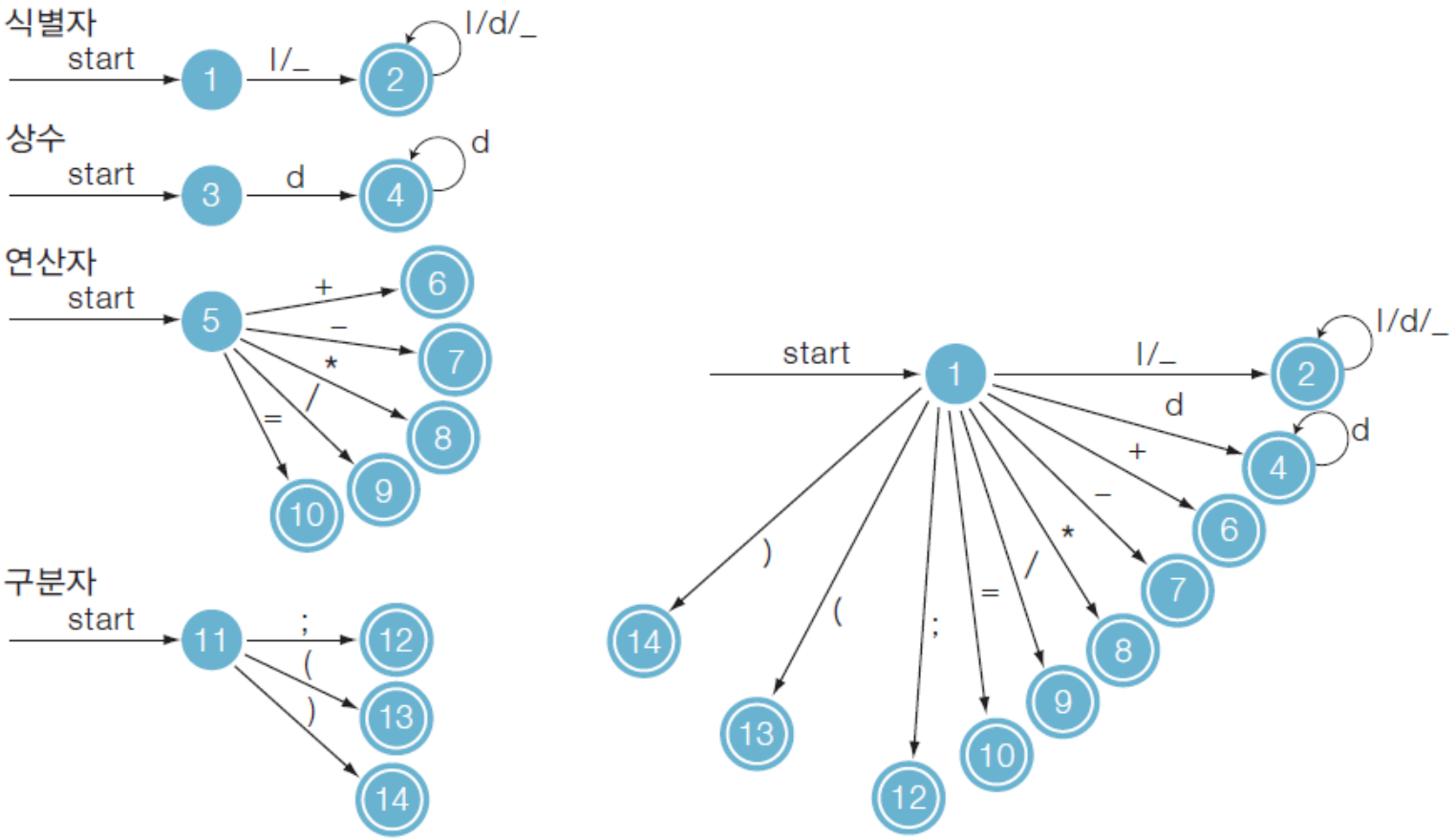
✓어휘분석기는 유한 오토마타

4.2 어휘분석

- [그림 2.4]의 문법에서 예약어가 허용되는지를 확인한다, [그림 2.4]에서는 예약어를 사용하지 않는다. 상수를 보면, 0이나 양의 정수를 허용한다. 연산자는 4칙 연산자인 +, -, *, / 를 사용하고 치환연산자인 =를 허용한다. 식별자도 허용하고 첫 자는 영문자나 언더바로 시작하고 두 번째 자 부터는 영문자나 숫자, 언더바를 사용하고 길이에는 제한이 없다. 구분자로는 세미콜론만이 사용된다. 이들을 정규 문법과 정규 표현으로 나타내면 다음과 같다.
  - 1) 식별자 : 정규 문법
  - S → lA | _A
  - A → lA | dA | _A | ε 로부터 정규표현으로 나타내기 위해서 정규 표현 방정식은 다음과 같다.
  - S = (l + _)A
  - A = lA | dA | _A | ε = (l + d +_)A + ε = (l + d +_)*
  - 이것으로부터 해를 구하면
  - S = (l + _)A = (l + _)(l + d +_)*
  - 2) 상수 : S= $(d)^{+}$
  - 3) 연산자 : S = + | - | * | / | '='
  - 4) 구분자 : ;, (,)

4.2 어휘분석

■ 이들을 받아들이는 유한 오토마타는 다음과 같다.



(a) 각각의 토큰에 대한 어휘 분석기 (b) 전체 토큰에 대한 어휘 분석기

그림 2-6 [그림 2-4]에 대한 어휘 분석기

4.2 어휘분석

- 어휘 분석기 구현 – 185페이지~190 참조
- 190 페이지 예제
  - ✓(sum + 470 / total
  - ✓결과 – 책 참조
- 토큰에 대한 표현은 (토큰 번호, 토큰 값)의 순서쌍으로 표현

(id, 1) (=, ~) (id, 2) (*, ~) (id, 3) (-, ~)(nu, 60) (+, ~) (id, 1) (/, ~) ((, ~) (id, 4) (+, ~) (nu, 50) (), ~) (;, ~)

- 주석(comment)도 이 과정에서 처리

4.3 파싱 문제

구문분석(syntax-analysis, 파싱(parsing))

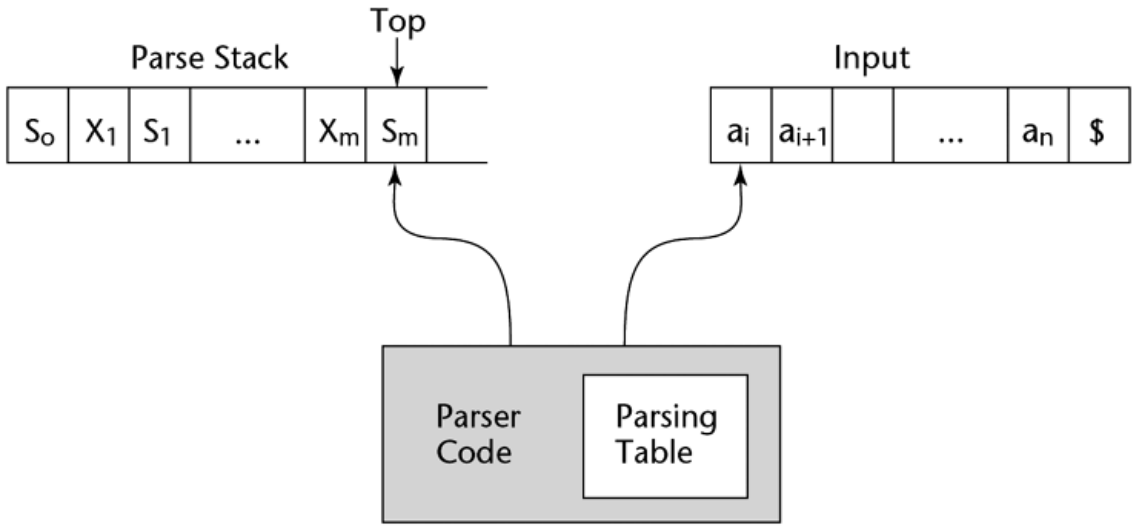
- 토큰들을 받아 이 토큰들이 주어진 문법(grammar)에 맞는지를 검사하고 올바른 문장에 대해서는 그 문장에 대한 파스 트리(parse tree)를 출력하고 올바르지 못한 문장에 대해서는 에러 메시지(error message)를 출력하는 과정을 구문 분석(syntax analysis) 혹은 파싱(parsing)
- 구문분석을 담당하는 도구(tool)를 구문 분석기(syntax analyzer) 혹은 파서(parser)라고 부른다.
- 구문 분석의 종류 –
  - 파스 트리를 생성하는 방향에 따라 하향식(top-down; 파스 트리가 루트 노드로 부터 잎 노드를 만드는 방법), 상향식(bottom-up; 파스 트리를 이 노드로 부터 루트 노드를 만드는 방법)
- 하향식 파싱 –
  - 백트래킹(backtracking) 발생으로 컴파일 시간 늦음
  - 문장, 문장 형태, 유도, 좌단 유도
  - 재귀하강 파서(recursive-descent parser), LL 알고리즘
- 상향식 파싱 –
  - 핸들(Handle) – 감축(reduce) 되는 부분
  - 우단 유도
  - 이동-감축(shift-reduce), LR 파서(LR 파서의 구조 – 그림 4.4)

4.3 파싱 문제

■

Figure 4.4

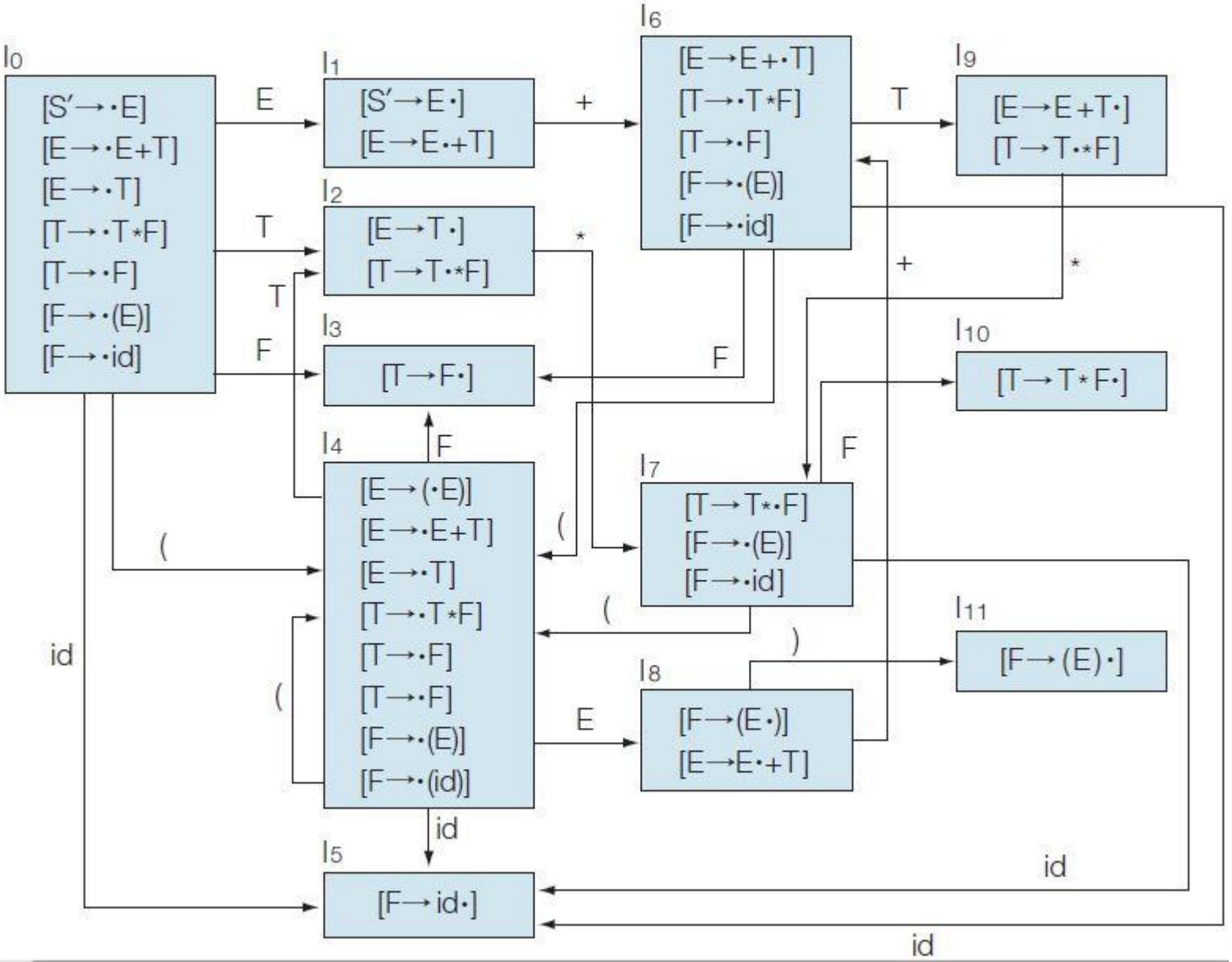
The structure of an LR parser



4.5 상향식 파싱

- 문법을 줄여서 해보자. **SLR 파싱표 구하기**
  - ✓ (1) 증가 문법
  - ✓ 0. S' → E
  - ✓ 1. E → E + T
  - ✓ 2. E → T
  - ✓ 3. T → T * F
  - ✓ 4. T → F
  - ✓ 5. F → (E)
  - ✓ 6. F → id

4.5 상향식 파싱



4.5 상향식 파싱

- FOLLOW 의 계산 :

  FOLLOW(E) = {$, +,)}

  FOLLOW(T) = {*, +,), $}

  FOLLOW(F) = {*, +,), $}

  ✓

4.5 상향식 파싱

- 212페이지
  - 파싱

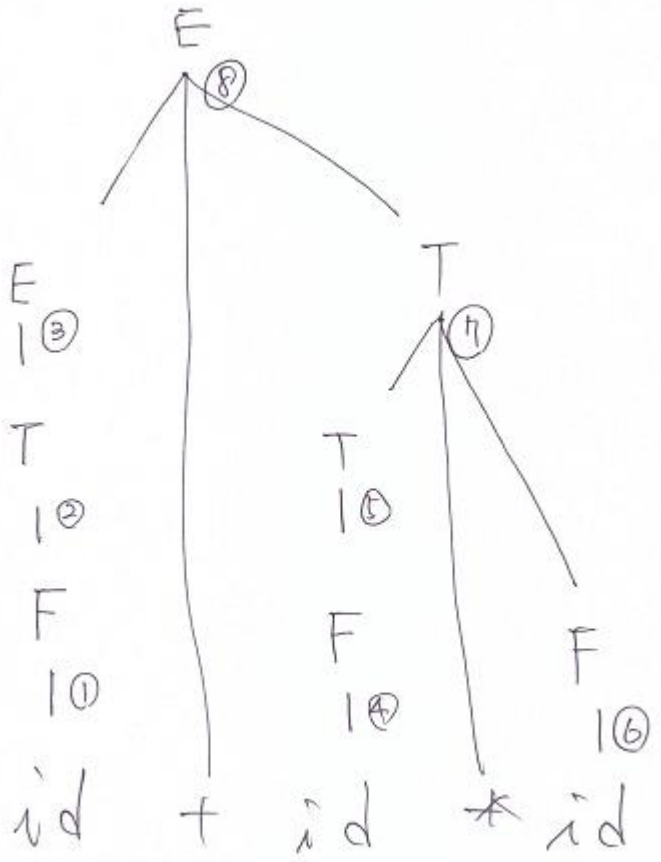
Figure 4.5

The LR parsing table for an arithmetic expression grammar

	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

4.5 상향식 파싱

- 문장 id+id*id에 대한 파싱
  - ✓ E ⇒ E + T
  - ✓ ⇒ E + T * F
  - ✓ ⇒ E + T * id
  - ✓ ⇒ E + F * id
  - ✓ ⇒ E + id * id
  - ✓ ⇒ T + id * id
  - ✓ ⇒ F + id * id
  - ✓ ⇒ id + id * id



4.5 상향식 파싱

- 파싱 과정 : 문장 id+id*id

step	스택	입력	행동		출력
0	0	id+id*id$	shift	5	
1	0id5	+id*id$	reduce	6	6
2	0F	+id*id$	GOTO	3	
3	0F3	+id*id$	reduce	4	4
4	0T	+id*id$	GOTO	2	
5	0T2	+id*id$	reduce	2	2
6	0E	+id*id$	GOTO	1	
7	0E1	+id*id$	Shift	6	
8	0E1+6	id*id$	Shift	5	
9	0E1+6id5	*id$	reduce	6	6
10	0E1+6F	*id$	GOTO	3	
11	0E1+6F3	*id$	reduce	4	4
12	0E1+6T	*id$	GOTO	9	
13	0E1+6T9	*id$	shift	7	
14	0E1+6T9*7	id$	shift	5	
15	0E1+6T9*7id5	$	reduce	6	6
16	0E1+6T9*7F	$	GOTO	10	
17	0E1+6T9*7F10	$	reduce	3	3
18	0E1+6T	$	GOTO	9	
19	0E1+6T9	$	reduce	1	1
20	0E	$	GOTO	1	
21	0E1	$	accept		