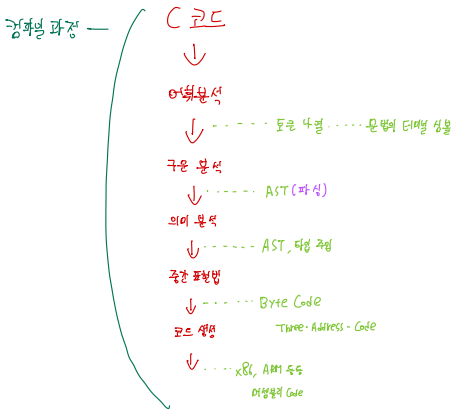
구문분석

목차

- 알파벳, 스트링, 언어, 문법
- 문맥자유문법(context free grammar, cfg)
- BNF 표기법
- 유도(derivation)
- 유도 연습
- Leftmost/rightmost derivation
- 유도의 트리 표현
- Parsing
  - Top-down vs. bottom-up
- parse tree, AST
- 문법
  - Ambiguous grammar
  - Precedence, associativity
    - C 언어 연산자 우선 순위, 결합 규칙
  - Left recursion
  - Left factoring
- Recursive descent parsing
- Shift-reduce parsing



Alphabet, String, Language, Grammar

- **알파벳**(**alphabet**)은 심볼(symbol)을 원소로 갖는, 공집합이 아닌, 유한집합이다. **스트링**(**string**)은 알파벳 내 심볼(들)의 유한 나열(possibly empty)이다. 스트링 w의 길이(**length**)는 w 내 심볼 개수이며 $|w|$로 표기한다. 어떤 심볼도 갖지 않는 스트링을 **empty string**이라 하며 λ 혹은 ε으로 표기한다($|\lambda| = |\varepsilon| = 0$). **언어**(**language**)는 스트링의 집합이다. 언어에 속하는 스트링을 **문장**(**sentence**)이라 한다.

예) 심볼 a, b를 원소로 갖는 알파벳 $\Sigma = \{a, b\}$
예) 알파벳 $\Sigma = \{a, b\}$로부터 얻을 수 있는 길이 1 이상 스트링들

$$a, b, aa, ab, bb, aaa, bbb, aab, baa, aaaaaaa, \ldots$$

예) 알파벳 $\Sigma = \{a, b\}$로부터 얻을 수 있는 언어

$$L = \{ab, aabb, aaabbb, \ldots\} = \{a^n b^n \mid n \geq 1\}$$

언어 생성 수단으로서의 문법
언어 $L = \{ab, aabb, aaabbb, \ldots\}$을 표현한 문법 G의 예

$G = (V, T, S, P)$
- Non-terminal 심볼 집합 $V = \{S\}$
- Terminal 심볼 집합 $T = \{a, b\}$
- 시작 심볼 S
- 생성규칙(production rule) 집합 $P = \{S \rightarrow aSb, S \rightarrow ab\}$

문법 G의 시작 심볼 S에서 시작하여 생성규칙들의 적용을 통해 언어 L 내 문장 $ab, aabb, aaabbb$들이 생성(generate)되는 **유도**(**derivation**) 과정들의 예

$S \rightarrow ab$

$S \Rightarrow ab$

aSb에 하나의 생성규칙 $S \rightarrow ab$가 적용되어 aSb가 $aabb$로 변경될 때 aSb로부터 $aabb$가 유도(derivation)되었다고 하고 $aSb \Rightarrow aabb$로 표기

$S \rightarrow aSb$ $S \rightarrow ab$

$S \Rightarrow aSb \Rightarrow aabb$

$S \rightarrow aSb$ $S \rightarrow ab$

$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb$

문맥자유문법

문법 $G = (V, T, S, P)$는 다음과 같이 정의된다
- V는 **variable(non-terminal symbol, 논터미널 심볼, 변수)**의 유한 집합
- T는 **terminal symbol(터미널 심볼)**의 유한 집합
- S는 **start variable(start symbol, 시작 심볼)**로 불리는 special symbol 이며 $S \in V$
- P는 **production rule(생성 규칙)**의 유한 집합으로 각 production은 $\alpha \rightarrow \beta$ 형태이며 α, β는 V, T 내 심볼로 이루어진 길이 0 이상 스트링이지만, α는 V 내 심볼을 하나 이상 포함하고 있어야 함. 다른 언급이 없으면 V와 T는 non-empty이며 disjoint

문맥자유문법(context-free grammar, cfg)
- $G = (V, T, S, P)$의 모든 생성규칙들이 $A \rightarrow \alpha$의 형식이면 문법 G는 context-free라고 한다. (A는 V의 원소이며 α는 V, T 내 심볼로 이루어진 길이 0 이상 스트링)

$G = (V, T, S, P)$
$V = \{E, T, F\}$
$T = \{+, -, *, /, (,), id\}$
$S = E$

- Nonterminal symbol(논터미널 심볼) → E, T, F
- Terminal symbol(터미널 심볼) → $+, -, *, /, (,), id$
- Start symbol(시작 심볼) → E
- 생성규칙 개수 → 8개

$P = \{E \rightarrow E + T, E \rightarrow E - T, E \rightarrow T, T \rightarrow T * F, T \rightarrow T / F, T \rightarrow F, F \rightarrow (E), F \rightarrow id\}$

이렇게 축약 가능

$E \rightarrow E + T \mid E - T \mid T$
$T \rightarrow T * F \mid T / F \mid F$
$F \rightarrow (E) \mid id$

right-hand side(RHS)
Left-hand side(LHS)

$F \rightarrow (E)$
$F \rightarrow id$

→ $F \rightarrow (E) \mid id$

- 문법 G를 위와 같이 생성규칙들만 나열하여 표현 가능
- 나열된 첫 생성규칙의 LHS 심볼이 시작 심볼이고, LHS가 동일한 규칙들은 기호 |를 이용하여 축약 표기한 것으로 가정하면 위 생성규칙 나열로부터 문법 G 복원 가능

BNF 표기법

BNF(Backus-Naur Form) 표기법

- BNF 표기법은 John Backus와 Peter Naur에 의해 개발되었으며 프로그래밍언어의 문법 정의에 사용됨
- 메타 기호로 **<**, **>**, **::=**, **|**를 사용하는데, 논터미널 심볼은 메타기호 **<**와 **>**로 묶어 표기하고, 메타기호 **::=**은 **is defined as**의 의미로 → 대신 사용하며, 메타기호 **|**은 **or**의 의미로 사용

```
E → E + T | E - T | T
T → T * F | T / F | F
F → ( E ) | id
```

BNF 표기법

```
<E> ::= <E> + <T> | <E> - <T> | <T>
<T> ::= <T> * <F> | <T> / <F> | <F>
<F> ::= ( <E> ) | id
```

Derivation

유도(derivation)

- 스트링 $w = u\alpha v$에 대해 **production rule** $\alpha \rightarrow \beta$가 적용가능(applicable)하다고 하며 $\alpha \rightarrow \beta$를 $w = u\alpha v$에 적용하면 w 내 α를 β로 대치(replace)하여 새로운 스트링 $z = u\beta v$ 를 얻게 되는데 이 과정을 $\boldsymbol{w \Rightarrow z}$로 표기하고 w가 z를 **유도한다(derive)**라고 하거나 혹은 z가 w로부터 유도되었다고 한다. Grammar의 production들을 임의 순서로 적용하여 새로운 스트링들이 유도될 수 있다

문법 G에 의해 생성되는 언어

- 문법 G에 의해 생성되는 언어는 G의 시작 심볼 S로부터 유도되는 모든 문장의 집합

아래 문법 G로부터 $abba$의 유도(과정)

$$G = (\{S, A, B\}, \{a, b\}, S, \{S \rightarrow aA \mid bB \mid \lambda, A \rightarrow bS, B \rightarrow aS\})$$

$$S \rightarrow aA \mid bB \mid \lambda$$
$$A \rightarrow bS$$
$$B \rightarrow aS$$

$$S \Rightarrow aA \Rightarrow abS \Rightarrow abbB \Rightarrow abbaS \Rightarrow abba$$

Sentential forms　　Sentence

$S \Rightarrow aA$　(생성 규칙 $S \rightarrow aA$ 적용)
$\Rightarrow abS$　(생성 규칙 $A \rightarrow bS$ 적용)
$\Rightarrow abbB$　(생성 규칙 $S \rightarrow bB$ 적용)
$\Rightarrow abbaS$　(생성 규칙 $B \rightarrow aS$ 적용)
$\Rightarrow abba$　(생성 규칙 $S \rightarrow \lambda$ 적용)

- S에 생성규칙 $S \rightarrow aA$가 적용되어 S가 aA로 변경될 때 S로부터 aA가 유도(derivation)되었다고 하고 $S \Rightarrow aA$로 표기
- aA에 생성규칙 $A \rightarrow bS$가 적용되어 aA가 abS로 변경될 때 aA로부터 abS가 유도(derivation)되었다고 하고 $aA \Rightarrow abS$로 표기

유도 연습 1

문법 $G = (V, T, S, P)$
$V = \{S, E, T, F, D\}$,
$T = \{+, *, (,), 0,1,2,3,4,5,6,7,8,9\}$

문법 G의 생성규칙집합 P

$S \rightarrow E$
$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow D$
$D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

- 문법 G를 사용하여 $1 + 2$의 유도과정을 보이시오

S
$\Rightarrow E$
$\Rightarrow E + T$
$\Rightarrow T + T$
$\Rightarrow F + T$
$\Rightarrow D + T$
$\Rightarrow 1 + T$
$\Rightarrow 1 + F$
$\Rightarrow 1 + D$
$\Rightarrow 1 + 2$

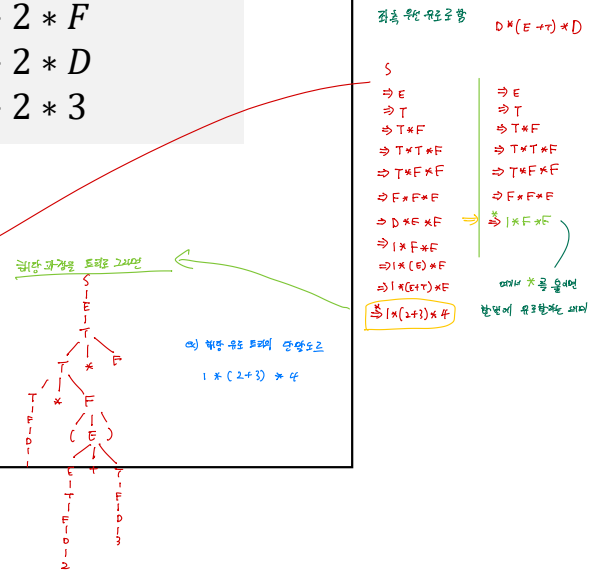
Sentential form 내 논터미널 심볼이 여러 개(예: E 와 T)인 경우 가장 좌측에 있는 논터미널(예: E)부터 규칙을 적용하기로 함
→ 좌단유도라고 함
(뒤에 다시 설명됨)

- 문법 G를 사용하여 $1 + 2 * 3$의 유도과정을 보이시오

S
$\Rightarrow E$
$\Rightarrow E + T$
$\Rightarrow T + T$
$\Rightarrow F + T$
$\Rightarrow D + T$
$\Rightarrow 1 + T$
$\Rightarrow 1 + T * F$
$\Rightarrow 1 + F * F$
$\Rightarrow 1 + D * F$
$\Rightarrow 1 + 2 * F$
$\Rightarrow 1 + 2 * D$
$\Rightarrow 1 + 2 * 3$

- 문법 G가 생성하는 문장의 예를 보이시오

$1 * (2 + 3) * 4$



유도 연습 2

문법 $G = (V, T, S, P)$
$V = \{Program, Stmts, S, ID, NUM\}$,
$T = \{\{, \}, =, ;, print, (,), a, b, c, 0,1,2,3,4,5,6,7,8,9\}$

문법 G의 생성규칙집합 P

$Program \rightarrow \{\ Stmts\ \}$
$Stmts \rightarrow S\ Stmts \mid \varepsilon$
$S \rightarrow ID = NUM\ ;$
$S \rightarrow print\ (\ ID\ );$
$ID \rightarrow a \mid b \mid c$
$NUM \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

- 문법 G로부터 $\{c = 3\}$을 유도해 보시오
- 문법 G로부터 $\{c = 3; print(b);\}$를 유도해 보고, 프로그래밍언어 관점에서 어떤 문제가 있는지 생각해 보시오.

- 문법 G를 사용하여 $\{a = 7;\}$의 유도과정을 보이시오

$Program$
$\Rightarrow \{Stmts\}$
$\Rightarrow \{S\ Stmts\}$
$\Rightarrow \{ID = NUM; Stmts\}$
$\Rightarrow \{a = NUM; Stmts\}$
$\Rightarrow \{a = 7; Stmts\}$
$\Rightarrow \{\boldsymbol{a = 7;}\}$

- 문법 G를 사용하여 $\{b = 5; print(b);\}$의 유도과정을 보이시오

$Program$
$\Rightarrow \{Stmts\}$
$\Rightarrow \{S\ Stmts\}$
$\Rightarrow \{ID = NUM; Stmts\}$
$\Rightarrow \{b = NUM; Stmts\}$
$\Rightarrow \{b = 5; Stmts\}$
$\Rightarrow \{b = 5; S\ Stmts\}$
$\Rightarrow \{b = 5; print(ID);\ Stmts\}$
$\Rightarrow \{b = 5; print(b);\ Stmts\}$
$\Rightarrow \{\boldsymbol{b = 5; print(b);}\}$

유도 연습 3

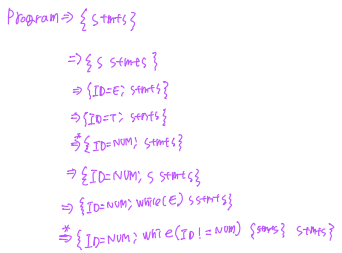
구문분석

$$Program \rightarrow \{\ Stmts\ \}$$
$$Stmts \rightarrow S\ Stmts \mid \varepsilon$$
$$S \rightarrow ID = E\ ;$$
$$S \rightarrow print\ (\ ID\ );$$
$$S \rightarrow while\ (\ E\ )\ S$$
$$S \rightarrow \{\ Stmts\ \}$$
$$E \rightarrow E\ !=\ E \mid T$$
$$T \rightarrow T + F \mid T - F \mid F$$
$$F \rightarrow ID \mid NUM$$
$$ID \rightarrow a \mid b \mid c$$
$$NUM \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

어휘분석

- 위 문법을 사용하여 아래 문장의 유도과정을 보이시오

$$\{a = 1; while(a\ != 9)\ \{print(a); a = a + 1;\}\}$$



$Program$
$\Rightarrow \{Stmts\}$
$\Rightarrow \{S\ Stmts\}$
$\Rightarrow \{ID = E;\ Stmts\}$
$\Rightarrow \{a = E;\ Stmts\}$
$\Rightarrow \{a = T;\ Stmts\}$
$\Rightarrow \{a = F;\ Stmts\}$
$\Rightarrow \{a = NUM;\ Stmts\}$
$\Rightarrow \{a = 1;\ Stmts\}$
$\Rightarrow \{a = 1; S\ Stmts\}$
$\Rightarrow \{a = 1; while(E)\ S\ Stmts\}$
$\Rightarrow \{a = 1; while(E!= E)\ S\ Stmts\}$
$\Rightarrow \{a = 1; while(T!= E)\ S\ Stmts\}$
$\Rightarrow \{a = 1; while(F!= E)\ S\ Stmts\}$
$\Rightarrow \{a = 1; while(ID!= E)\ S\ Stmts\}$
$\Rightarrow \{a = 1; while(a!= E)\ S\ Stmts\}$
$\Rightarrow \{a = 1; while(a!= T)\ S\ Stmts\}$
$\Rightarrow \{a = 1; while(a!= F)\ S\ Stmts\}$
$\Rightarrow \{a = 1; while(a!= NUM)\ S\ Stmts\}$
$\Rightarrow \{a = 1; while(a!= 9)\ S\ Stmts\}$
$\Rightarrow \{a = 1; while(a!= 9)\ \{Stmts\}\ Stmts\}$
$\Rightarrow \{a = 1; while(a!= 9)\ \{S\ Stmts\}\ Stmts\}$
$\Rightarrow \{a = 1; while(a!= 9)\ \{print(ID);\ Stmts\}\ Stmts\}$
$\Rightarrow \{a = 1; while(a!= 9)\ \{print(a);\ Stmts\}\ Stmts\}$
$\Rightarrow \{a = 1; while(a!= 9)\ \{print(a); S\ Stmts\}\ Stmts\}$
$\Rightarrow \{a = 1; while(a!= 9)\ \{print(a); ID = E;\ Stmts\}\ Stmts\}$
$\Rightarrow \{a = 1; while(a!= 9)\ \{print(a); a = E;\ Stmts\}\ Stmts\}$
$\Rightarrow \{a = 1; while(a!= 9)\ \{print(a); a = T;\ Stmts\}\ Stmts\}$
$\Rightarrow \{a = 1; while(a!= 9)\ \{print(a); a = T + F;\ Stmts\}\ Stmts\}$
$\Rightarrow \{a = 1; while(a!= 9)\ \{print(a); a = F + F;\ Stmts\}\ Stmts\}$
$\Rightarrow \{a = 1; while(a!= 9)\ \{print(a); a = ID + F;\ Stmts\}\ Stmts\}$
$\Rightarrow \{a = 1; while(a!= 9)\ \{print(a); a = a + F;\ Stmts\}\ Stmts\}$
$\Rightarrow \{a = 1; while(a!= 9)\ \{print(a); a = a + NUM;\ Stmts\}\ Stmts\}$
$\Rightarrow \{a = 1; while(a!= 9)\ \{print(a); a = a + 1;\ Stmts\}\ Stmts\}$
$\Rightarrow \{a = 1; while(a!= 9)\ \{print(a); a = a + 1;\}\ Stmts\}$
$\Rightarrow \{a = 1; while(a!= 9)\ \{print(a); a = a + 1;\}\}$

예제 `while(b!=3) print(b)`

어휘분석
⇓
구문분석 — 토큰 나열을 입력 받음 / 파싱 해서 AST를 생성한다.
⇓
의미분석
⇓
중간 표현 생성
⇓
코드생성

파싱을 함 (유도 트리를 찾는 것)
⇒ 유도 트리 (파스트리)
파스트리 → AST

유도 과정

Program ⇒ { Stmts }

⇒ { S Stmts }

⇒ { while(E) S Stmts }

⇒ { while(E != E) S Stmts }

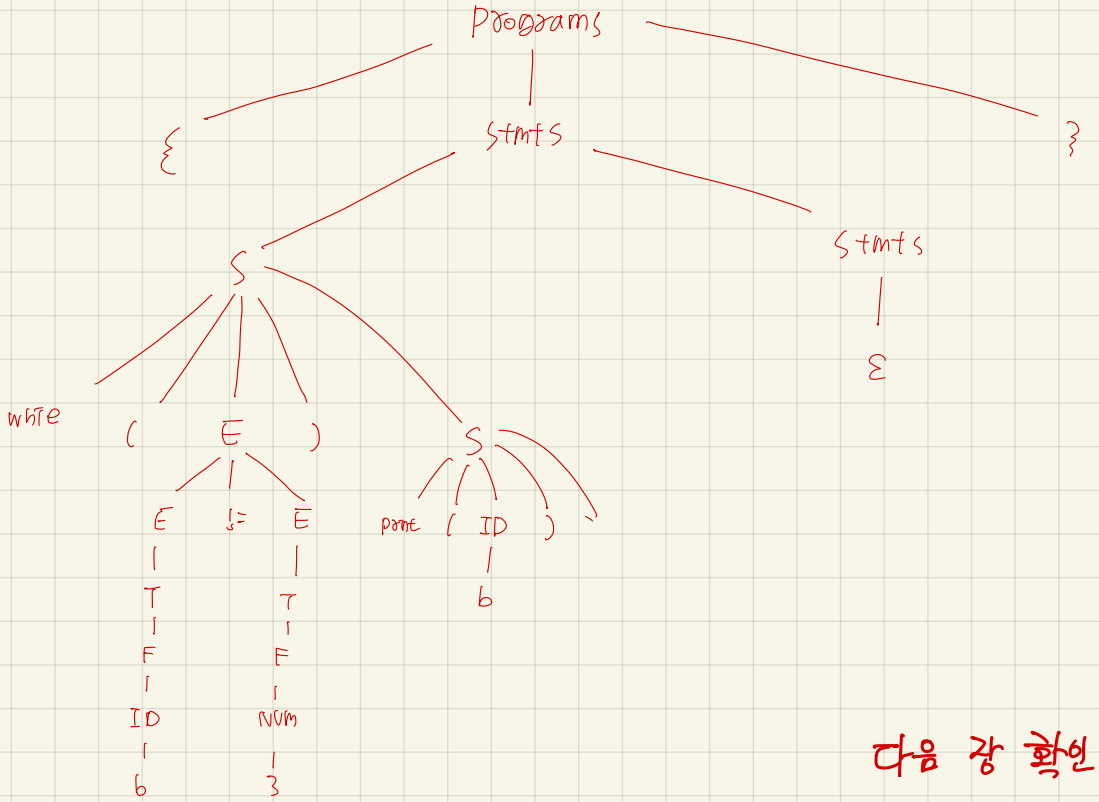
⇒ { while(T != E) S Stmts }

⇒* { while(b != 3) S Stmts }

⇒ { while(b != 3) print(ID); Stmts }'

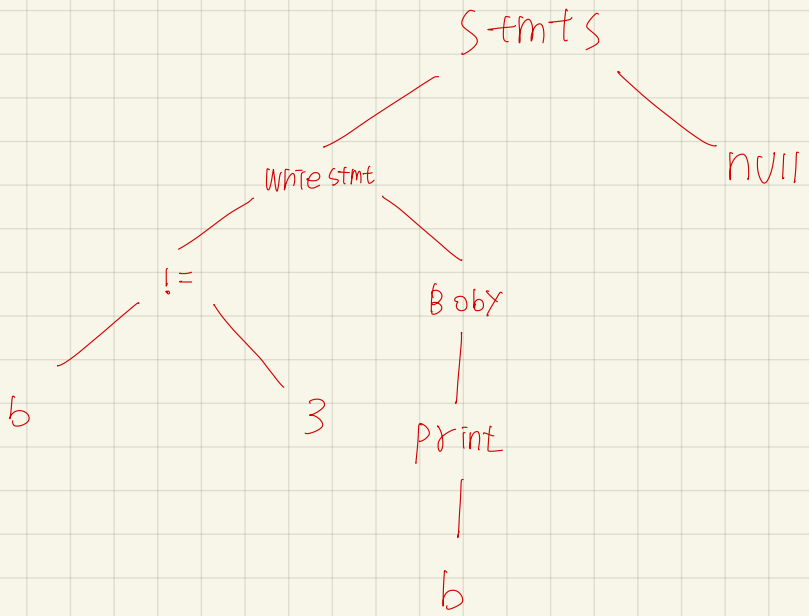
⇒* { while(b != 3) Print(b); }

Parse Tree 구현



다음 강 확인 ㅎㅎ

위 Parse Tree를 AST로 바꾸기



시험

해당코드에서 리터럴을 모두 찾으시오

해당코드에서 연산자를 모두 찾으시오

유도 연습 4

Program → Subpgms
Subpgms → Subpgms Subpgm | Subpgm
Subpgm → def Id ( ) { Stmts } | def Id (Params) { Stmts }
Params → Params , Id | Id
Stmts → Stmts S | ε
S → Id = E ; | read (Id) ; | write (E) ; | return ; | return E ; | E ; | ;
S → if (E) S | if (E) S else S
S → while (E) S
S → { Stmts }
E → F | E < F | E <= F | E > F | E >= F
F → G | F + G | F − G
G → H | G * H | G / H | G % H
H → Id | Num | Str | (E) | Id ( ) | Id (Args)
Args → Args , E | E

- 위 문법을 사용하여 아래 문장의 유도과정을 보이시오

추후 어휘분석기를 구현할 때 알아야할 것

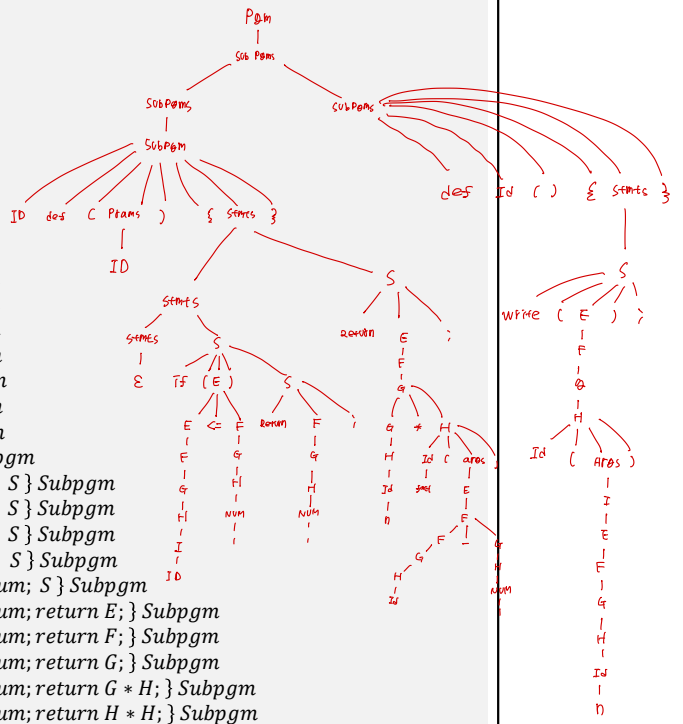
(ID, fact) — 이렇게 해야 ID가 여러개가 올 수 있다
(LPAREN, ( ) 그리고 LPAREN의 경우 (형태로 여러개가 올 수 있다
그래서 ((,(로 표현 가능하다.

```
def fact(n){
   if(n<=1) return 1;
   return n*fact(n-1);
}
def main() {
   read(n);
   write("Result=");
   write(fact(n));
}
```

해당 코드를 유도 할 때 유도가 되면 오픈 X

유도가 안된다면 오픈 O

Program
⇒ *Subpgms*
⇒ *Subpgms Subpgm*
⇒ *Subpgm Subpgm*
⇒ *def Id (Params) { Stmts } Subpgm*
⇒ *def Id (Id) { Stmts } Subpgm*
⇒ *def Id (Id) { Stmts S } Subpgm*
⇒ *def Id (Id) { Stmts S S } Subpgm*
⇒ *def Id (Id) { S S } Subpgm*
⇒ *def Id (Id) { if (E) S S } Subpgm*
⇒ *def Id (Id) { if (E <= F) S S } Subpgm*
⇒ *def Id (Id) { if (F <= F) S S } Subpgm*
⇒ *def Id (Id) { if (G <= F) S S } Subpgm*
⇒ *def Id (Id) { if (H <= F) S S } Subpgm*
⇒ *def Id (Id) { if (Id <= F) S S } Subpgm*
⇒ *def Id (Id) { if (Id <= G) S S } Subpgm*
⇒ *def Id (Id) { if (Id <= H) S S } Subpgm*
⇒ *def Id (Id) { if (Id <= Num) S S } Subpgm*
⇒ *def Id (Id) { if (Id <= Num) return E; S } Subpgm*
⇒ *def Id (Id) { if (Id <= Num) return F; S } Subpgm*
⇒ *def Id (Id) { if (Id <= Num) return G; S } Subpgm*
⇒ *def Id (Id) { if (Id <= Num) return H; S } Subpgm*
⇒ *def Id (Id) { if (Id <= Num) return Num; S } Subpgm*
⇒ *def Id (Id) { if (Id <= Num) return Num; return E; } Subpgm*
⇒ *def Id (Id) { if (Id <= Num) return Num; return F; } Subpgm*
⇒ *def Id (Id) { if (Id <= Num) return Num; return G; } Subpgm*
⇒ *def Id (Id) { if (Id <= Num) return Num; return G * H; } Subpgm*
⇒ *def Id (Id) { if (Id <= Num) return Num; return H * H; } Subpgm*
⇒ *def Id (Id) { if (Id <= Num) return Num; return Id * H; } Subpgm*
⇒ *def Id (Id) { if (Id <= Num) return Num; return Id * Id(Args); } Subpgm*
⇒ *def Id (Id) { if (Id <= Num) return Num; return Id * Id(E); } Subpgm*
⇒ *def Id (Id) { if (Id <= Num) return Num; return Id * Id(F); } Subpgm*
⇒ *def Id (Id) { if (Id <= Num) return Num; return Id * Id(F − G); } Subpgm*
⇒ *def Id (Id) { if (Id <= Num) return Num; return Id * Id(G − G); } Subpgm*
⇒ *def Id (Id) { if (Id <= Num) return Num; return Id * Id(H − G); } Subpgm*
⇒ *def Id (Id) { if (Id <= Num) return Num; return Id * Id(Id − G); } Subpgm*
⇒ *def Id (Id) { if (Id <= Num) return Num; return Id * Id(Id − H); } Subpgm*
⇒ *def Id (Id) { if (Id <= Num) return Num; return Id * Id(Id − Num); } Subpgm*
⇒ *def Id (Id) { if (Id <= Num) return Num; return Id * Id(Id − Num); } def Id( ) { Stmts }*
…



Leftmost derivation, rightmost derivation

Leftmost derivation (좌단유도)
- 유도의 각 단계에서 sentential form 내 leftmost non-terminal symbol(가장 왼쪽 논터미털 심볼)을 교체하여 얻어진 derivation을 leftmost derivation이라 함

Rightmost derivation (우단유도)
- 유도의 각 단계에서 sentential form 내 rightmost non-terminal symbol(가장 오른쪽 논터미털 심볼)을 교체하여 얻어진 derivation을 rightmost derivation이라 함

$E \rightarrow E + T \mid E - T \mid T$
$T \rightarrow T * F \mid T \,/\, F \mid F$
$F \rightarrow (E) \mid id$

$id + (id * id)$의 leftmost derivation	$id + (id * id)$의 rightmost derivation
E	E
$\Rightarrow E + T$	$\Rightarrow E + T$
$\Rightarrow T + T$	$\Rightarrow E + F$
$\Rightarrow F + T$	$\Rightarrow E + (E)$
$\Rightarrow id + T$	$\Rightarrow E + (T)$
$\Rightarrow id + F$	$\Rightarrow E + (T * F)$
$\Rightarrow id + (E)$	$\Rightarrow E + (T * id)$
$\Rightarrow id + (T)$	$\Rightarrow E + (F * id)$
$\Rightarrow id + (T * F)$	$\Rightarrow E + (id * id)$
$\Rightarrow id + (F * F)$	$\Rightarrow T + (id * id)$
$\Rightarrow id + (id * F)$	$\Rightarrow F + (id * id)$
$\Rightarrow id + (id * id)$	$\Rightarrow id + (id * id)$

유도의 트리 표현 1

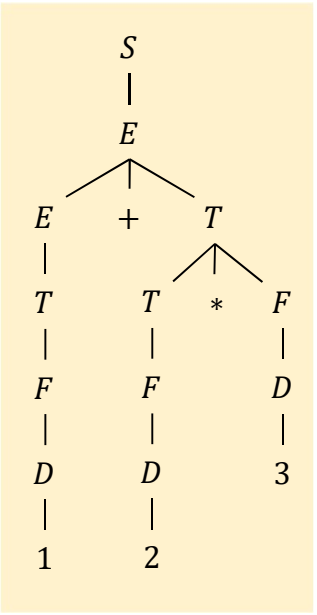
문법 $G = (V, T, S, P)$
$V = \{S, E, T, F, D\}$,
$T = \{+, *, (,), 0,1,2,3,4,5,6,7,8,9\}$

문법 G의 생성규칙집합 P

$S \rightarrow E$
$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow D$
$D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

- 문법 G로부터 $\mathbf{1 + 2 * 3}$의 유도 과정과 유도 트리

S
$\Rightarrow E$
$\Rightarrow E + T$
$\Rightarrow T + T$
$\Rightarrow F + T$
$\Rightarrow D + T$
$\Rightarrow 1 + T$
$\Rightarrow 1 + T * F$
$\Rightarrow 1 + F * F$
$\Rightarrow 1 + D * F$
$\Rightarrow 1 + 2 * F$
$\Rightarrow 1 + 2 * D$
$\Rightarrow 1 + 2 * 3$

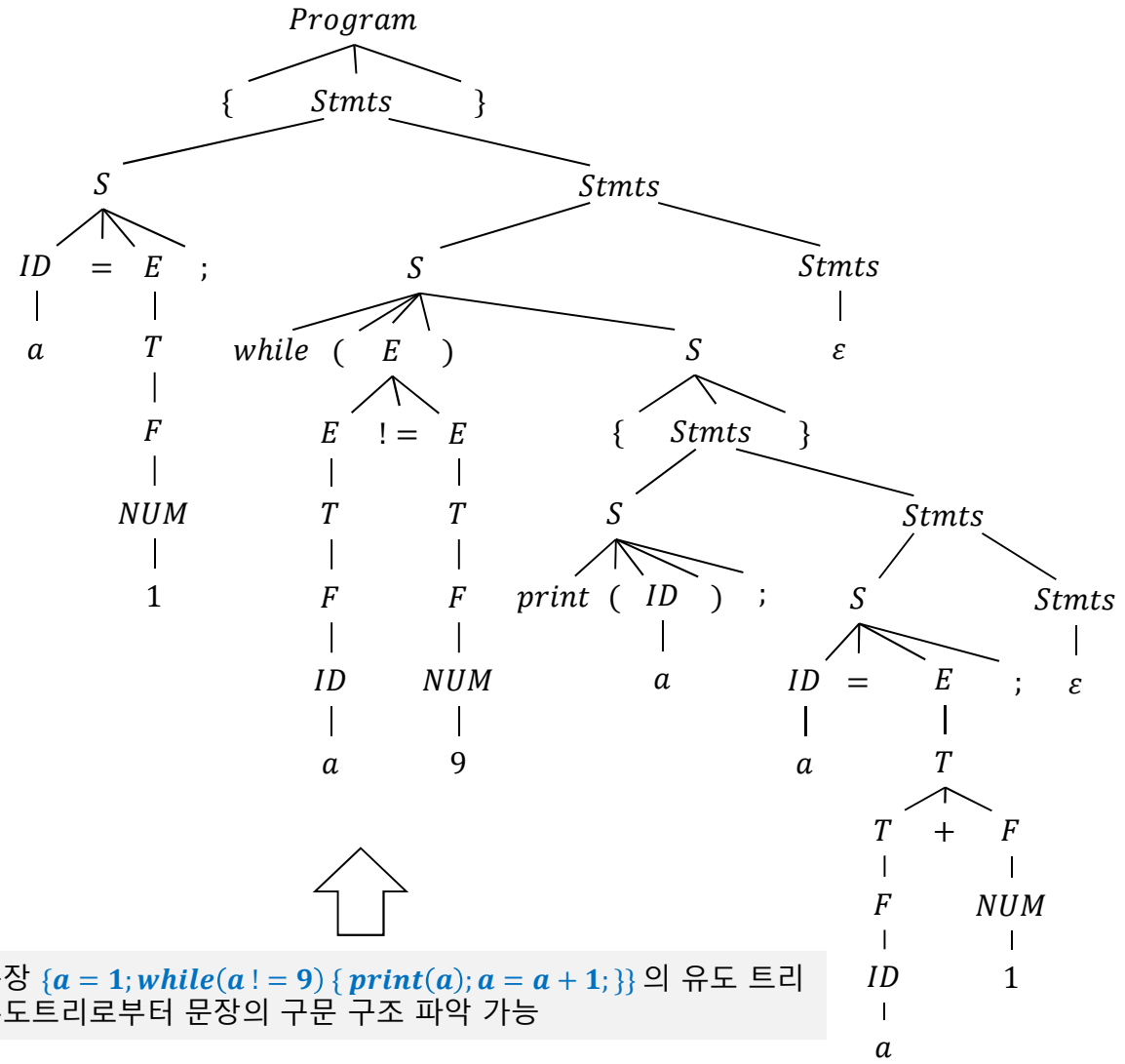


- 유도트리로부터 문장의 구문 구조 파악 가능
- $\mathbf{(1 + 2) * 3}$ 가 아니라 $\mathbf{1 + (2 * 3)}$임

유도의 트리 표현 2

$Program \rightarrow \{\ Stmts\ \}$
$Stmts \rightarrow S\ Stmts \mid \varepsilon$
$S \rightarrow ID = E\ ;$
$S \rightarrow print\ (\ ID\ );$
$S \rightarrow while\ (\ E\ )\ S$
$S \rightarrow \{\ Stmts\ \}$
$E \rightarrow E\ != E \mid T$
$T \rightarrow T + F \mid T - F \mid F$
$F \rightarrow ID \mid NUM$
$ID \rightarrow a \mid b \mid c$
$NUM \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$Program$
$\Rightarrow \{Stmts\}$
$\Rightarrow \{S\ Stmts\}$
$\Rightarrow \{ID = E;\ Stmts\}$
$\Rightarrow \{a = E;\ Stmts\}$
$\Rightarrow \{a = T;\ Stmts\}$
$\Rightarrow \{a = F;\ Stmts\}$
$\Rightarrow \{a = NUM;\ Stmts\}$
$\Rightarrow \{a = 1;\ Stmts\}$
$\Rightarrow \{a = 1; S\ Stmts\}$
$\Rightarrow \{a = 1; while(E)\ S\ Stmts\}$
$\Rightarrow \{a = 1; while(E != E)\ S\ Stmts\}$
$\Rightarrow \{a = 1; while(T != E)\ S\ Stmts\}$
$\Rightarrow \{a = 1; while(F != E)\ S\ Stmts\}$
$\Rightarrow \{a = 1; while(ID != E)\ S\ Stmts\}$
$\Rightarrow \{a = 1; while(a != E)\ S\ Stmts\}$
$\Rightarrow \{a = 1; while(a != T)\ S\ Stmts\}$
$\Rightarrow \{a = 1; while(a != F)\ S\ Stmts\}$
$\Rightarrow \{a = 1; while(a != NUM)\ S\ Stmts\}$
$\Rightarrow \{a = 1; while(a != 9)\ S\ Stmts\}$
$\Rightarrow \{a = 1; while(a != 9)\ \{Stmts\}\ Stmts\}$
$\Rightarrow \{a = 1; while(a != 9)\ \{S\ Stmts\}\ Stmts\}$
$\Rightarrow \{a = 1; while(a != 9)\ \{print(ID);\ Stmts\}\ Stmts\}$
$\Rightarrow \{a = 1; while(a != 9)\ \{print(a);\ Stmts\}\ Stmts\}$
$\Rightarrow \{a = 1; while(a != 9)\ \{print(a); S\ Stmts\}\ Stmts\}$
$\Rightarrow \{a = 1; while(a != 9)\ \{print(a); ID = E;\ Stmts\}\ Stmts\}$
$\Rightarrow \{a = 1; while(a != 9)\ \{print(a); a = E;\ Stmts\}\ Stmts\}$
$\Rightarrow \{a = 1; while(a != 9)\ \{print(a); a = T;\ Stmts\}\ Stmts\}$
$\Rightarrow \{a = 1; while(a != 9)\ \{print(a); a = T + F;\ Stmts\}\ Stmts\}$
$\Rightarrow \{a = 1; while(a != 9)\ \{print(a); a = F + F;\ Stmts\}\ Stmts\}$
$\Rightarrow \{a = 1; while(a != 9)\ \{print(a); a = ID + F;\ Stmts\}\ Stmts\}$
$\Rightarrow \{a = 1; while(a != 9)\ \{print(a); a = a + F;\ Stmts\}\ Stmts\}$
$\Rightarrow \{a = 1; while(a != 9)\ \{print(a); a = a + NUM;\ Stmts\}\ Stmts\}$
$\Rightarrow \{a = 1; while(a != 9)\ \{print(a); a = a + 1;\ Stmts\}\ Stmts\}$
$\Rightarrow \{a = 1; while(a != 9)\ \{print(a); a = a + 1;\}\ Stmts\}$
$\Rightarrow \{a = 1; while(a != 9)\ \{print(a); a = a + 1;\}\}$



- 문장 $\{a = 1; while(a\ != 9)\ \{\ print(a); a = a + 1;\}\}$ 의 유도 트리
- 유도트리로부터 문장의 구문 구조 파악 가능

Parsing

Parsing(파싱)

- 문법 G와 스트링 w가 주어질 때, G로부터 w가 유도될 때 사용된 생성규칙들의 나열을 찾는 과정을 파싱(parsing)이라 함

  문법 $S \rightarrow SS \mid aSb \mid bSa \mid \lambda$에 대해 스트링 $w = aabb$를 파싱
  - $S \Rightarrow aSb \Rightarrow aa\boldsymbol{S}bb \Rightarrow \boldsymbol{aabb}$

- 소스프로그램이 언어의 문법에 맞게 작성된 것인지 확인하는 것으로 파싱 방법에는 top-down parsing(하향식 파싱)과 bottom-up parsing(상향식 파싱)이 있음

Top-down parsing(하향식 파싱)

- 시작 심볼에서 시작하여 생성 규칙을 적용하여 입력 스트링을 유도하는 방법
- 시작 심볼에서부터 하향식 트리 확장을 통해 parse tree를 찾는 방법
- 입력 스트링의 좌단유도를 찾는 과정으로 볼 수 있음
- Recursive descent parsing(재귀하강파싱)과 LL parsing(LL 파싱) 방법이 있음
- GCC, V8 등 recursive descent parser로 구현됨

Bottom-up parsing(상향식 파싱)

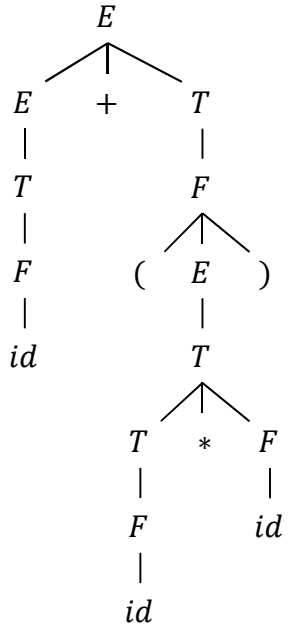
- 입력 스트링에서 시작하여 생성 규칙을 반대 방향으로 적용하여 시작 심볼을 얻어내는 방법
- Shift-reduce parsing 방법이 있음

Parse tree

Parse tree (파스 트리, derivation tree, 유도 트리, concrete syntax tree)
- Derivation을 ordered rooted tree로 보인 것

$E \to E + T \mid E - T \mid T$
$T \to T * F \mid T / F \mid F$
$F \to (E) \mid id$

$id + (id * id)$의 leftmost derivation	$id + (id * id)$의 rightmost derivation
E	E
$\Rightarrow E + T$	$\Rightarrow E + T$
$\Rightarrow T + T$	$\Rightarrow E + F$
$\Rightarrow F + T$	$\Rightarrow E + (E)$
$\Rightarrow id + T$	$\Rightarrow E + (T)$
$\Rightarrow id + F$	$\Rightarrow E + (T * F)$
$\Rightarrow id + (E)$	$\Rightarrow E + (T * id)$
$\Rightarrow id + (T)$	$\Rightarrow E + (F * id)$
$\Rightarrow id + (T * F)$	$\Rightarrow E + (id * id)$
$\Rightarrow id + (F * F)$	$\Rightarrow T + (id * id)$
$\Rightarrow id + (id * F)$	$\Rightarrow F + (id * id)$
$\Rightarrow id + (id * id)$	$\Rightarrow id + (id * id)$



- $id + (id * id)$에 대한 leftmost derivation의 파스 트리와 rightmost derivation의 파스 트리가 동일함

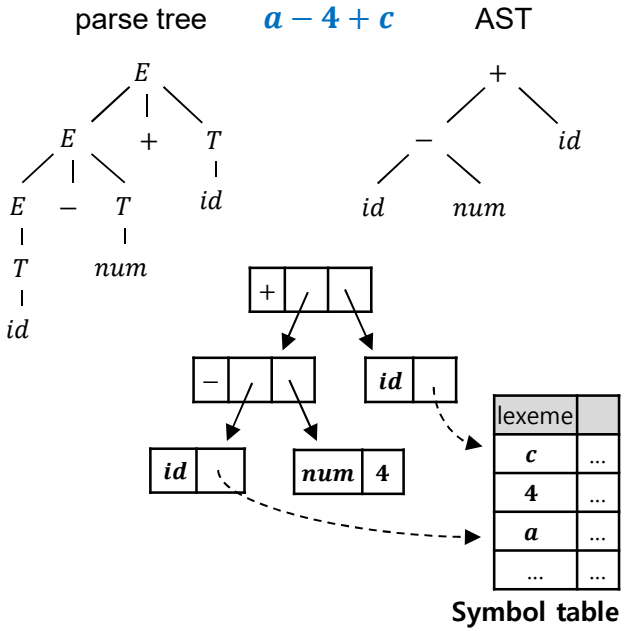
Abstract syntax tree (syntax tree)

Abstract syntax tree(AST, syntax tree, 구문트리, 추상구문트리)

- (문법으로부터의 유도를 표현하는 parse tree와 달리) 프로그램의 필수 구조를 트리로 표현한 것 → parse tree와 달리 AST에서는 파싱 이후 단계에서 불필요한 요소들(예: 괄호, 세미콜론 등)은 표현되지 않음 → abstract syntax tree와의 구별을 강조하기 위해 parse tree를 concrete syntax tree라고도 부름
- 파싱이 수행되는 동안 AST 노드(node)들이 만들어지며 노드에 속성(attribute)값들이 추가됨
- AST의 노드는 하나의 programming construct(프로그램 구성 요소)에 대응되며 construct의 의미를 구성하는 노드들을 자식 노드로 갖는다

No	Production	Semantic rule
1	$E \rightarrow E_1 + T$	$E.node = new\ Node(" + ", E_1.node, T.node)$
2	$E \rightarrow E_1 - T$	$E.node = new\ Node(" - ", E_1.node, T.node)$
3	$E \rightarrow T$	$E.node = T.node$
4	$T \rightarrow (E)$	$T.node = E.node$
5	$T \rightarrow \boldsymbol{id}$	$T.node = new\ Leaf("\boldsymbol{id}", \boldsymbol{id}.entry)$
6	$T \rightarrow \boldsymbol{num}$	$T.node = new\ Leaf("\boldsymbol{num}", \boldsymbol{num}.val)$

- $new\ Leaf(op, val)$ → 2개 필드를 갖는 단말 노드 생성. op는 노드 레이블. val은 노드의 lexical value
- $new\ Node(op, c_1, c_2, \ldots, c_k)$ → k+1개 필드를 갖는 내부 노드 생성. op는 노드 레이블. c_i는 i번째 자식 노드
- $\boldsymbol{id}.entry$ points to the symbol table

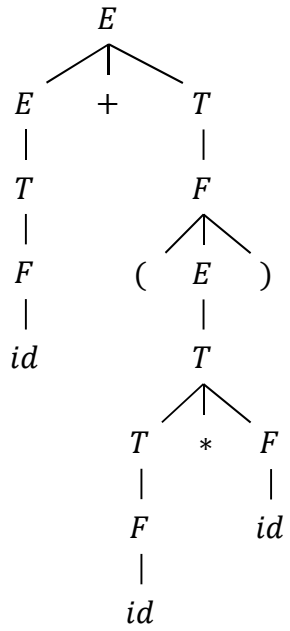


Parse tree를 후위순회(postorder traversal)하면서 AST 생성 시, 생성 규칙 적용 순서 → 5, 6, 2, 5, 1

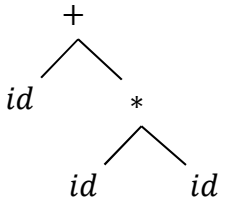
Parse tree vs. syntax tree

$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T * F \mid T / F \mid F$$
$$F \rightarrow (E) \mid id$$

$$\boldsymbol{id} + (\boldsymbol{id} * \boldsymbol{id})$$



parse tree

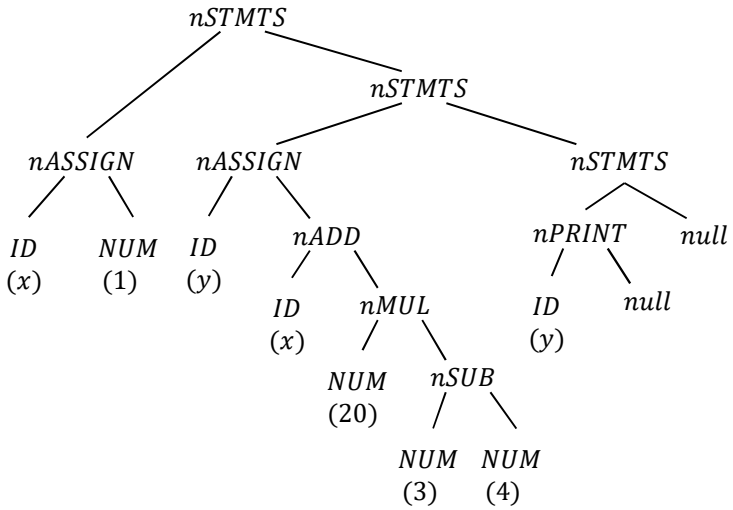


AST

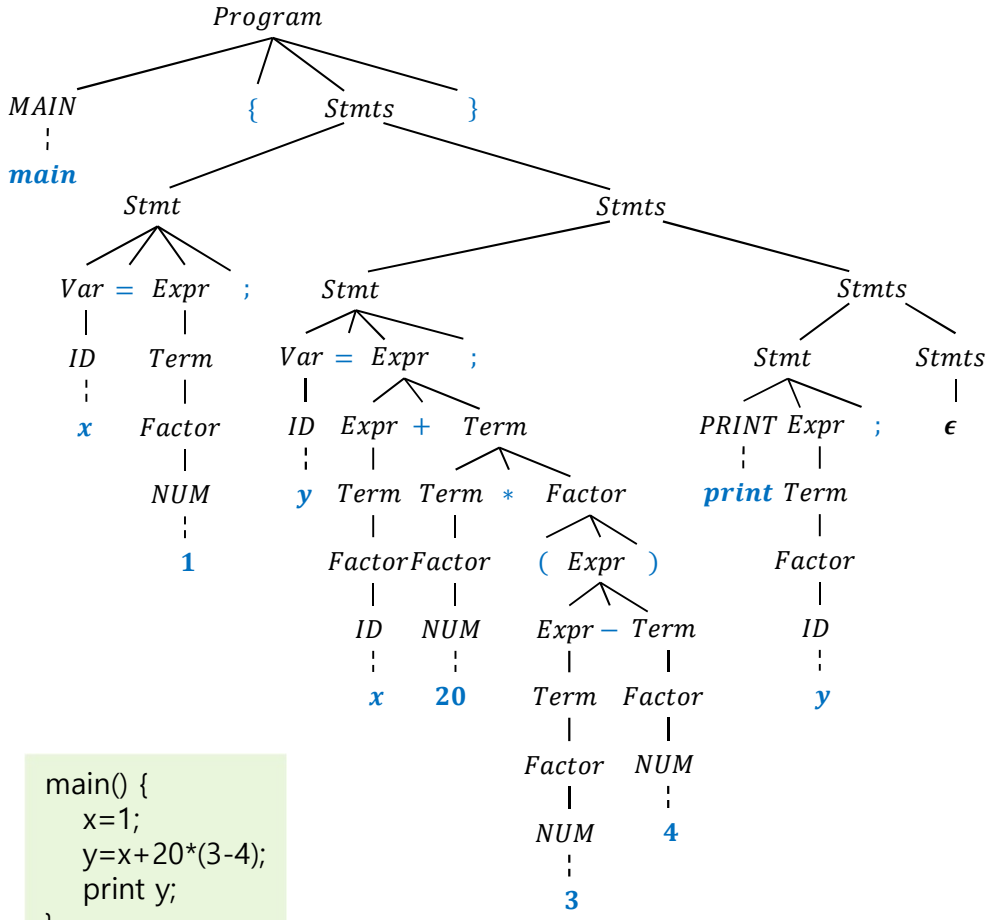
Parse tree vs. syntax tree

$Program \rightarrow \textbf{main}\ (\ )\ \{\ Stmts\ \}$
$Stmts \rightarrow Stmt\ \ Stmts\ \mid \epsilon$
$Stmt \rightarrow ID = Expr\ ;$
$Stmt \rightarrow \textbf{print}\ Expr\ ;$
$Expr \rightarrow Expr + Term \mid Term$
$Term \rightarrow Term * Factor \mid Factor$
$Factor \rightarrow (\ Expr\ ) \mid \textbf{NUM} \mid \textbf{ID}$

ID	$[a-z]$
NUM	$[0-9]+(\.[0-9]*)?$



AST



parse tree

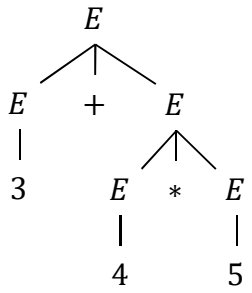
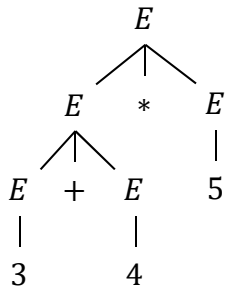
```
main() {
    x=1;
    y=x+20*(3-4);
    print y;
}
```

Ambiguous grammar (1/2)

Ambiguous grammar

- 하나의 문장에 대해 2개 이상의 서로 다른 파스 트리를 생성하는 문법을 ambiguous grammar라고 한다

$E \rightarrow E + E \mid E - E \mid E * E \mid E \,/\, E \mid (E) \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$3 + 4 * 5$의 leftmost derivation, parse tree, 의미	
E $\Rightarrow E + E$ $\Rightarrow 3 + E$ $\Rightarrow 3 + E * E$ $\Rightarrow 3 + 4 * E$ $\Rightarrow 3 + 4 * 5$	E $\Rightarrow E * E$ $\Rightarrow E + E * E$ $\Rightarrow 3 + E * E$ $\Rightarrow 3 + 4 * E$ $\Rightarrow 3 + 4 * 5$
	
의미 → $3 + (4 * 5) = 23$	의미 → $(3 + 4) * 5 = 35$

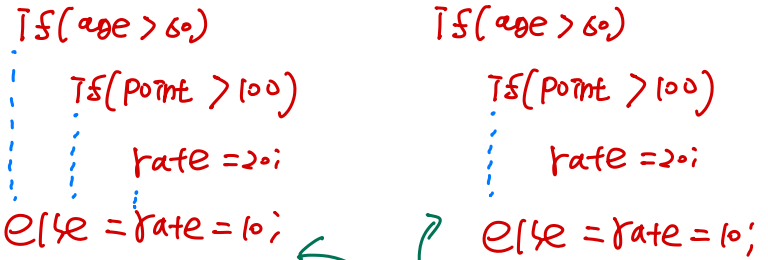
- $3 + 4 * 5$라는 하나의 문장에 대해 서로 다른 2개의 파스 트리가 생성되므로 ambiguous 문법임
- Ambiguous 문법의 경우 하나의 문장에 대해 서로 다른 해석을 갖는 서로 다른 파스 트리들이 생성 가능
- Ambiguous 문법에 대한 구문 분석기는 파스 트리를 결정적으로 구성하기 어려움 → 2가지 해결 방법
- 해결 방법 ① ambiguous 문법을 unambiguous 문법으로 변환하여 사용. 그러나 모든 ambiguous 문법이 unambiguous 문법으로 변환 가능한 것은 아님
- 해결 방법 ② ambiguous 문법으로 얻어진 구문분석기의 파싱표 내 충돌을 제거

Ambiguous grammar (2/2)

Ambiguous grammar

- 하나의 문장에 대해 2개 이상의 서로 다른 파스 트리를 생성하는 문법을 ambiguous grammar라고 한다

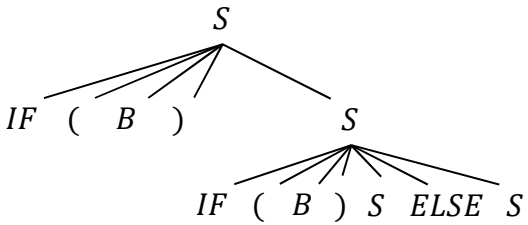
$S \rightarrow IF\ (\ B\ )\ S$
$S \rightarrow IF\ (\ B\ )\ S\ ELSE\ S$
$S \rightarrow id = B\ ;$
$B \rightarrow E > E \mid E$
$E \rightarrow id \mid num$

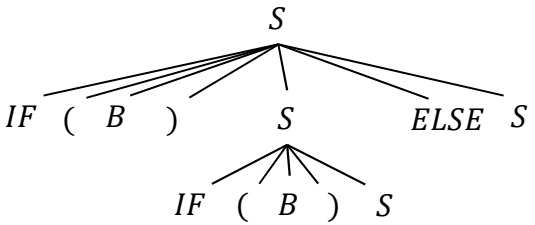


아래 문장에 대해 2개 서로 다른 파스 트리가 생성되므로 위 문법은 ambiguous 문법임

if (age>60) if(point>100) rate=20; else rate=10;

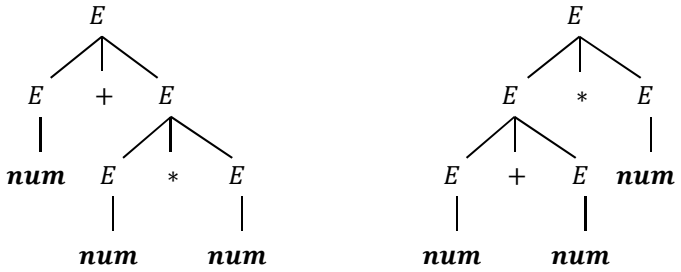
이 문법은 코드는 같지만 if-else 짝을 의미로 해석할 수 없다.





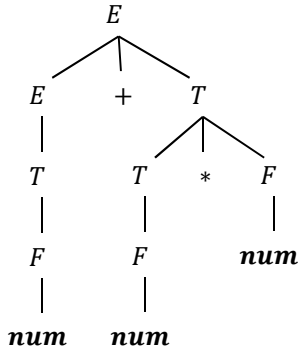
Ambiguous 문법 → unambiguous 문법

$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow (E) \mid id \mid num$$



3 + 4 * 5의 parse tree들
- 왼쪽 parse tree는 3+(4*5)를 의미
- 오른쪽 parse tree는 (3+4)*5를 의미

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id \mid num$$

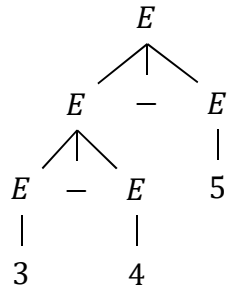
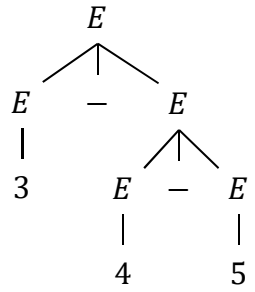


- 하나의 parse tree가 얻어짐

Ambiguous 문법 → unambiguous 문법

Ambiguous grammar

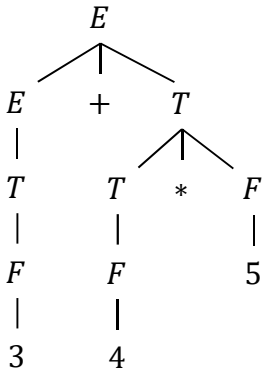
$E \to E + E \mid E - E \mid E * E \mid E / E$
$E \to (E)$
$E \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$3-4-5$의 leftmost derivation, parse tree, 의미	
E $\Rightarrow E - E$ $\Rightarrow E - E - E$ $\Rightarrow 3 - E - E$ $\Rightarrow 3 - 4 - E$ $\Rightarrow 3 - 4 - 5$	E $\Rightarrow E - E$ $\Rightarrow 3 - E$ $\Rightarrow 3 - E - E$ $\Rightarrow 3 - 4 - E$ $\Rightarrow 3 - 4 - 5$
	
$(3-4)-5=-6$	$3-(4-5)=4$

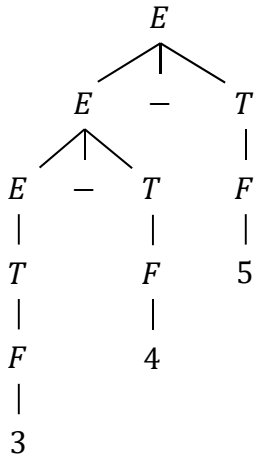
Unambiguous grammar

$E \to E + T \mid E - T \mid T$
$T \to T * F \mid T / F \mid F$
$F \to (E) \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

문장 $3 + 4 * 5$에 대해
유일한 파스트리 생성됨



문장 $3 - 4 - 5$에 대해
유일한 파스트리 생성됨

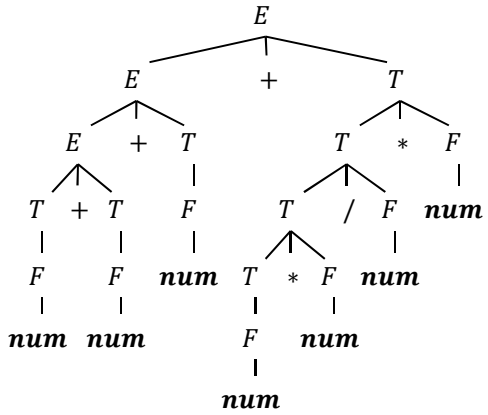


문법 작성: precedence, associativity 반영

우선순위 (precedence)	연산자 (operator)	결합규칙(associativity)
1	*, /	Left to right
2	+, -	Left to right

Operator precedence, associativity
- *, /가 +, -보다 우선순위가 높음
- *, /는 우선순위 동일하며, left associative
- +, -는 우선순위 동일하며, left associative

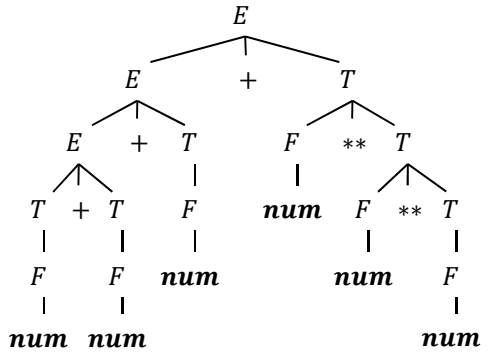
$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T * F \mid T / F \mid F$$
$$F \rightarrow (E) \mid id \mid num$$



$$1 + 2 + 3 + 4 * 5/6 * 7 = \Big(\big((1 + 2) + 3\big) + \Big(\big((4 * 5)/6\big) * 7\Big)\Big)$$

Operator precedence, associativity
- **가 +보다 우선순위가 높음
- **는 right associative
- +는 left associative

$$E \rightarrow E + T \mid T$$
$$T \rightarrow F \mid F ** T$$
$$F \rightarrow (E) \mid id \mid num$$



$$1 + 2 + 3 + 4 ** 2 ** 3 = \Big(\big((1 + 2) + 3\big) + \Big(4^{(2^3)}\Big)\Big)$$

문법에 연산자 우선순위, 결합법칙 반영

연습 1
- 연산자 $+, -, *, /$에 좌측 결합성 적용
- 괄호 내 식을 가장 먼저 계산
- 연산자 $*, /$의 우선순위가 연산자 $+, -$ 보다 높음

$E \rightarrow E + T \mid E - T \mid T$
$T \rightarrow T * F \mid T / F \mid F$
$F \rightarrow (E) \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

연습 2
- 연산자 $+, -, *, /$에 좌측 결합성 적용
- 연산자 ^에 우측 결합성 적용
- 괄호 내 식을 가장 먼저 계산
- 단항연산자 $-$의 우선순위가 거듭제곱연산자 ^ 보다 높음
- 연산자 ^의 우선순위가 연산자 $*, /$보다 높음
- 연산자 $*, /$의 우선순위가 연산자 $+, -$ 보다 높음

$E \rightarrow E + T \mid E - T \mid T$
$T \rightarrow T * F \mid T / F \mid F$
$F \rightarrow P \mid P \hat{} F$
$P \rightarrow -P \mid H$
$H \rightarrow (E) \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

연습 3
- 연산자 $*, -$에 좌측 결합성 적용
- 연산자 $+, /$에 우측 결합성 적용
- 괄호 내 식을 가장 먼저 계산
- 연산자 $+, /$의 우선순위가 연산자 $*, -$보다 높음

$E \rightarrow E * T \mid E - T \mid T$
$T \rightarrow F \mid F + T \mid F / T$
$F \rightarrow (E) \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

시험
좌결합성을 가진 T 항끼리에서
우 결합성을 가지도록
작성해라 ㅎ

C 언어 연산자 우선순위, 결합규칙

우선순위 (precedence)	연산자 (operator)	설명	결합규칙(associativity)
높음 ↕ 낮음	++ --	후위 증가/감소(postfix increment/decrement)	Left to Right
	++ -- + - !	전위 증가/감소(prefix increment/decrement) Unary plus/minus Logical NOT	**Right to Left**
	* / %	Multiplication, division, modulo	Left to Right
	+ -	Addition, subtraction	Left to Right
	< <= > >=	LT, LE, GT, GE	Left to Right
	== !=	EQ, NE	Left to Right
	&&	Logical AND	Left to Right
	||	Logical OR	Left to Right
	=	Assignment	**Right to Left**

3 - 4 - 5 = -6

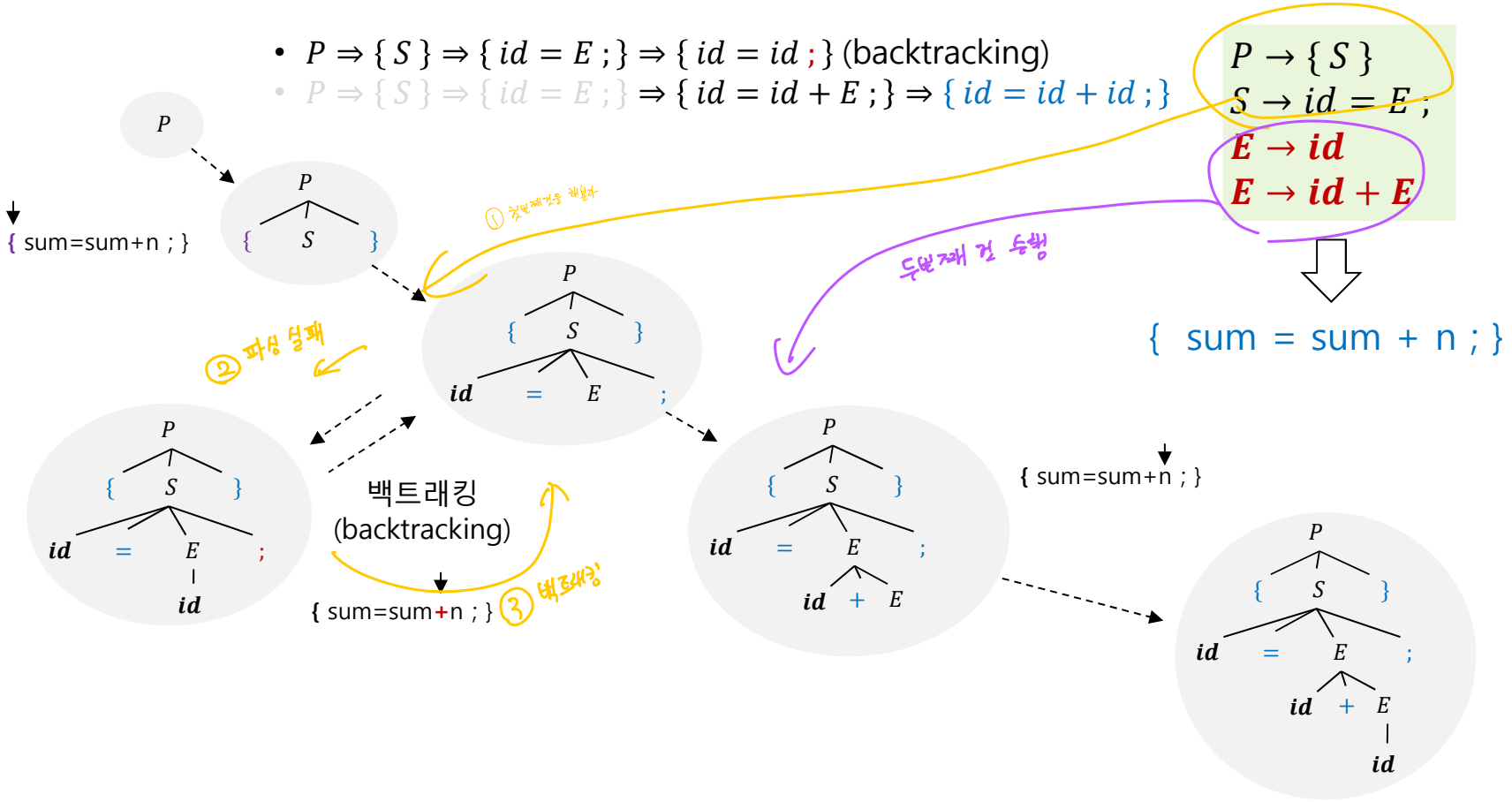
(3 - 4) - 5 = -6

3 - (4 - 5) = 4

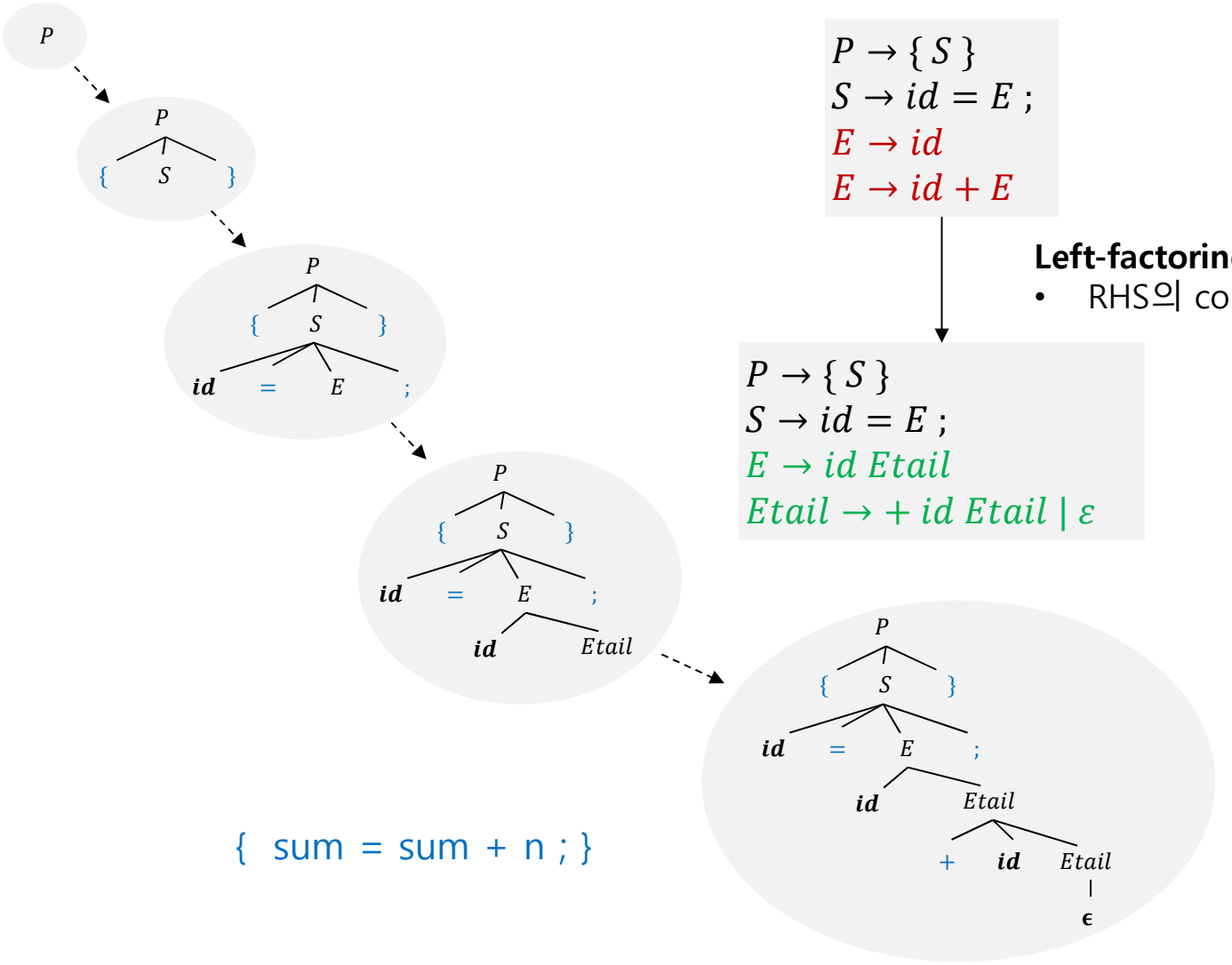
Top-down parsing with backtracking

Top-down parsing(하향식 파싱)

- 시작 심볼에서 시작하여 생성 규칙을 적용하여 입력 스트링을 유도하는 방법 → 유도 과정에서 항상 유일한 규칙을 적용할 수 있다면 백트래킹(backtracking) 없는 효율적 파싱 가능



문법 작성: left factoring



$P \rightarrow \{\ S\ \}$
$S \rightarrow id = E\ ;$
$E \rightarrow id$
$E \rightarrow id + E$

Left-factoring

- RHS의 common prefix 제거

$P \rightarrow \{\ S\ \}$
$S \rightarrow id = E\ ;$
$E \rightarrow id\ Etail$
$Etail \rightarrow +\ id\ Etail \mid \varepsilon$

{ sum = sum + n ; }

Recursive descent parsing

다음 token
식별자 저장

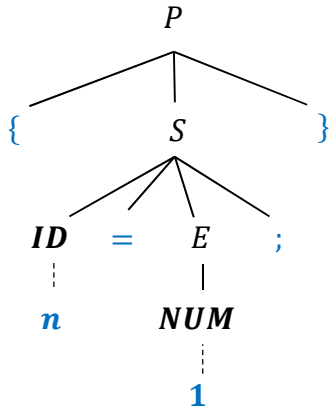
$P \rightarrow \{ S \}$
$S \rightarrow ID = E ;$
$E \rightarrow ID \mid NUM$

```
int token, ID=258, NUM=259;
void match(int t){
    if(token!=t) { printf("error₩n"); exit(1); }
    token=nextToken();
}
void P() {
    match('{');
    S();
    match('}');
}
void S() {
    match(ID);  match('=');  E(); match(';');
}
void E() {
   if (token==ID) match(ID);
   else match(NUM);
}
int main(void){
    token=nextToken();
    P();
    printf("parsing success₩n");
}
```

Recursive descent parsing(재귀하강파싱)

- 파싱 → 소스프로그램이 언어의 문법에 맞게 작성된 것인지 확인하는 것
- Recursive descent parser(재귀하강파서)는 언어 문법의 non-terminal symbol에 대응하는 (recursive) 프로시저들의 모음으로 구현 → 각 non-terminal에 대응하는 하나의 프로시저 작성(RHS 내 terminal은 다음 토큰과 일치(match)하도록 구현, RHS 내 non-terminal은 대응하는 프로시저 호출로 구현)

{ n = 1 ; }

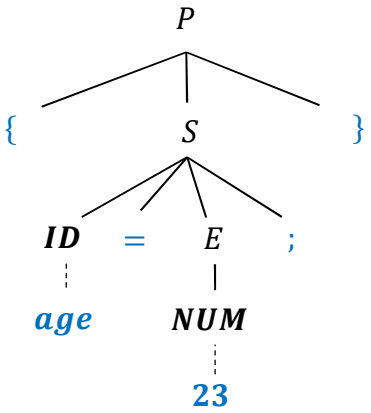



```
P()
    {
    S()
        match(ID)
        match('=')
        E()
                match(NUM)
        match(';')
    }
```

재귀하강파서 구현 연습 1

$P \rightarrow \{ S \}$
$S \rightarrow ID = E ;$
$E \rightarrow ID \mid NUM$

{ age = 23 ; }



컴파일 및 실행

```
flex lex.l
gcc -o lexer.exe lex.yy.c -LC:₩GnuWin32₩lib –lfl
lexer < input1.txt
parsing success
lexer < input2.txt
error: invalid token
lexer < input3.txt
error: eof not found
```

input1.txt

```
{ age = 23; }
```

input2.txt

```
{ age = 23 }
```

input3.txt

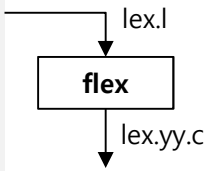
```
{ age = 23; } { v = 1; }
```

lex.l

```
%{
#include <stdio.h>
int token, ID=258, NUM=259;
%}

%%
[0-9]+                  { return NUM; }
[a-zA-Z][a-zA-Z0-9]*    { return ID; }
[{}=;]                  { return yytext[0]; }
[ ₩t₩n]                 {  }
.                       { return yytext[0]; }
%%

void P();   void S();   void E();
int nextToken(){ return yylex(); }
void match(int t){
    if(token!=t) { printf("error: invalid token₩n"); exit(1); }
    token=nextToken();
}
void P() {    match('{'); S(); match('}');    }
void S() {    match(ID); match('='); E(); match(';');    }
void E() {
    if (token==ID) match(ID);
    else match(NUM);
}
int main(void){
    token=nextToken();
    P();
    if(0==nextToken()) printf("parsing success₩n");
    else printf("error: eof not found₩n");
}
```

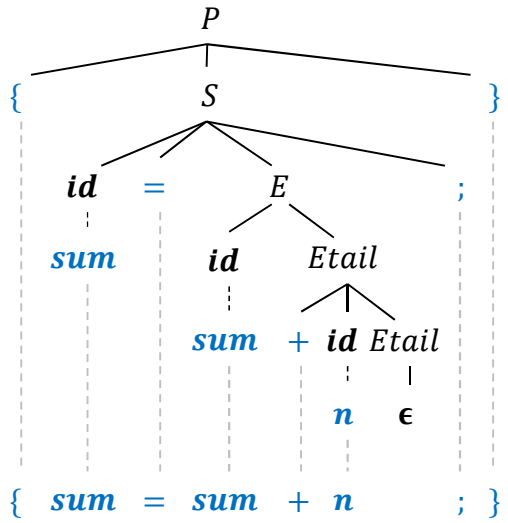


- **yylex()**
- 입력에서 읽은 토큰 반환
- 파일 끝(end-of-file) 도달 시 **0** 반환

재귀하강파서 구현 연습 2

$P \rightarrow \{ S \}$
$S \rightarrow ID = E ;$
$E \rightarrow ID$
$E \rightarrow ID + E$

$P \rightarrow \{ S \}$
$S \rightarrow ID = E ;$
$E \rightarrow ID\ Etail$
$Etail \rightarrow +ID\ Etail \mid \varepsilon$



다음 토큰이 ; 이면
$Etail \rightarrow \varepsilon$ 적용

컴파일 및 실행

```
flex lex.l
gcc -o lexer.exe lex.yy.c -LC:₩GnuWin32₩lib –lfl
lexer < input1.txt
parsing success
lexer < input2.txt
parsing success
```

input1.txt

```
{ sum = sum + n; }
```

input2.txt

```
{ a = b + c + d; }
```

```
%{
#include <stdio.h>
int token, ID=258;
%}

%%
[a-zA-Z][a-zA-Z0-9]*     { return ID; }
[{}=;+]                  { return yytext[0]; }
[ ₩t₩n]                  { }
.                        { return yytext[0]; }
%%

void P();   void S();   void E();   void Etail();
int nextToken(){ return yylex(); }
void match(int t){
    if(token!=t) { printf("error: invalid token₩n"); exit(1); }
    token=nextToken();
}
void P() {   match('{'); S(); match('}');   }
void S() {   match(ID); match('='); E(); match(';');   }
void E() {   match(ID); Etail();   }
void Etail() {
   if (token==';') return;
   match('+'); match(ID); Etail(); // 재귀호출
}
int main(void){
    token=nextToken();
    P();
    if(0==nextToken()) printf("parsing success₩n");
    else printf("error: eof not found₩n");
}
```

문법 작성: left recursion 제거

Left recursive grammar

- Left-recursion에는 direct left-recursion과 indirect left-recursion이 있음
- Left-recursion이 포함된 문법을 left recursive라고 하며 left-recursive 문법은 recursive descent parsing에서 무한 재귀호출을 발생시킴

direct
left-recursion 발생

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id \mid num$$

Left-recursion
제거

$$E \rightarrow T\ E'$$
$$E' \rightarrow +\ T\ E' \mid \varepsilon$$
$$T \rightarrow F\ T'$$
$$T' \rightarrow *\ F\ T' \mid \varepsilon$$
$$F \rightarrow (E) \mid id \mid num$$

recursive descent parser

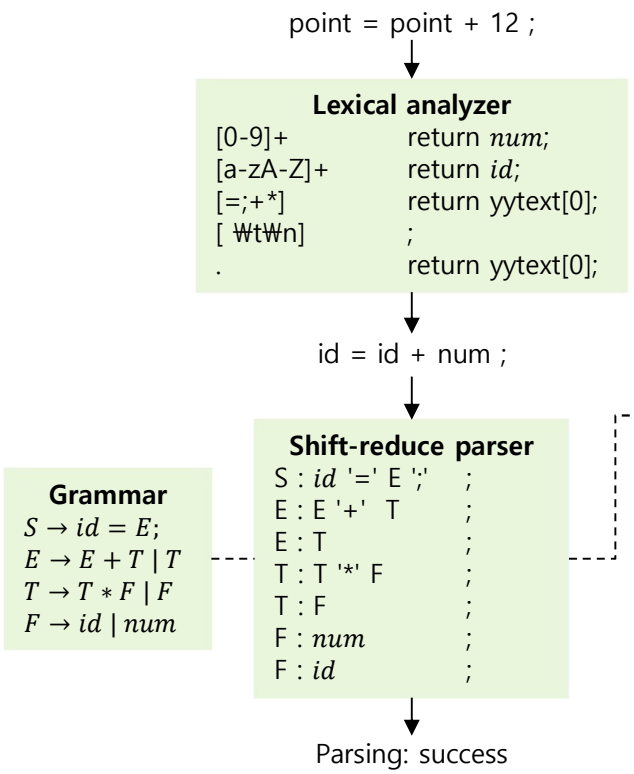
```
…
void E() {
   E();  // 무한 재귀호출
   match('+');
   T();
}
…
```

- Left-recursion이 제거된 문법
- Top-down parsing에 사용 가능

Shift-reduce parsing

Shift-reduce parsing

- Shift-reduce parsing은 shift와 reduce 동작에 기반한 파싱 방법으로, 파스 트리를 상향식(bottom-up)으로 점증적으로(incrementally) 만드는 효과가 있다
- Shift(이동) → 스택의 톱에서 handle 미발견 시, 다음 입력 심볼을 스택으로 옮기는 것
- Reduce(감축) → 스택의 톱에서 발견된 handle을, handle이 RHS에 매치되는 규칙의 LHS 심볼로 대체하는 것
- Accept(수락) → 초기 공백 스택과 입력 문장에서 시작하여, shift와 reduce 동작을 거친 후, 입력을 소진하고 스택의 톱에 문법의 start symbol이 발견되면 입력 문장을 문법에 맞는 문장으로 받아들이는 것



스택	입력	동작	규칙
$	$id = id + num; $	Shift	
$id	= id + num; $	Shift	
$id =	id + num; $	Shift	
$id = id	+num; $	Reduce	$F \rightarrow id$
$id = F	+num; $	Reduce	$T \rightarrow F$
$id = T	+num; $	Reduce	$E \rightarrow T$
$id = E	+num; $	Shift	
$id = E +	num; $	Shift	
$id = E + num	; $	Reduce	$F \rightarrow num$
$id = E + F	; $	Reduce	$T \rightarrow F$
$id = E + T	; $	Reduce	$E \rightarrow E + T$
$id = E	; $	Shift	
$id = E;	$	Reduce	$S \rightarrow id = E;$
$S	$	Accept	

$id = id + num;$의 rightmost derivation

S
$\Rightarrow id = E;$
$\Rightarrow id = E + T;$
$\Rightarrow id = E + F;$
$\Rightarrow id = E + num;$
$\Rightarrow id = T + num;$
$\Rightarrow id = F + num;$
$\Rightarrow id = id + num;$

$id = id + num;$의 rightmost derivation을 역순으로 생성

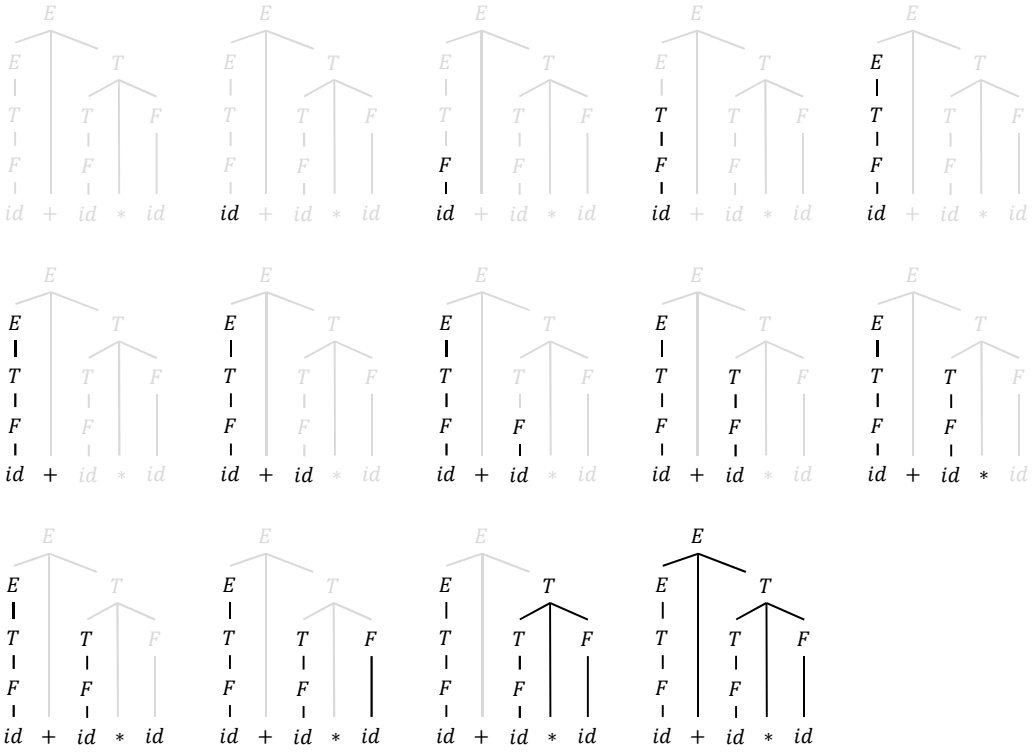
Shift-reduce parsing

Bottom-up parser 의 종류 중 하나임

Shift-reduce parsing

- Shift와 reduce 동작의 연속으로 수행되는 파싱을 shift-reduce parsing이라 함
- Shift(이동) → 스택의 톱에서 handle 미발견 시, 다음 입력 심볼을 스택으로 옮기는 것
- Reduce(감축) → 스택의 톱에서 발견된 handle을, handle이 RHS에 매치되는 규칙의 LHS 심볼로 대체하는 것
- Accept(수락) → 초기 공백 스택과 입력 문장에서 시작하여, shift와 reduce 동작을 거친 후, 입력을 소진하고 스택의 톱에 문법의 start symbol이 발견되면 입력 문장을 문법에 맞는 문장으로 받아들이는 것

밑에서부터 역으로 제약을 봄.



$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id$$

$id + id * id$의 rightmost derivation:

E
$\Rightarrow E + T$
$\Rightarrow E + T * F$
$\Rightarrow E + T * id$
$\Rightarrow E + F * id$
$\Rightarrow E + id * id$
$\Rightarrow T + id * id$
$\Rightarrow F + id * id$
$\Rightarrow id + id * id$

스택	입력	동작	규칙
$	$id + id * id$$	Shift id	
$$id$	$+ id * id$$	Reduce	$F \rightarrow id$
$$F$	$+ id * id$$	Reduce	$T \rightarrow F$
$$T$	$+ id * id$$	Reduce	$E \rightarrow T$
$$E$	$+ id * id$$	Shift $+$	
$$E +$	$id * id$$	Shift id	
$$E + id$	$* id$$	Reduce	$F \rightarrow id$
$$E + F$	$* id$$	Reduce	$T \rightarrow F$
$$E + T$	$* id$$	Shift $*$	
$$E + T *$	$id$$	Shift id	
$$E + T * id$	$	Reduce	$F \rightarrow id$
$$E + T * F$	$	Reduce	$T \rightarrow T * F$
$$E + T$	$	Reduce	$E \rightarrow E + T$
$$E$	$	Accept	

Shift-reduce conflict

if (age>60) if(point>100) rate=20; else rate=10;

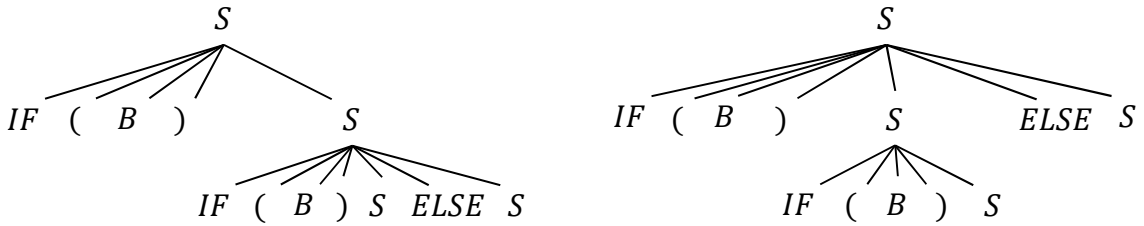
↓

$IF\ (id > num)\ IF(id > num)\ id = num;\ ELSE\ id = num;$

↓

스택	입력	동작	규칙
$	$IF\ (id > num)\ IF(id > num)\ id = num;\ ELSE\ id = num;\ \$$	Shift	
$IF	$(id > num)\ IF(id > num)\ id = num;\ ELSE\ id = num;\ \$$	Shift	
$IF(	$id > num)\ IF(id > num)\ id = num;\ ELSE\ id = num;\ \$$	Shift	
$IF(id	$> num)\ IF(id > num)\ id = num;\ ELSE\ id = num;\ \$$	Reduce	$E \rightarrow id$
$IF(E	$> num)\ IF(id > num)\ id = num;\ ELSE\ id = num;\ \$$	Shift	
$IF(E >	$num)\ IF(id > num)\ id = num;\ ELSE\ id = num;\ \$$	Shift	
$IF(E > num	$)\ IF(id > num)\ id = num;\ ELSE\ id = num;\ \$$	Reduce	$E \rightarrow num$
$IF(E > E	$)\ IF(id > num)\ id = num;\ ELSE\ id = num;\ \$$	Reduce	$B \rightarrow E > E$
$IF(B	$)\ IF(id > num)\ id = num;\ ELSE\ id = num;\ \$$	Shift	
...		...	...
$IF(B)\ IF(B)\ S	$ELSE\ id = num;\ \$$	Shift or Reduce	

$S \rightarrow IF\ (\ B\ )\ S$
$S \rightarrow IF\ (\ B\ )\ S\ ELSE\ S$
$S \rightarrow id = B\ ;$
$B \rightarrow E > E \mid E$
$E \rightarrow id \mid num$



Reduce-reduce conflict

References

- 박두순. (2016). (내공 있는 프로그래머로 길러주는)컴파일러의 이해. 한빛아카데미.
- 창병모. (2021). 프로그래밍 언어론 : 원리와 실제. 인피니티북스.
- Aho, A., Lam, M., Sethi, R., Ullman, J. (2006). Compilers: Principles, Techniques, and Tools (2nd Ed.). Addison Wesley.
- Nystrom, R. Crafting interpreter. https://craftinginterpreters.com/
- Sebesta, R. (2012). Concepts of Programming Languages (10th. ed.). Pearson.
- Thain, D. (2023). Introduction to Compilers and Language Design.
- Paxson, V. (1995). Flex, version 2.5.
- Donnelly, C., Stallman, R. (2008). Bison.
- LexAndYacc.pdf (epaperpress.com)
- Tom Niemann, LEX & YACC. http://epaperpress.com/lexandyacc