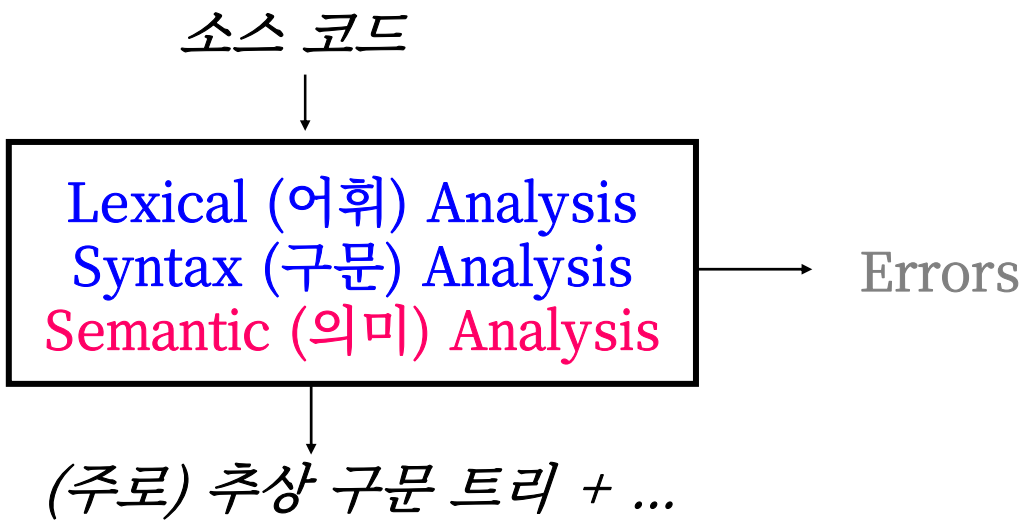
Compiler (컴파일러)
Semantic Analysis I
- Syntax Directed Definitions

2007년 2학기
충남대학교 전기정보통신공학부
조은선

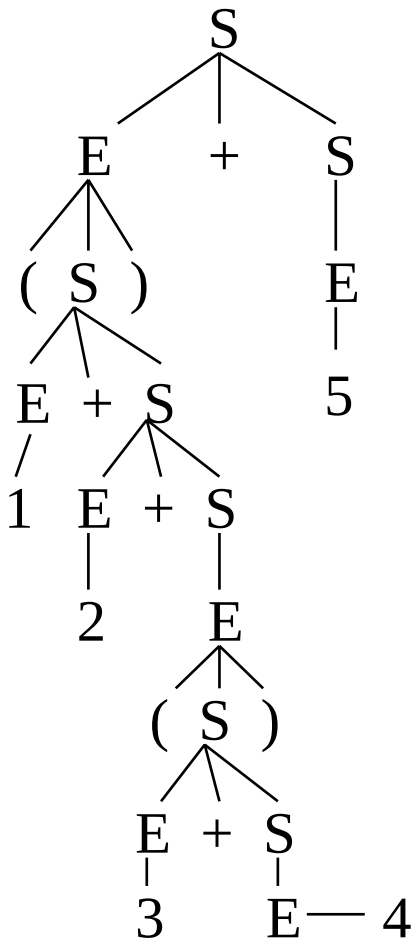
Semantic Analysis

- Semantic Analysis = Syntax Analysis + α



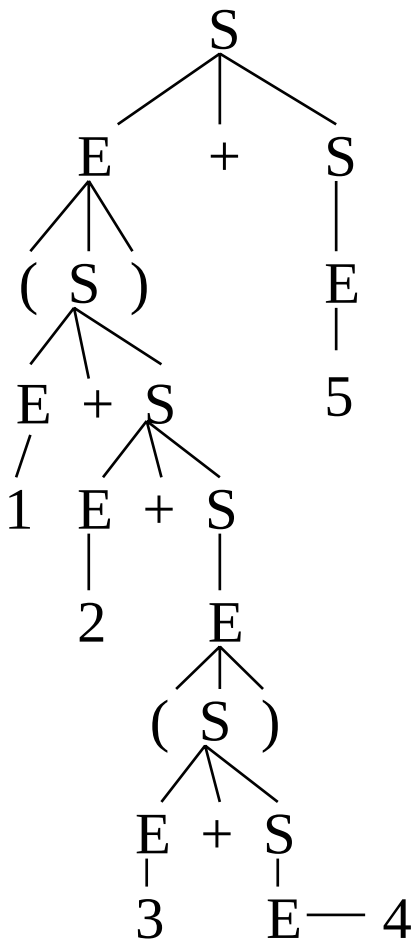
- 먼저 두 가지 도구
  - Abstract Syntax Tree (AST)
  - Syntax-Directed Definition/Translation (SDT)

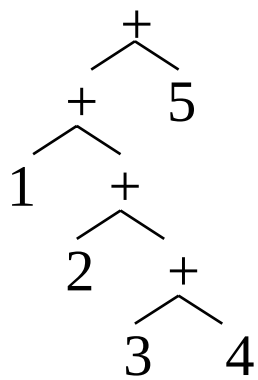
Parse Tree



- 파스트리 (parse tree)
  - 유도과정을 트리로 나타냄
  - terminal 이 leaf 노드
  - non-terminal이 중간노드
  - 유도 순서(left/right) 는 표현 안됨

Abstract Syntax Tree (AST)

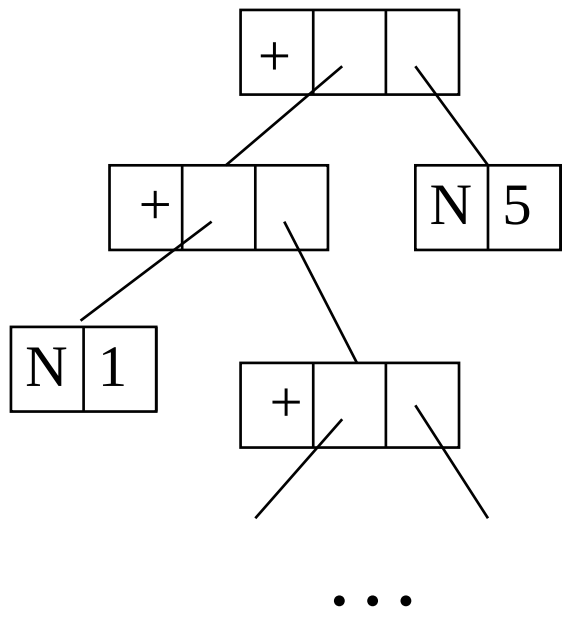




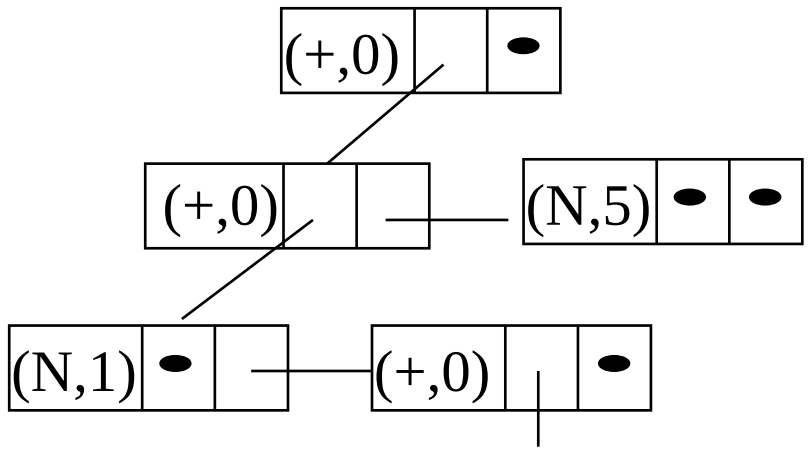
- AST
  - 파스트리에서 불필요한 정보를 제거한 형태

AST 자료구조 – Java 예

```
Abstract class Expr{}
class Add extends Expr {
    Expr left, right;
    Add(Expr L, Expr R) {
       left=L; right=R;
    }
}
class Num extends Expr {
    int value;
    Num(int v) {value = v;}
}
```



AST 자료구조 – C 예




```
struct tokenType {
   int tokenNumber;
   char * tokenValue;
}

typedef struct nodeType{
   struct tokenType token; // 의미 있는 token 만 다룸
   struct nodeType * son;
   struct nodeType * brother;
}
// n-ary tree를 binary tree로 나타냄
```

S→ aAb
A→ aS | b

파싱 단계에서 AST 만들기 - LL

Recursive descent parser에서non terminal 프로시져를 변형

```
void pS() {
    if (nextSymbol == ta) {
            pa(); pA(); pb();
    }
}
```

```
Node_S pS() {
   if (nextSymbol == ta) {
            Node_a x1 = pa();
            Node_A x2 = pA();
            Node_b x3 = pb();
            return new Node_S(x1,x2,x3);
   } else return null;
}
```

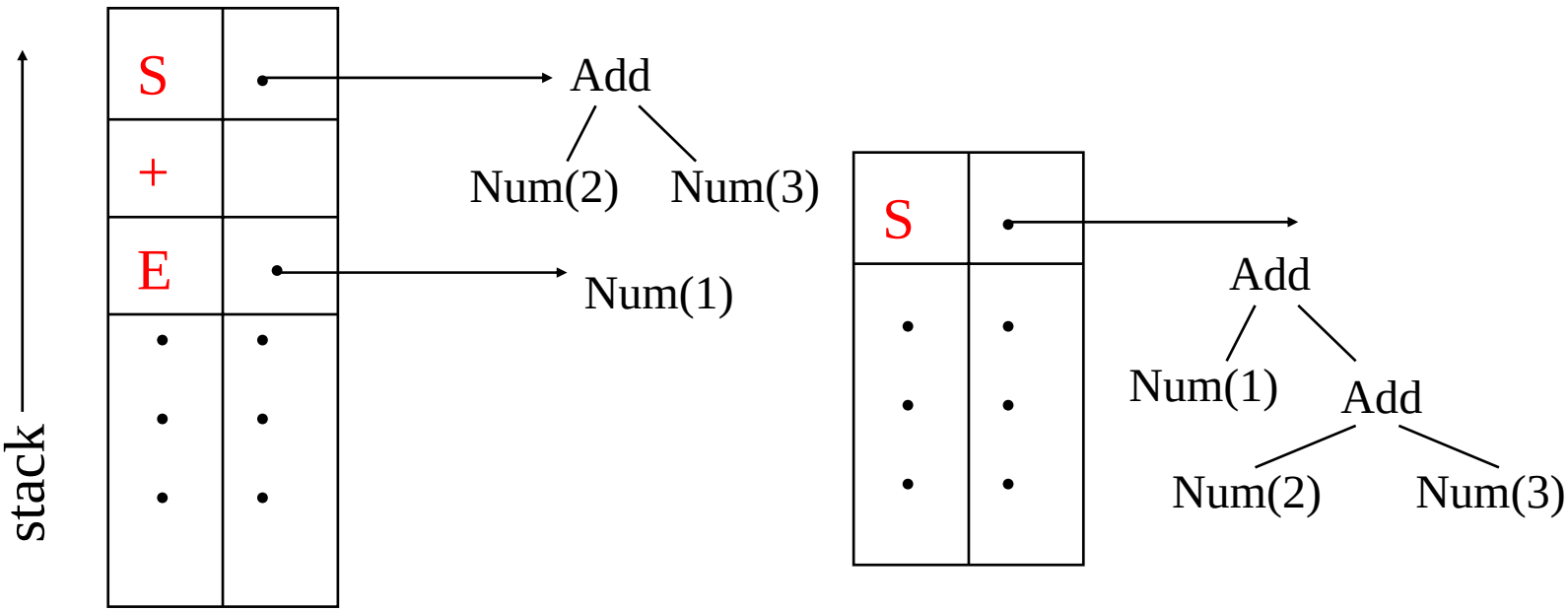
파싱 단계에서 AST 만들기 - LR

- Shift a : 의미있는 terminal일 경우 단말 노드를 만듬
- Reduce $A \rightarrow X_1X_2X_3$ ... :
  - 의미 있는 생성 규칙일 때 : 지금까지 만들었던 노드들을 한데 묶어 subtree를 구성
    - in C
      - 형제 노드를 연결하고 ($X_1X_2X_3$ ...)
      - 새 노드를 만들어 형제 노드를 자식 노드로 삼는다.
  - 의미 있는 생성 규칙이 아닐 때 : skip
    - in C : 새 노드를 만들지 않고 형제 노드만 연결한다.

파싱 단계에서 AST 만들기 – LR (예)

S → E + S | E
E → num | (S)

input string: “1 + 2 + 3”



Before reduction: S → E + S

After reduction: S → E + S

in Java

Class Problem

아래 구문을 LR 파싱할 때 언제 어떻게 AST가 만들어지게 되는지 간단히 정리하여 적으시오.

1. E → E + T	2. E → T	3. T → T * F
4. T → F	5. F → (E)	6. F → a

a + a * a	⇒ F + a * a	(6)
	⇒ T + a * a	(64)
	⇒ E + a * a	(642)
	⇒ E + F * a	(6426)
	⇒ E + T * a	(64264)
	⇒ E + T * F	(642646)
	⇒ E + T	(6426463)
	⇒ E	(64264631)

∴ reduce sequence : 64264631

Class Problem - 참고

과정	스 택	입력 버퍼	구문분석행동
0	$	a+a*a$	shift a
1	$a	+a*a$	reduce F → a
2	$F	+a*a$	reduce T → F
3	$T	+a*a$	reduce E → T
4	$E	+a*a$	shift +
5	$E+	a*a$	shift a
6	$E+a	*a$	reduce F → a
7	$E+F	*a$	reduce T → F
8	$E+T	*a$	shift *
9	$E+T*	a$	shift a
10	$E+T*a	$	reduce F → a
11	$E+T*F	$	reduce T → T*F
12	$E+T	$	reduce E → E+T
13	$E	$	accept

Syntax-Directed Definition

- AST
  - 파싱 단계에서 의미 있는 일을 함
    - LL : production rule derivation 할 때
    - LR : reduce 할 때
- Syntax-Directed Definition
  - AST처럼 파싱 단계에서 의미 있는 일을 하면서..
  - 여러 다양한 일을 할 수 있도록, 이걸 좀 체계적으로 확장
    - production 규칙마다 '할 일(!)'을 적어 놓음
      - 할일 = an associated semantic action (code)
    - 해당 production rule이 reduce 될 때 (LR 경우), derive 될 때 (LL 경우) '할 일'이 수행됨
  - 파서 생성기 (즉, yacc 등)에서 많이 사용
    - 생성된 파서에 삽입될 semantic action을 사용자가 정의가능

Semantic Actions

- Action은 파서의 스택을 접근하는 코드
  - 스택 entry로, terminal, nonterminal, 파서 상태 이런 것들 말고도 좀 추가해야함.
  - 앞서 LR에서 AST 만들 때 스택에 트리 넣은 것도 이것의 일종
- E → E + E 규칙에 대한 action
  - 세 E 가 구별이 되야 action code를 쓰므로...
  - yacc/bison에서는 $$, $1, $2, ...

```
eg. expr ::= expr PLUS expr {$$ = $1 + $3;}
```

- SDD 예1) AST를 yacc에서 SDD 스타일로 만들기

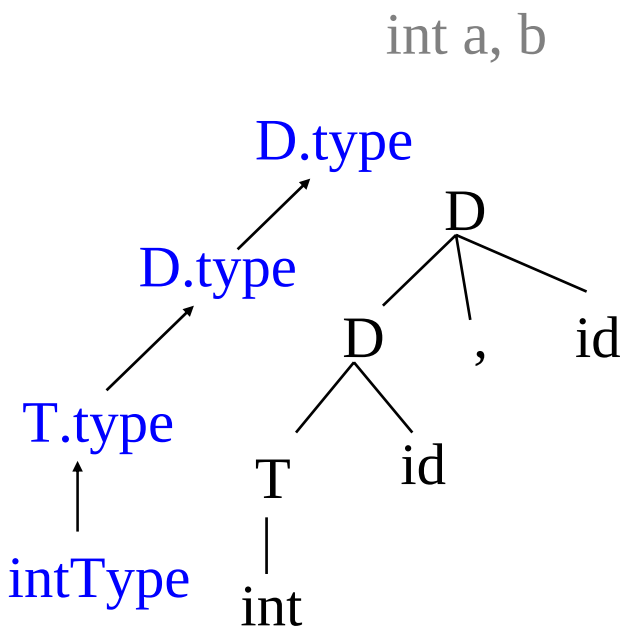
```
expr ::= NUM                {$$ = new Num($1.val); }
expr ::= expr PLUS expr     {$$ = new Add($1, $3); }
expr ::= expr MULT expr     {$$ = new Mul($1, $3); }
expr ::= LPAR expr RPAR     {$$ = $2; }
```

SDD 예2) Type Declaration

{AddType($2, $1.type);
$$.type = $1.type; }

in yacc

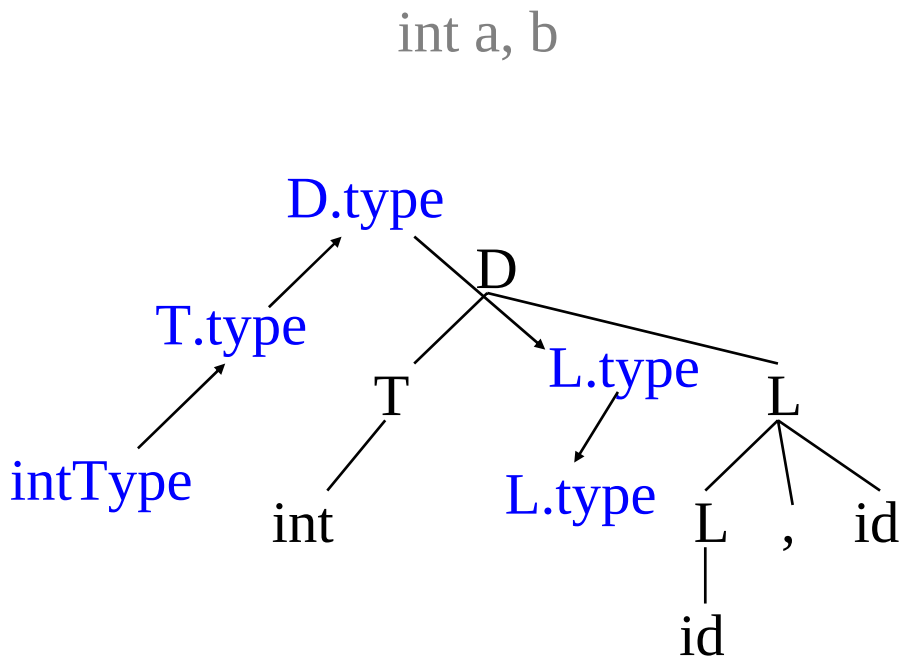
D → T id	{AddType(id, T.type); D.type = T.type; }
D → D1, id	{AddType(id, D1.type); D.type = D1.type; }
T → int	{T.type = intType; }
T → float	{T.type = floatType; }



값이 bottom-up으로 propagation

SDD 예3) Type Declaration

D → TL	{AddType(id, T.type); D.type = T.type; L.type = D.type; }
T → int	{T.type = intType; }
T → float	{T.type = floatType; }
L → L1, id	{AddType(id, L1.type); ??? }
L → id	{AddType(id, ???); }



값이 bottom-up/top-down 둘 다로 propagation!!

속성 (Attributes)

- AST vs. SDD
  - AST가 SDD에 의해 정의되는 것으로 보기도 하지만,
  - SDD를 AST의 각 노드에 속성을 붙여 evaluation 해가는 것으로 보기도 함.

- 속성의 종류 (A→XYZ 에서)
  - synthesized attr.
    - children에 의해 계산 (bottom-up)
    - A.attr = f(X.attr, Y.attr, Z.attr);
    - terminal은 synthesized attr. 뿐
  - inherited attr.
    - parent, sibling에 의해 계산
    - Y.attr = f(A.attr, X.attr, Z.attr);

Attribute Evaluation

- Parse tree method
  - AST 만드는 순서를 속성값 계산 순서대로 조정
    - 속성값 계산 순서 : 속성간에 계산 dependecy graph를 만들고 topological sort 를 하면 나옴
  - dependency graph의 cycle 등 있으면 fail위험
- Rules based
  - 각 production 마다 attr. evaluation 순서를 미리 정의
- On-the-fly
  - node 방문 순서대로 그냥 따라 evaluation (e.g., LL 일 때 topdown, LR 일때 bottom-up)
  - 가장 efficient, but restriction이 있음 (다음 경우만...LR 관점)
    - S-attributed SDD : synthesized attr. 만 가지고 있던지,
    - L-attributed SDD : synthesized attr. 만 가지거나, 아니면 값이 왼쪽에서 오른쪽으로 흘러 계산이 이루어지는 경우

Semantic Analysis

- 의미분석 (Semantic analysis)
  - 프로그램 constructs (변수, 객체, 식, 문장...) 의 올바른 사용 여부를 분석, 확인
    - Scope 관련 : 변수가 선언되기 전에 쓰였나? 두 번 정의됐나?
    - 타입 관련 : 변수와 assign되는 값과 서로 타입이 맞는가?
  - 의미분석 ≅"속성 계산하고 값 check"
- Single Pass analysis
  - 기존AST에 첨부된 속성 계산하고 값 check
    - mixed with parser code
  - 장점 – 효율, 단점 - AST 바꾸고 싶을 때 checking 도 바꿔야
- Multi Pass analysis
  - (code 생성이 아닌) Semantic checking을 위한 별도의 AST를 만들고, 이 tree를 traverse 하면 check
  - trusted

Class Problem

다음에서 오류가 발생된다면
어휘분석, 구문분석, 의미분석 중 어디에 속하는지 알아보시오.

```
int a;
a = 1.0;
```

```
int a; b
b = a;
```

```
{ int a;
   a = 1;
}
{ a = 2;
}
```

```
in a;
a = 1;
```

```
int foo(int a)
{
 foo = 3;
}
```