Chapter 2

2. 컴파일러 구조: Part II

Review

1	E	$\rightarrow$	I
2	E	$\rightarrow$	$E + E$
3	E	$\rightarrow$	$E * E$
4	E	$\rightarrow$	(E)
5	I	$\rightarrow$	a
6	I	$\rightarrow$	b
7	I	$\rightarrow$	Ia
8	I	$\rightarrow$	Ib
9	I	$\rightarrow$	$I0$
10	I	$\rightarrow$	$I1$

Review

1	E	$\rightarrow$	I
2	E	$\rightarrow$	$E + E$
3	E	$\rightarrow$	$E * E$
4	E	$\rightarrow$	(E)
5	I	$\rightarrow$	a
6	I	$\rightarrow$	b
7	I	$\rightarrow$	Ia
8	I	$\rightarrow$	Ib
9	I	$\rightarrow$	$I0$
10	I	$\rightarrow$	$I1$

Nonterminal ?
Terminals ?
Can you rewrite the productions like this?

E → I | E + E | E * E | (E)

Review

1	E	$\rightarrow$	I
2	E	$\rightarrow$	$E + E$
3	E	$\rightarrow$	$E * E$
4	E	$\rightarrow$	(E)
5	I	$\rightarrow$	a
6	I	$\rightarrow$	b
7	I	$\rightarrow$	Ia
8	I	$\rightarrow$	Ib
9	I	$\rightarrow$	$I0$
10	I	$\rightarrow$	$I1$

Is the following string correct?

a
b
b0
b00
a+b00
(a+b00)
a(a+b00)

Review

1	E	$\rightarrow$	I
2	E	$\rightarrow$	$E + E$
3	E	$\rightarrow$	$E * E$
4	E	$\rightarrow$	(E)
5	I	$\rightarrow$	a
6	I	$\rightarrow$	b
7	I	$\rightarrow$	Ia
8	I	$\rightarrow$	Ib
9	I	$\rightarrow$	$I0$
10	I	$\rightarrow$	$I1$

Write more than two correct strings for the grammar.

수업 목표

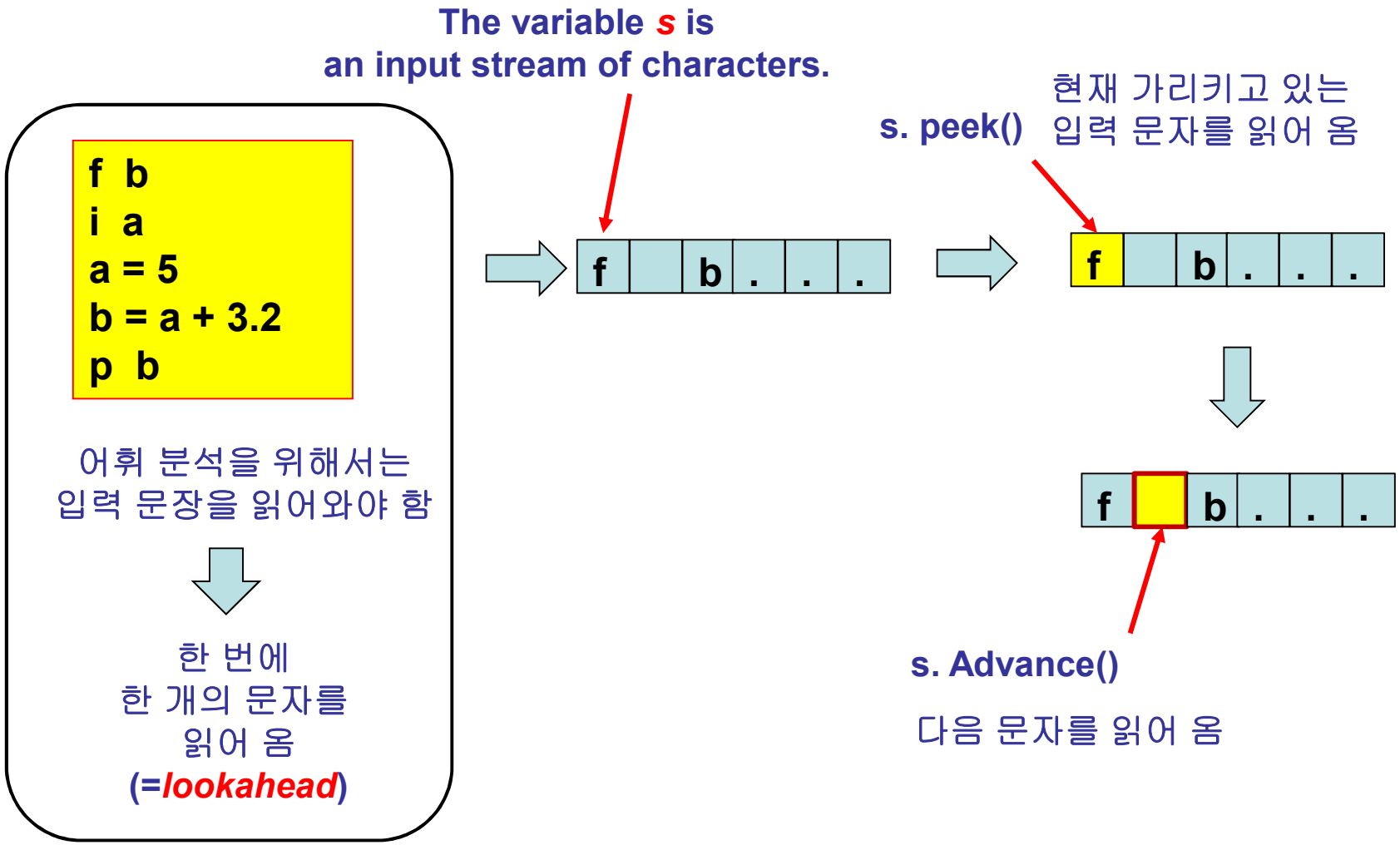
■ 컴파일 과정에 대한 개념 정립

- 간단한 프로그래밍 언어에 대한 컴파일 전 과정을 자세히 들여다 봄

Phases of a Simple Compiler

- **Scanner**
- **Parser**
  - Recursive descent 방식을 사용
- **Semantic analysis**

Scanner for the ac language

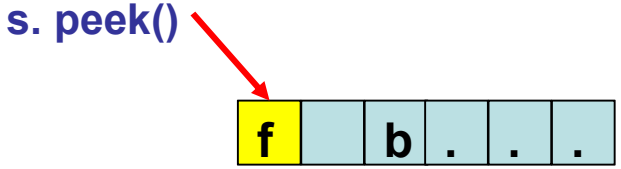


Scanner for the ac language

```
function SCANNER() returns Token
    while s.peek() = blank do call s.Advance( )
    if s.EOF( )
    then ans.type ← $
    else
        if s.peek() ∈ {0, 1, ..., 9}
        then ans ← ScanDigits( )
        else
            ch ← s.Advance( )
            switch (ch)
                case {a, b, ..., z} − {i, f, p}
                    ans.type ← id
                    ans.val ← ch
                case f
                    ans.type ← floatdcl
                case i
                    ans.type ← intdcl
                case p
                    ans.type ← print
                case =
                    ans.type ← assign
                case +
                    ans.type ← plus
                case -
                    ans.type ← minus
                case default
                    call LexicalError()
    return (ans)
end
```

Scanner for the ac language

```
f b
i a
a = 5
b = a + 3.2
p b
```



입력에서 읽어 온 문자가 **'f'**

Terminal 기호 중 floatdcl

Token 종류(type)는 floatdcl

ans. type ← floatdcl

Terminal = Token

Terminal	Regular Expression
floatdcl	"f"
intdcl	"i"
print	"p"
id	[a – e] | [g – h] | [j – o] | [q – z]
assign	"="
plus	"+"
minus	"-"
inum	$[0-9]^+$
fnum	$[0-9]^+.[0-9]^+$
blank	(" ")$^+$

type **value**

Scanner for the ac language

Terminal	Regular Expression
floatdcl	"f"
intdcl	"i"
print	"p"
id	[a – e] | [g – h] | [j – o] | [q – z]
assign	"="
plus	"+"
minus	"-"
inum	$[0-9]^+$
fnum	$[0-9]^+.[0-9]^+$
blank	(" ")$^+$

```
function SCANNER() returns Token
    while s.peek() = blank do call s.Advance( )
    if s.EOF( )
    then ans.type ← $
    else
        if s.peek() ∈ {0, 1, ..., 9}
        then ans ← ScanDigits( )
        else
            ch ← s.Advance( )
            switch (ch)
                case {a, b, ..., z} – {i, f, p}
                    ans.type ← id
                    ans.val ← ch
                case f
                    ans.type ← floatdcl
                case i
                    ans.type ← intdcl
                case p
                    ans.type ← print
                case =
                    ans.type ← assign
                case +
                    ans.type ← plus
                case -
                    ans.type ← minus
                case default
                    call LexicalError()
    return (ans)
end
```

Scanner for the ac language

입력 문자가
end of file(EOF)이면
더 이상 읽을
문자가 없다는 뜻!

입력 문자가
숫자이면

```
function SCANNER() returns Token
    while s.peek() = blank do call s.Advance( )
    if s.EOF( )
    then ans.type ← $
    else
        if s.peek() ∈ {0, 1, ..., 9}
        then ans ← ScanDigits( )
        else
            ch ← s.Advance( )
            switch (ch)
                case {a, b, ..., z} − {i, f, p}
                    ans.type ← id
                    ans.val ← ch
                case f
                    ans.type ← floatdcl
                case i
                    ans.type ← intdcl
                case p
                    ans.type ← print
                case =
                    ans.type ← assign
                case +
                    ans.type ← plus
                case -
                    ans.type ← minus
                case default
                    call LexicalError()
    return (ans)
end
```

Scanner for the ac language : ScanDigits (1/2)

```
function ScanDigits( ) returns token
    tok.val ← " "
    while s.peek() ∈ {0, 1, . . . , 9} do
        tok.val ← tok.val + s.Advance( )
    if s.peek() ≠ "."
    then tok.type ← inum
    else
        tok.type ← fnum
        tok.val ← tok.val + s.Advance( )
        while s.peek() ∈ {0, 1, . . . , 9} do
            tok.val ← tok.val + s.Advance( )
    return (tok)
end
```

소수점이 없으면 정수(inum)

정수(inum)란
한 개 이상의
숫자(0~9)로 이루어짐
1, 123, 0000, …

inum	$[0-9]^+$
fnum	$[0-9]^+.[0-9]^+$

실수(fnum)란
소수점(.)이 있는 숫자
0.0, 12.0, 123.456, …

Finding inum or fnum tokens for the ac language.

Scanner for the ac language : ScanDigits (2/2)

```
function ScanDigits( ) returns token
    tok.val ← " "
    while s.peek() ∈ {0, 1, . . . , 9} do
        tok.val ← tok.val + s.Advance( )
    if s.peek() ≠ "."
    then tok.type ← inum
    else
        tok.type ← fnum
        tok.val ← tok.val + s.Advance( )
        while s.peek() ∈ {0, 1, . . . , 9} do
            tok.val ← tok.val + s.Advance( )
    return (tok)
end
```

+는 덧셈 연산이 아니라 **string concatenation**

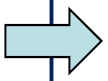
"alpha" + "go"
→ "alphago"

inum	$[0-9]^+$
fnum	$[0-9]^+.[0-9]^+$

Finding inum or fnum tokens for the ac language.

컴파일러 구조

Terminal	Regular Expression
floatdcl	"f"
intdcl	"i"
print	"p"
id	[a – e] | [g – h] | [j – o] | [q – z]
assign	"="
plus	"+"
minus	"-"
inum	$[0-9]^+$
fnum	$[0-9]^+.[0-9]^+$
blank	(" ")$^+$



```
function SCANNER() returns Token
    while s.peek() = blank do call s.Advance( )
    if s.EOF( )
    then ans.type ← $
    else
        if s.peek() ∈ {0, 1, ..., 9}
        then ans ← ScanDigits( )
        else
            ch ← s.Advance( )
            switch (ch)
                case {a, b, ..., z} – {i, f, p}
                    ans.type ← id
                    ans.val ← ch
                case f
                    ans.type ← floatdcl
                case i
                    ans.type ← intdcl
                case p
                    ans.type ← print
                case =
                    ans.type ← assign
                case +
                    ans.type ← plus
                case -
                    ans.type ← minus
                case default
                    call LexicalError()
    return (ans)
end
```

Recursive-descent parsing

■The simplest parsing technique

- 구문 트리(syntax tree)에서 부모 노드로부터 자식 노드에 연결된 링크를 따라 아래로 내려가는(descent) 방식
  - 이 과정에서 반복(recursive) 호출이 발생

- Nonterminal은 자신의 이름으로 된 parsing procedure를 갖는다.
  - 비단말기호의 오른쪽(RHS)에 놓인 기호들을 순서대로 찾는다.
  - It is responsible for determining if the token stream contains a sequence of tokens ***derivable from that nonterminal***.

Context-free grammar for ac

1	Prog	→	Dcls Stmts $
2	Dcls	→	Dcl Dcls
3		|	λ
4	Dcl	→	floatdcl id
5		|	intdcl id
6	Stmts	→	Stmt Stmts
7		|	λ
8	Stmt	→	id assign Val Expr
9		|	print id
10	Expr	→	plus Val Expr
11		|	minus Val Expr
12		|	λ
13	Val	→	id
14		|	inum
15		|	fnum

Context-free grammar for ac.

Recursive-descent parsing

```
Stmt → id assign Val Expr
     | print id
```

```
procedure STMT( )
    if ts. peek( ) = id
    then
        call Match (ts, id )
        call Match (ts, assign)
        call Val ( )
        call Expr( )
    else
        if ts. peek( ) = print
        then
            call Match (ts, print )
            call Match (ts, id )
        else
            call Error( )                    (7)
end
```

(6)

2개의 production 중 무엇을 선택할까?

terminal은 입력과 직접 비교가 가능
➔ Match (*ts*, id)는 production의 terminal id 와 입력에서 읽어 온 token(*ts*)을 비교

Nonterminal은 해당 프로시저를 호출

Recursive-descent parsing procedure for Stmt. The variable *ts* is an input stream of tokens.

Recursive-descent parsing

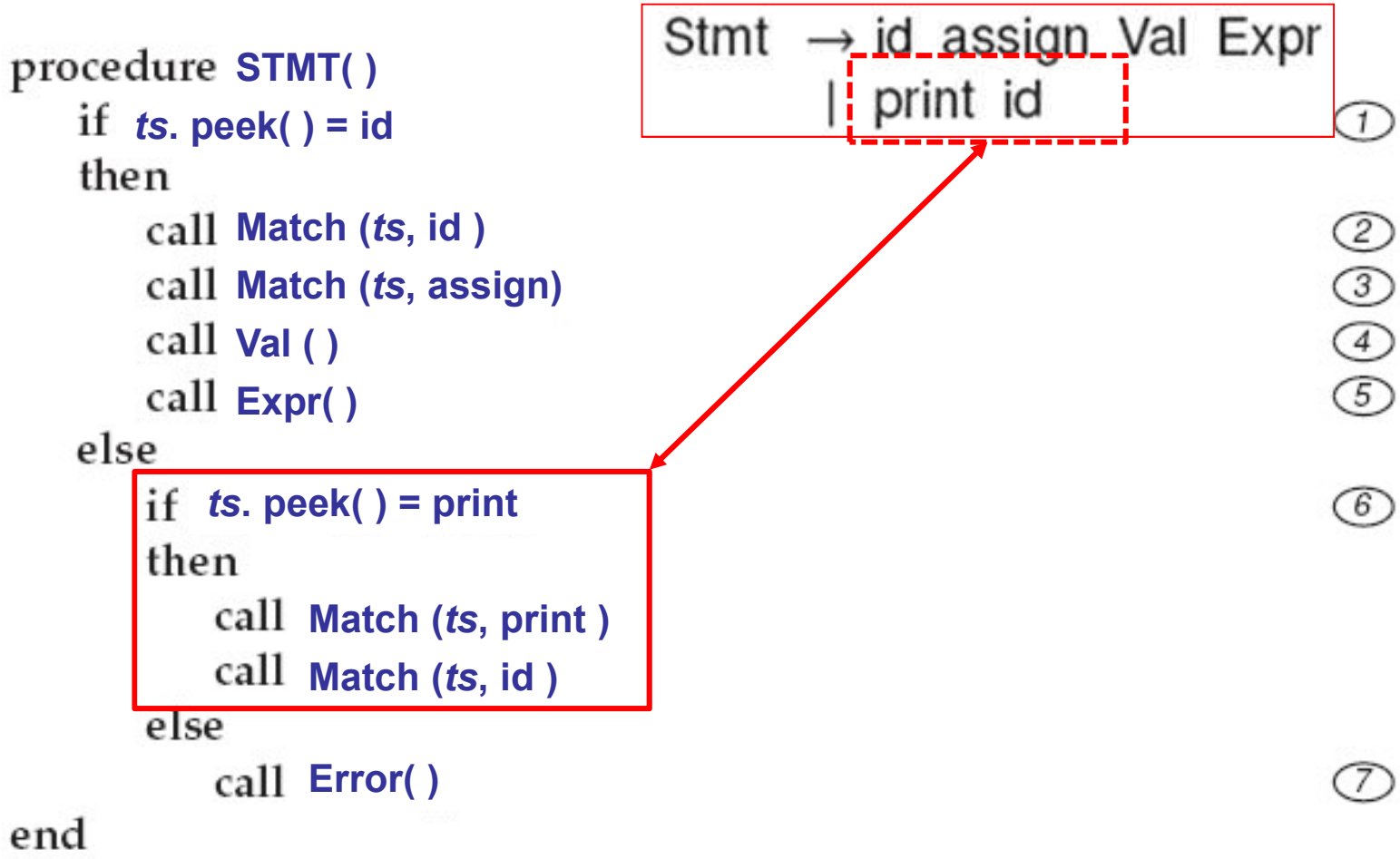
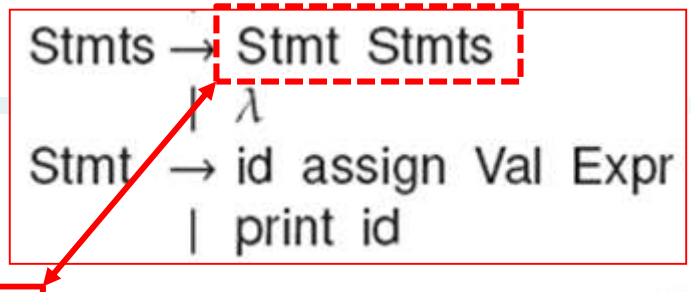


Figure 2.7: Recursive-descent parsing procedure for Stmt. The variable *ts* is an input stream of tokens.

Recursive-descent parsing




```
procedure STMTS( )
    if ts. peek( ) = id or ts. peek( ) = print                    (8)
    then
        call STMT( )                                              (9)
        call STMTS( )                                             (10)
    else
        if ts. peek( ) = $                                        (11)
        then
            /★  do nothing for λ-production                ★/     (12)
        else call Error( )
end
```

Recursive-descent parsing procedure for Stmts.

Recursive-descent parsing

```
Stmts → Stmt  Stmts
      | λ
Stmt  → id  assign  Val  Expr
      | print  id
```

```
procedure  STMTS( )
    if  ts. peek( ) = id  or  ts. peek( ) = print                    (8)
    then
        call  STMT( )                                                (9)
        call  STMTS( )                                               (10)
    else
        if  ts. peek( ) = $                                          (11)
        then
            /★   do nothing for λ-production                         ★/ (12)
        else  call  Error( )
end
```

Recursive-descent parsing procedure for Stmts.

Write a procedure for each Nonterminal listed in the following.

```
2  Dcls  → Dcl  Dcls
3        |  λ
```

Hint: 3번 규칙은 **ts.peek( )**의 값이 **id, print** 일 때

```
4  Dcl   → floatdcl  id
5        |  intdcl  id
```

```
10  Expr  → plus  Val  Expr
11        |  minus  Val  Expr
12        |  λ
```

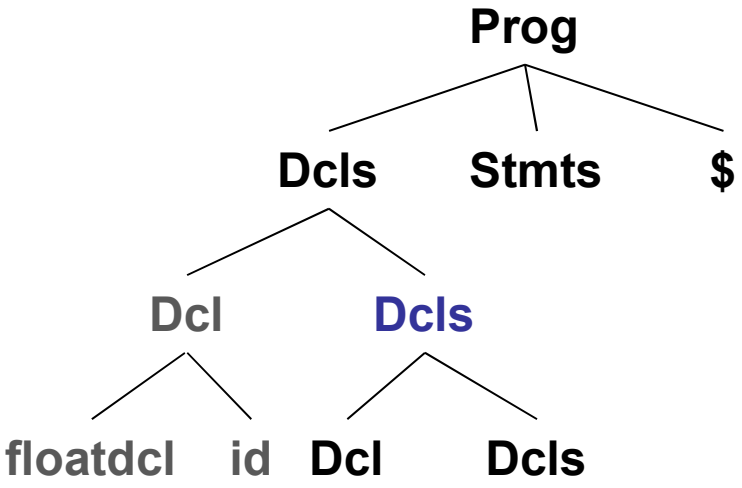
Hint: 12번 규칙은 ***ts.peek( )***의 값이 **id, print** 일 때

```
13  Val   → id
14        |  inum
15        |  fnum
```

토큰을 저장하고 있는 전역 변수 : ***ts***
Terminal 기호와 비교하는 함수 : ***Match(ts, Terminal)***

Draw the parse tree for a given input.

```
f  b
i  a
a = 5
b = a + 3.2
p  b
```



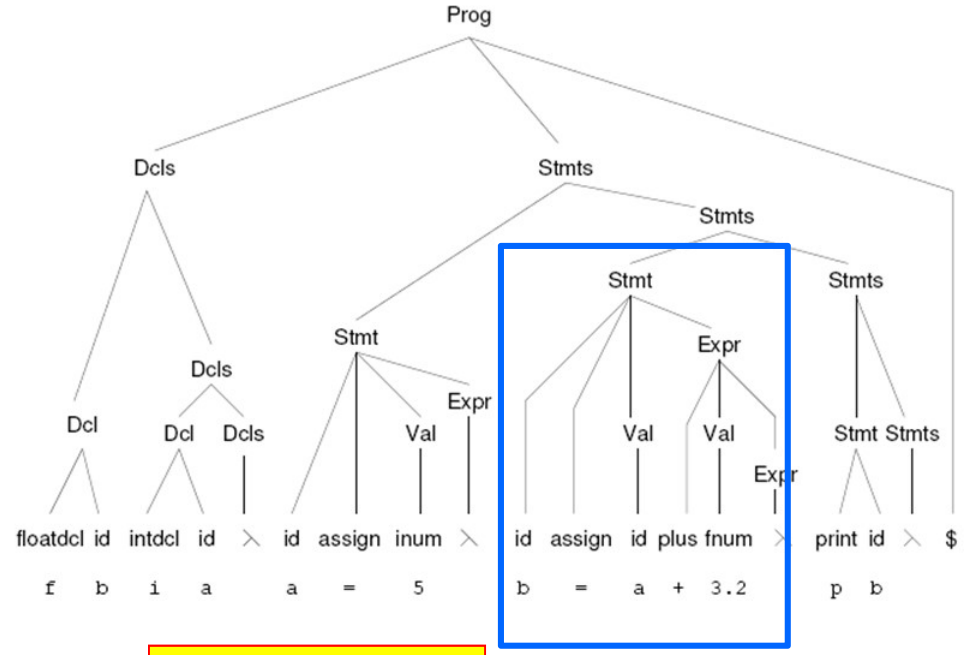
1	Prog	→	Dcls Stmts $
2	Dcls	→	Dcl Dcls
3		|	λ
4	Dcl	→	floatdcl id
5		|	intdcl id
6	Stmts	→	Stmt Stmts
7		|	λ
8	Stmt	→	id assign Val Expr
9		|	print id
10	Expr	→	plus Val Expr
11		|	minus Val Expr
12		|	λ
13	Val	→	id
14		|	inum
15		|	fnum

Abstract Syntax Tree (AST)

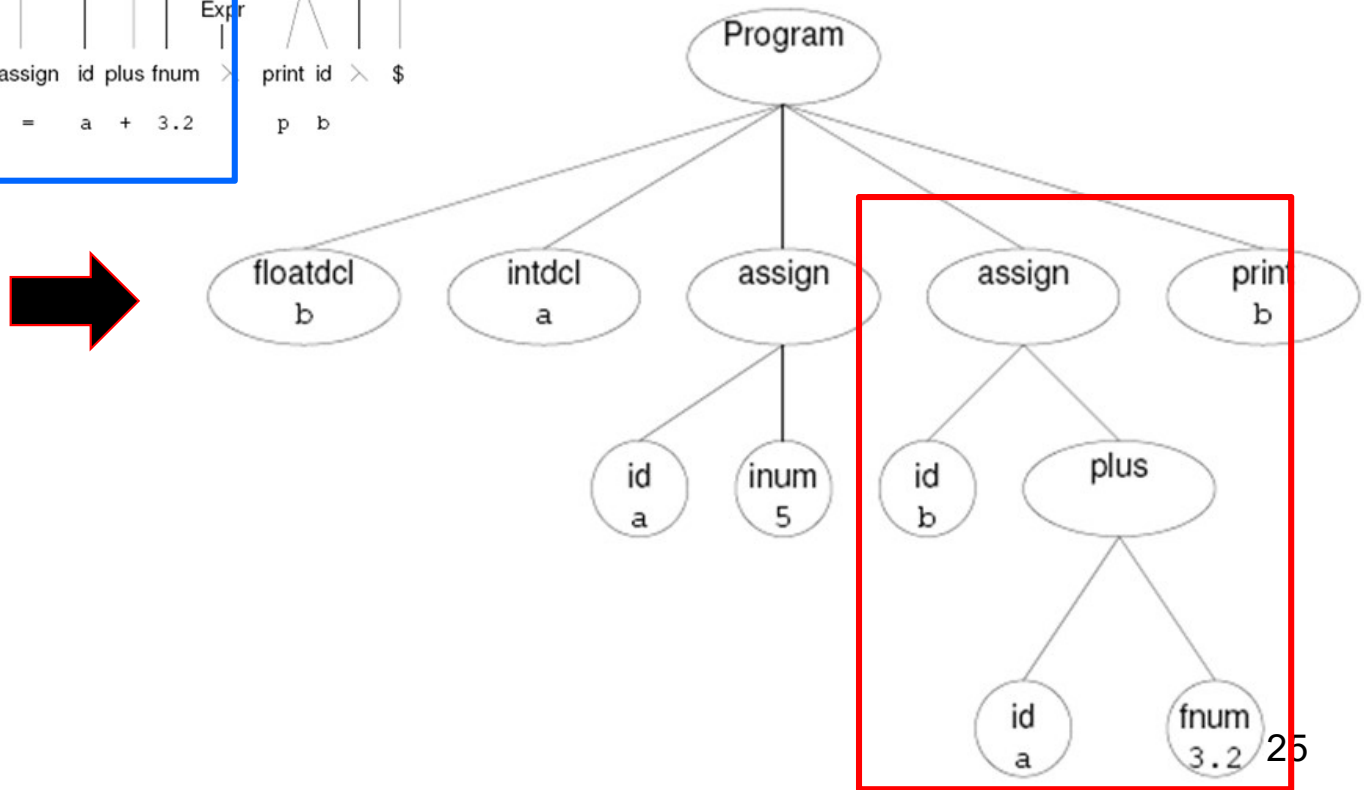
- **Parse tree**는 필요 없는 정보들을 많이 갖고 있기 때문에 복잡하다.
  - AST는 parse tree로부터 꼭 필요한 정보만을 가져오기 때문에 구조가 간단해진다.

- **AST serves as a *common, intermediate representation* of a program.**
  - 문장의 실행 순서가 구체적으로 나타나야 한다.
  - 데이터 형 선언은 한 개의 node로 나타낸다.
  - 할당 문을 표시하는 node는 LHS의 identifier가 left child가 된다.
  - 연산을 표시하는 node는 연산 종류만 있으면 된다.
  - **print** 문은 출력할 변수만 있으면 된다.

AST 구성(1/5) : 실행 순서를 구체적으로 표현 (+ ➔ assign)



f b
i a
a = 5
b = a + 3.2
p b

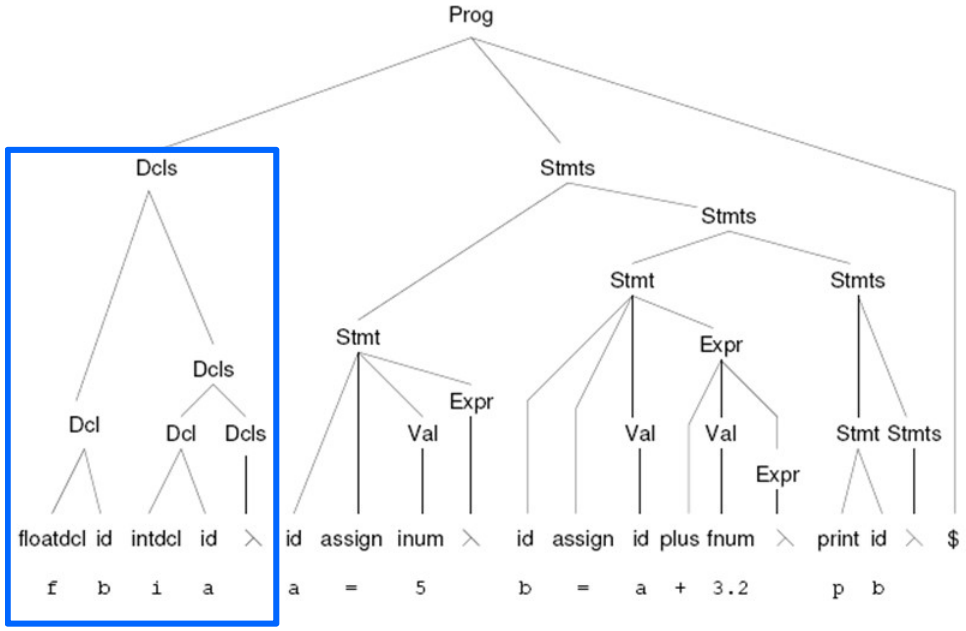


Abstract Syntax Tree

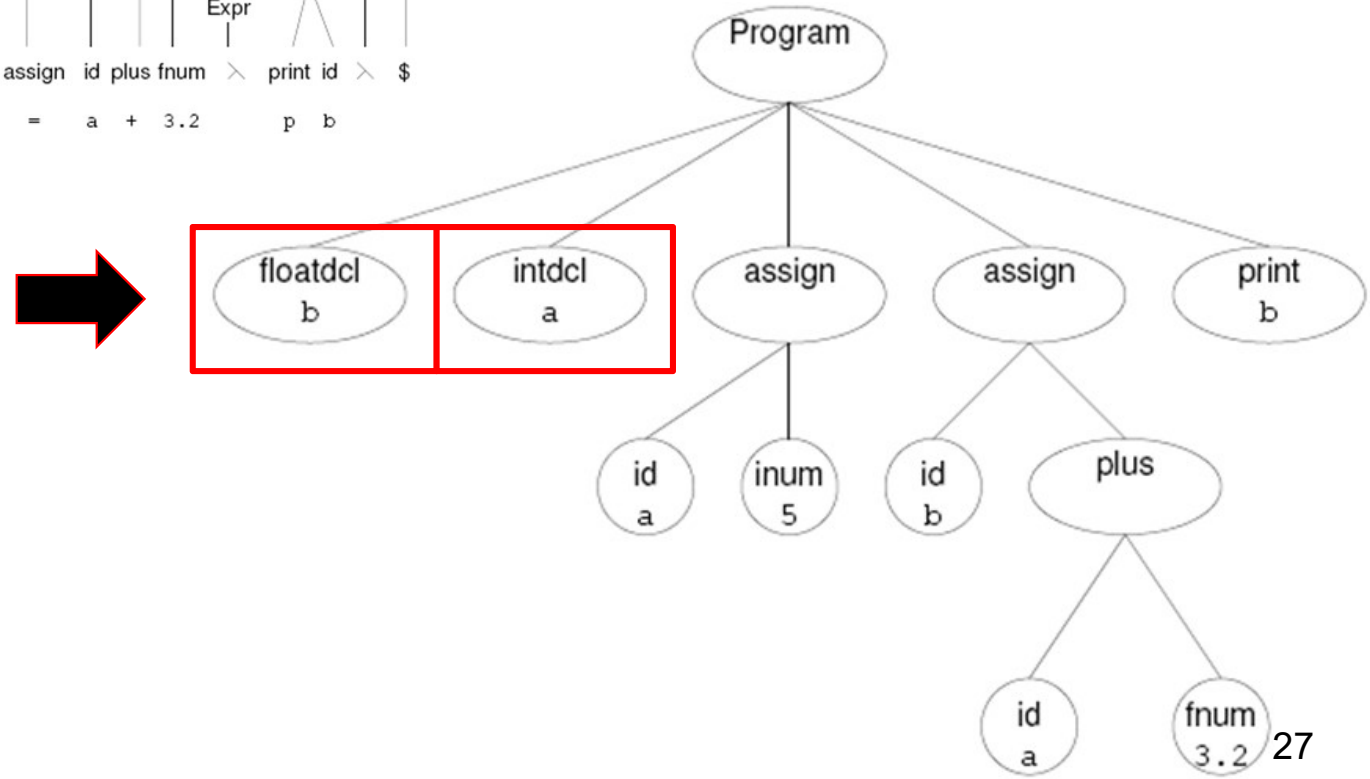
- **Parse tree**는 필요 없는 정보들을 많이 갖고 있기 때문에 복잡하다.
  - AST는 parse tree로부터 꼭 필요한 정보만을 가져오기 때문에 구조가 간단해진다.

- **AST serves as a *common, intermediate representation* of a program.**
  - 문장의 실행 순서가 구체적으로 나타나야 한다.
  - 데이터 형 선언은 한 개의 **node**로 나타낸다.
  - 할당 문을 표시하는 node는 LHS의 identifier가 left child가 된다.
  - 연산을 표시하는 node는 연산 종류만 있으면 된다.
  - **print** 문은 출력할 변수만 있으면 된다.

AST 구성(2/5) : 데이터 형 선언은 하나의 node로 표시



f b
i a
a = 5
b = a + 3.2
p b

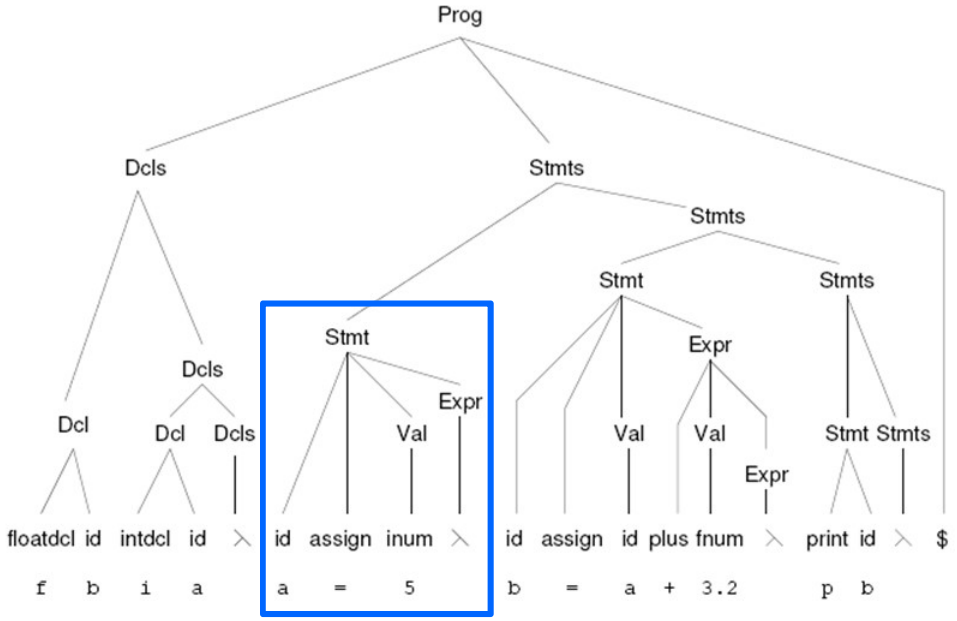


Abstract Syntax Tree

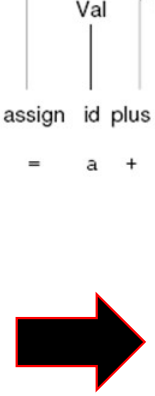
- **Parse tree**는 필요 없는 정보들을 많이 갖고 있기 때문에 복잡하다.
  - AST는 parse tree로부터 꼭 필요한 정보만을 가져오기 때문에 구조가 간단해진다.

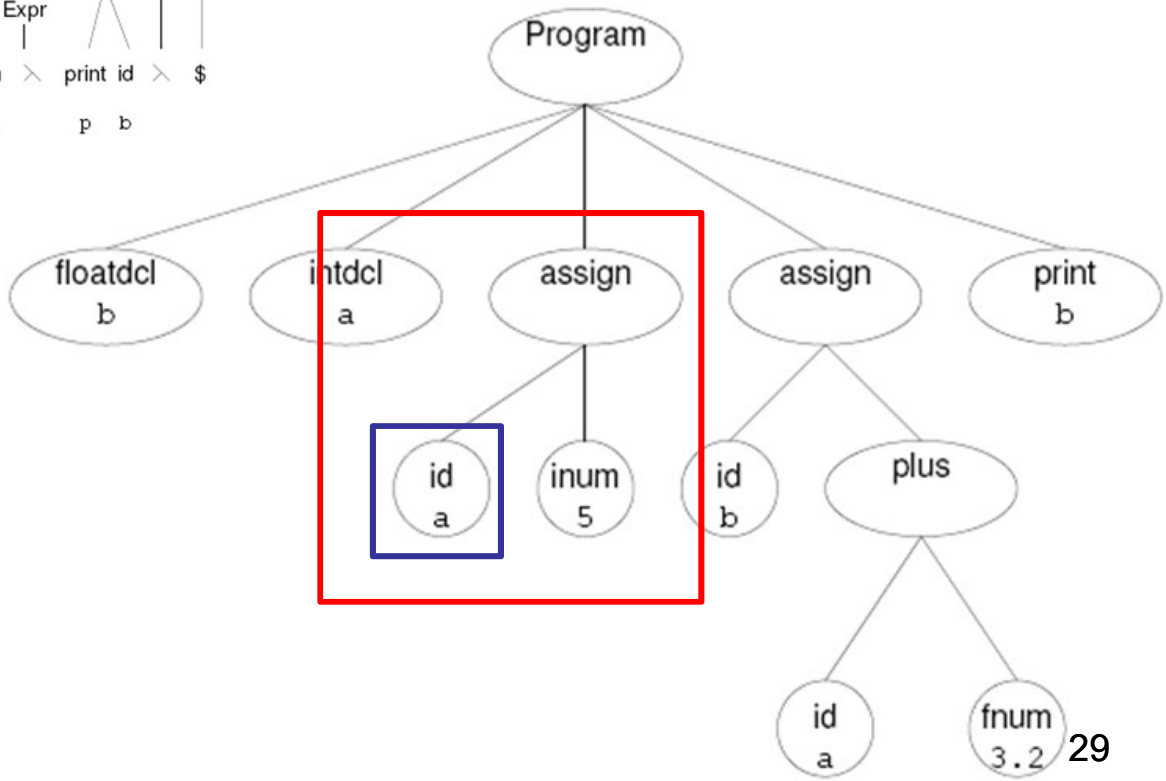
- **AST serves as a *common, intermediate representation* of a program.**
  - 문장의 실행 순서가 구체적으로 나타나야 한다.
  - 데이터 형 선언은 한 개의 node로 나타낸다.
  - 할당 문을 표시하는 **node**는 **LHS**의 **identifier**가 **left child**가 된다.
  - 연산을 표시하는 node는 연산 종류만 있으면 된다.
  - **print** 문은 출력할 변수만 있으면 된다.

AST 구성(3/5) : 할당 문에서 LHS identifier가 left child



f b
i a
a = 5
b = a + 3.2
p b



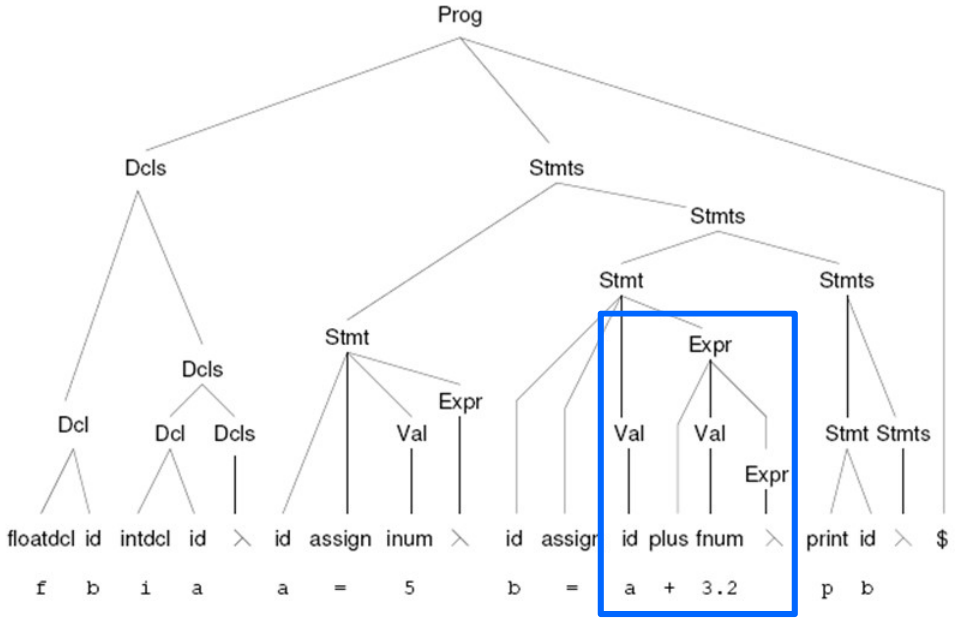


Abstract Syntax Tree

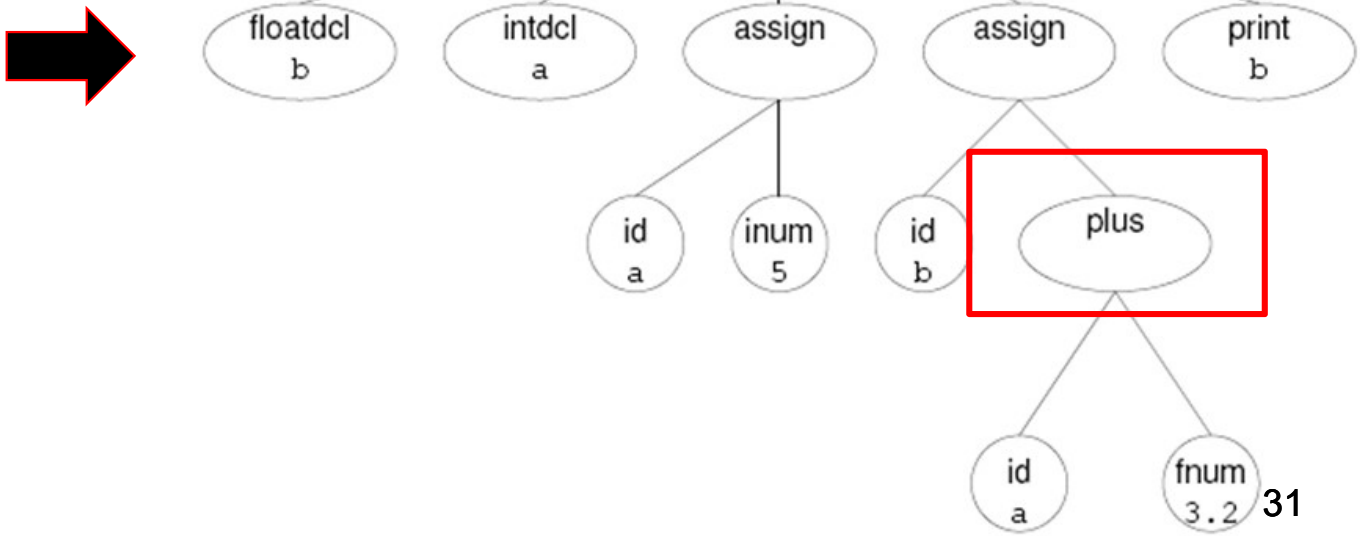
- **Parse tree**는 필요 없는 정보들을 많이 갖고 있기 때문에 복잡하다.
  - AST는 parse tree로부터 꼭 필요한 정보만을 가져오기 때문에 구조가 간단해진다.

- **AST serves as a *common, intermediate representation* of a program.**
  - 문장의 실행 순서가 구체적으로 나타나야 한다.
  - 데이터 형 선언은 한 개의 node로 나타낸다.
  - 할당 문을 표시하는 node는 LHS의 identifier가 left child가 된다.
  - 연산을 표시하는 **node**는 연산 종류만 있으면 된다.
  - **print** 문은 출력할 변수만 있으면 된다.

AST 구성(4/5) : 연산 node는 어떤 연산인지 표시




```
f  b
i  a
a = 5
b = a + 3.2
p  b
```

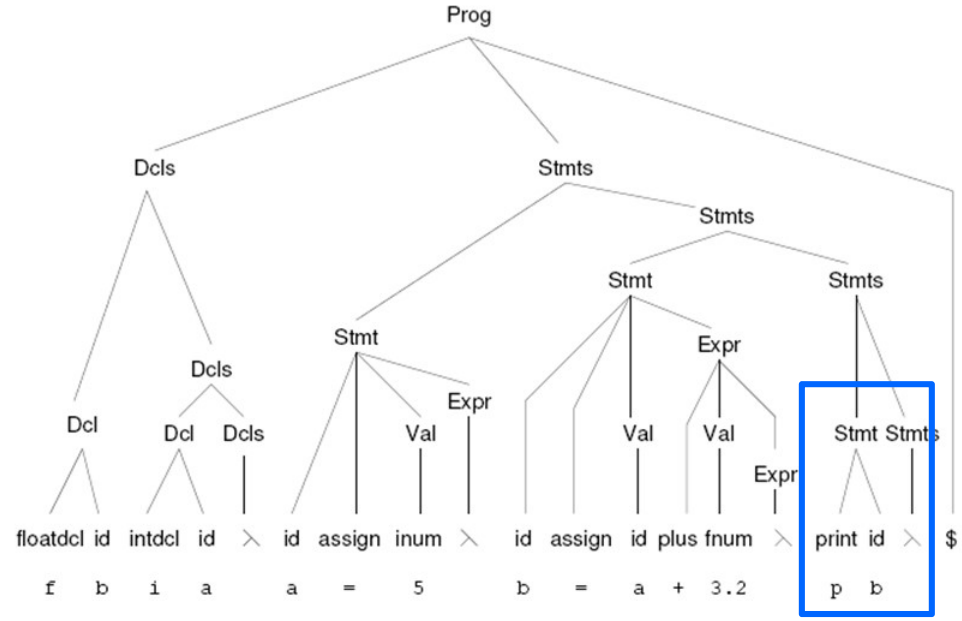


Abstract Syntax Tree

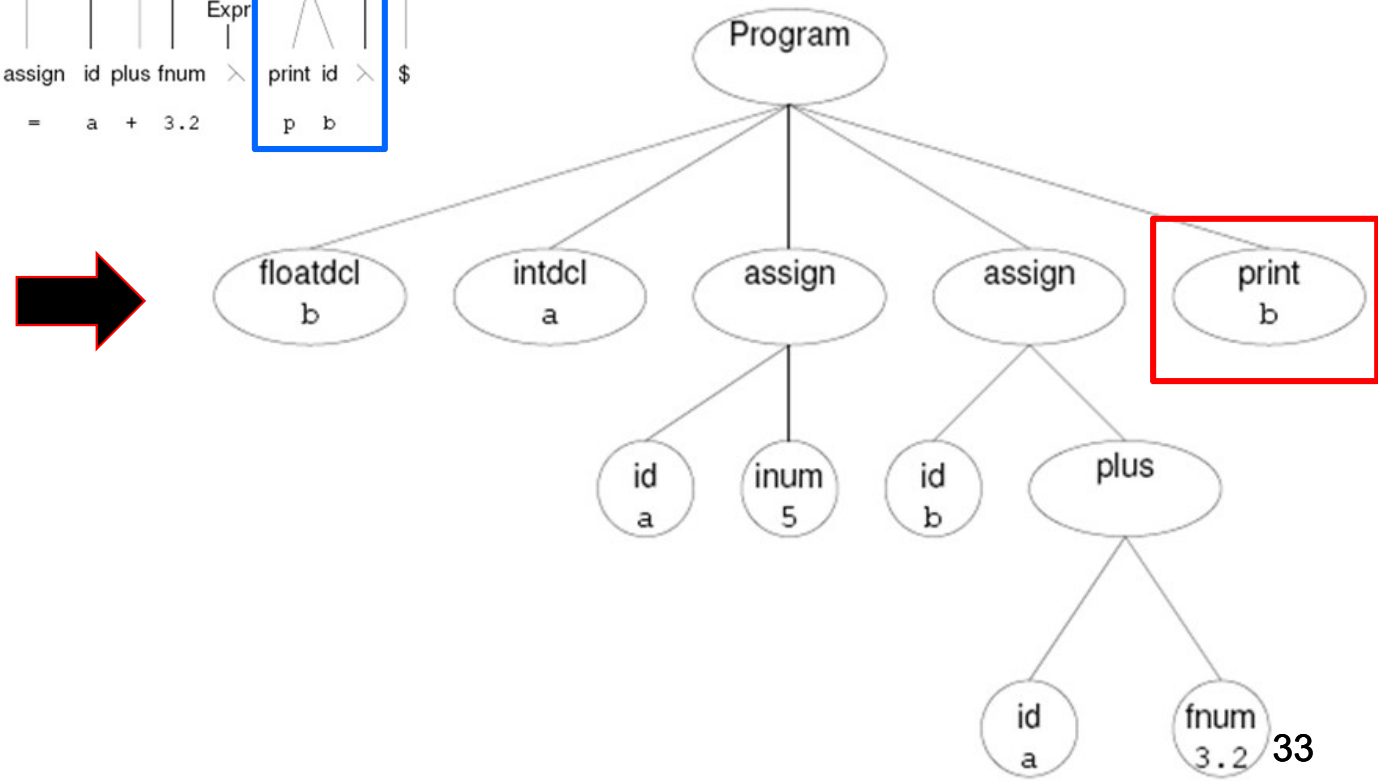
- **Parse tree**는 필요 없는 정보들을 많이 갖고 있기 때문에 복잡하다.
  - AST는 parse tree로부터 꼭 필요한 정보만을 가져오기 때문에 구조가 간단해진다.

- **AST serves as a *common, intermediate representation* of a program.**
  - 문장의 실행 순서가 구체적으로 나타나야 한다.
  - 데이터 형 선언은 한 개의 node로 나타낸다.
  - 할당 문을 표시하는 node는 LHS의 identifier가 left child가 된다.
  - 연산을 표시하는 node는 연산 종류만 있으면 된다.
  - **print 문은 출력할 변수만 있으면 된다.**

AST 구성(5/5) : print 문은 출력할 변수만 표시



f b
i a
a = 5
b = a + 3.2
p b

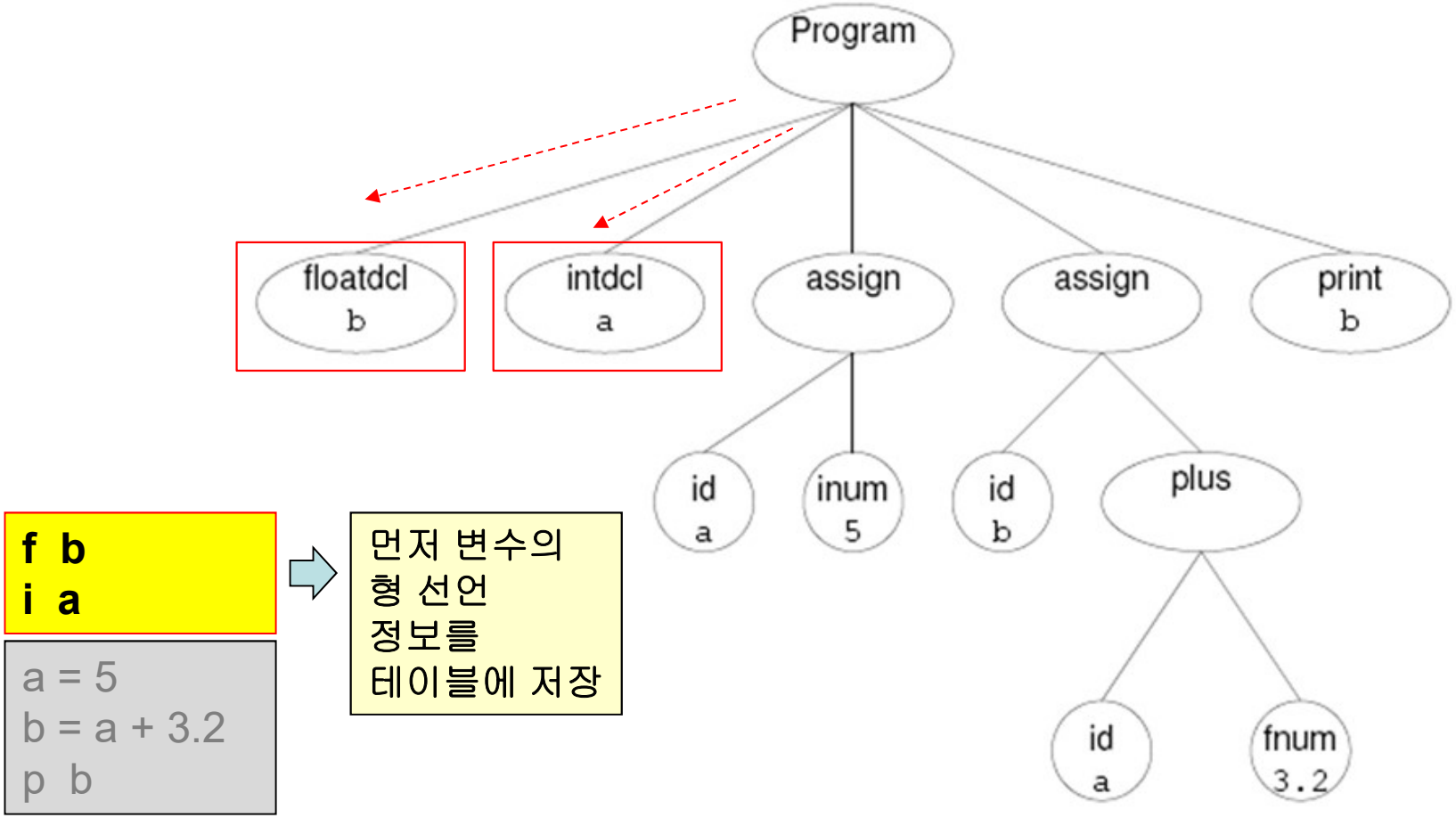


Semantic Analysis (의미 분석)

- 구문 분석 과정에서 처리하지 못했던 **language definition**에 관한 상세 내용들을 처리
  - Identifier 선언 및 범위 (scope)가 적절한가?
  - 사용자 정의 형 (type)이 알맞게 선언되었는가?
  - 연산 및 메모리 참조 과정에서 type이 일치하는가?

  - 예: a + b ➔ 실수 + 정수 ➔ 실수 덧셈
    - 연산 결과를 실수 형으로 저장

Semantic Analysis



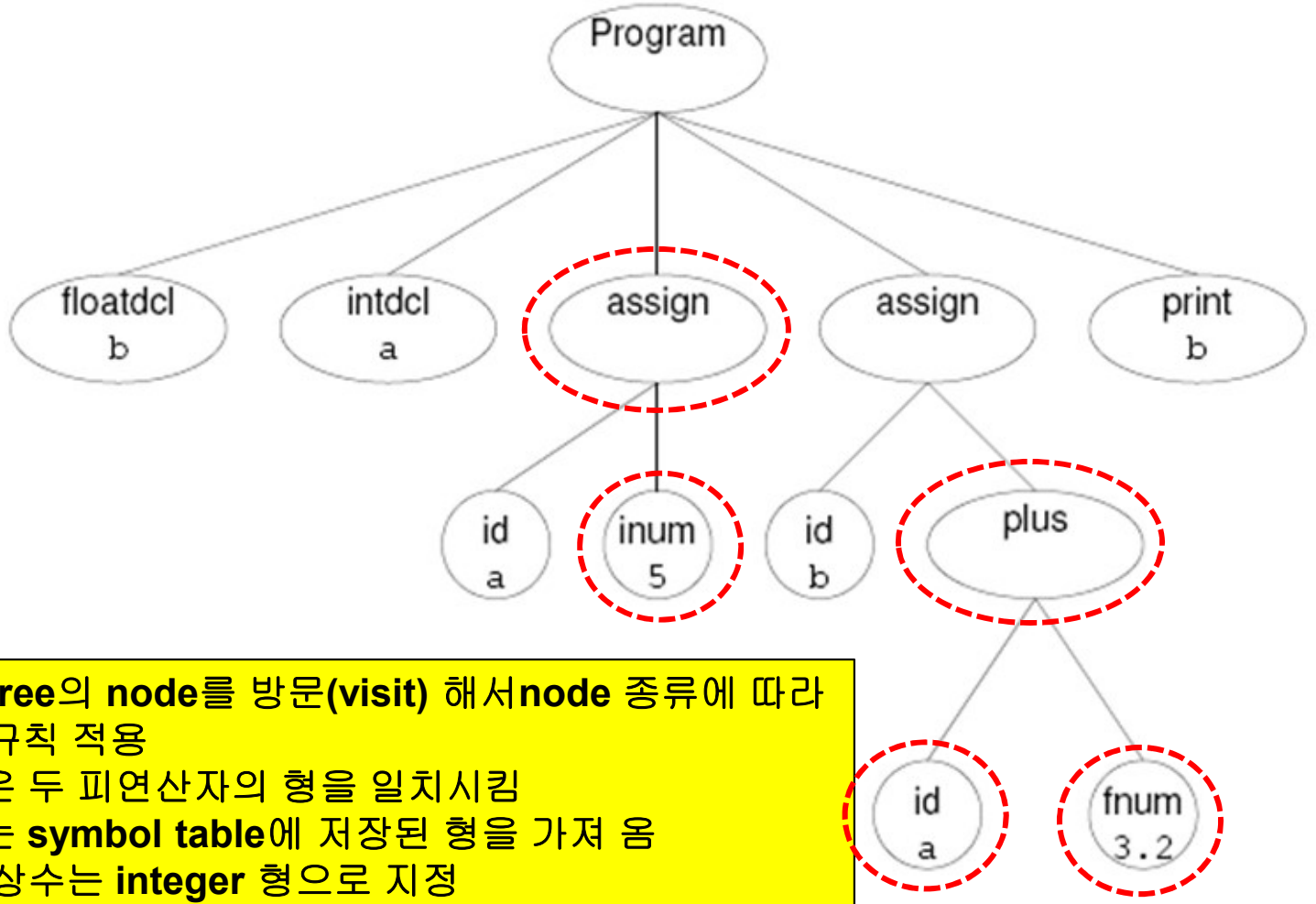
Type checking

- 모든 변수는 사용하기 전에 형을 미리 선언해야 함
  - 의미 분석 단계에서 symbol table을 구성하면 변수 형(type)을 알 수 있음
    - type checking이 가능

- **Type hierarchy (자동 형 변환 규칙)**
  - A float type is *wider* (i.e., more general) than an integer
    - Every integer can be represented as a float.
    - *Narrowing* a float to an integer loses precision for some float values.

- **Type checking**
  - Walks the AST ***bottom-up***, from its leaves toward its root.

Type analysis for ac(1/8)



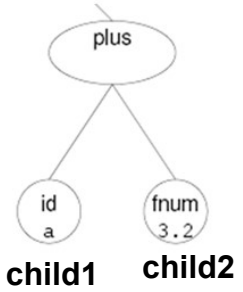
Syntax tree의 **node**를 방문**(visit)** 해서**node** 종류에 따라 형 변환 규칙 적용
→ 연산은 두 피연산자의 형을 일치시킴
→ 변수는 **symbol table**에 저장된 형을 가져 옴
→ 정수 상수는 **integer** 형으로 지정
→ 실수 상수는 **float** 형으로 지정

Type analysis for ac(2/8)

/* Visitor methods */

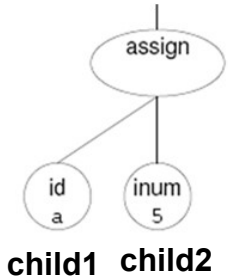
```
procedure Visit (Computing n)
    n.type ← Consistent (n.child1, n.child2)
end
```

노드가 연산 기호



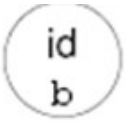

```
procedure Visit (Assigning n)
    n.type ← Convert (n.child2, n.child1.type)
end
```

노드가 할당 기호




```
procedure Visit (SymReferencing n)
    n.type ← LookupSymbol (n.id)
end
```

노드가 변수



/* Type-checking utilities */

```
function Consistent (c1, c2) returns type
function Generalize (t1, t2) returns type
procedure Convert (n, t)
```

```
procedure Visit (IntConsting n)
    n.type ← integer
end
```

노드가 정수 상수

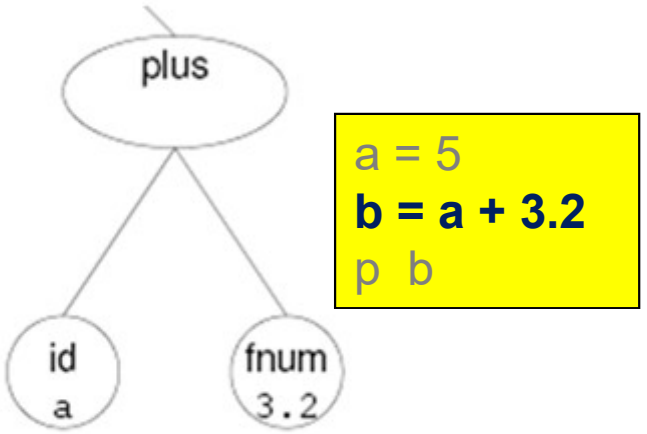

```
procedure Visit (FloatConsting n)
    n.type ← float
end
```

노드가 실수 상수

Type analysis for ac(3/8)

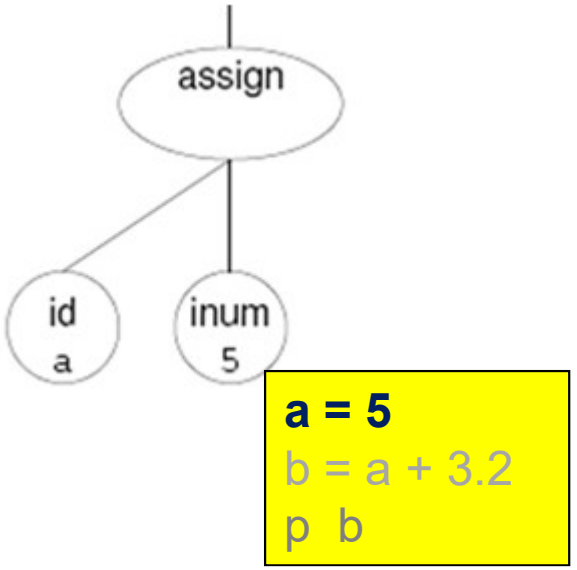
procedure Visit (*Computing n*)
** *n.type* ← Consistent (*n.child1, n.child2*)**
end

node n은 plus, child1은 a, child2는 3.2
피연산자 **n.*child2*(=3.5)**가 실수 이므로
다른 피연산자 **n.*child1*(=a)**를 실수로 변환
연산 결과 역시 실수로 저장



procedure Visit (*Assigning n*)
** *n.type* ← Convert (*n.child2, n.child1.type*)**
end

node n은 assign, child1은 a, child2는 5
n.*child2*(=5)의 **type**을 **n.*child1*(=a)**의 **type**으로 형 변환

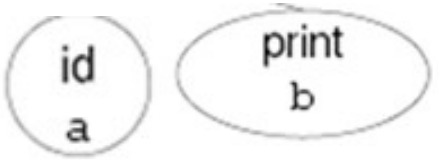


Type analysis for ac(4/8)

```
procedure Visit (SymReferencing n)
    n.type ← LookupSymbol (n.id)
end
```

```
b = a + 3.2
p b
```

symbol(변수)일 경우
symbol table 참조



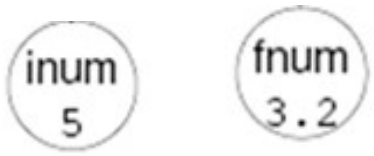
Symbol	Type
a	integer
b	float

```
procedure Visit (IntConsting n)
    n.type ← integer
end
procedure Visit (FloatConsting n)
    n.type ← float
end
```

```
a = 5
```

```
b = a + 3.2
```

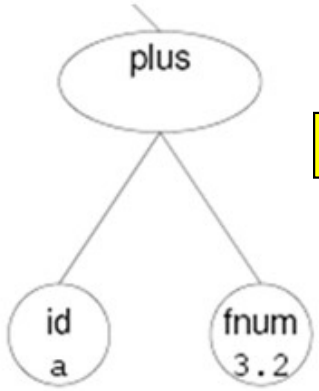
노드 값을 보고
노드 형**(type)**을 결정



Type analysis for ac(5/8)

```
procedure Visit (Computing n)
    n.type ← Consistent (n.child1, n.child2)
end
```

```
function Consistent (c1, c2) returns type
    m ← Generalize (c1.type, c2.type)
    call Convert (c1, m)
    call Convert (c2, m)
    return (m)
end
```



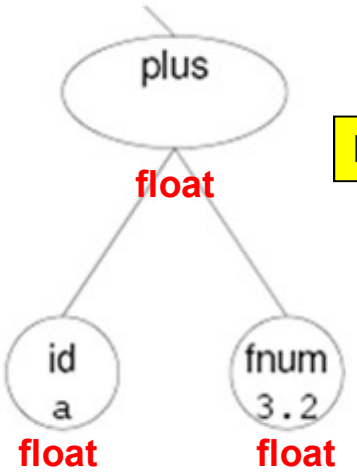
b = a + 3.2

```
function Generalize (t1, t2) returns type
    if t1 = float or t2 = float
    then ans ← float
    else ans ← integer
    return (ans)
end
```

하나라도 **float** 이면
연산 결과는 **float**

Type analysis for ac(6/8)

```
procedure Visit (Computing n)
    n.type ← Consistent (n.child1, n.child2)
end
```



b = a + 3.2

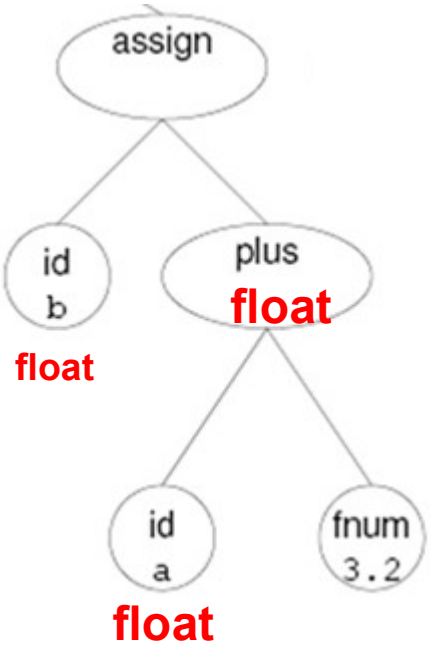
```
function Consistent (c1, c2) returns type
    m ← Generalize (c1.type, c2.type)
    call Convert (c1, m)
    call Convert (c2, m)
    return (m)
end

procedure Convert (n, t)
    if n.type = float  and  t = integer
    then call Error( "Illegal type conversion" )
    else
        if n.type = integer and t = float
        then
          /* replace node n by convert-to-float of node n */
    else  /* nothing needed  */
end
```

type n 을 type t로 변환
n (float) → *t* (integer) error!

type n 을 type t로 변환
n (integer) → *t* (float) O.K.

Type analysis for ac(7/8)

```
procedure Visit (SymReferencing n)
    n.type ← LookupSymbol (n.id)
end
```




```
f  b
i  a
a = 5
b = a + 3.2
p  b
```

Type analysis for ac(8/8)

```
procedure Visit (Assigning n)
    n.type ← Convert (n.child2, n.child1.type)
end
```

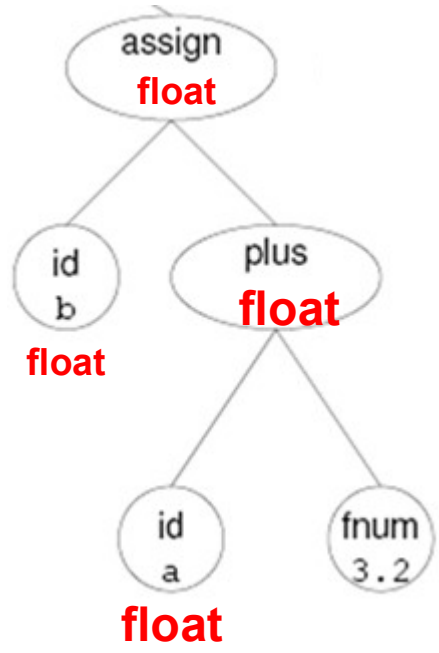
```
procedure Convert (n, t)
    if n.type = float  and  t = integer
    then call Error( "Illegal type conversion" )
    else
        if n.type = integer and t = float
        then
          /* replace node n by convert-to-float of node n */
    else  /* nothing needed  */
end
```

type n 을 type t로 변환

n (float) → *t* (integer) error!

n (integer) → *t* (float) O.K

b = a + 3.2



AST after semantic analysis