Chapter 6
Semantic Analysis

한양대학교 컴퓨터공학부
컴파일러
2014년 2학기

Overview

- also called “**static semantic analysis**”
  의미의
- involves
  - building a symbol table
    - To keep track of the meanings of names
  - performing type inference and type checking

Semantic Analysis

- Can be divided into two categories
  - Analysis of a program required by the rules of the PLs
    - To establish correctness
    - To guarantee proper execution

Ada
- Static

Pascal, C, …

LISP, Smalltalk
- Dynamically oriented lang

  - Analysis performed by a compiler
    - To enhance the efficiency of execution
    - A.K.A. “optimization”

Static semantic analysis

- description of the analyses
  - identify attributes
  - write attribute equations, or semantic rules
- implementation of the analyses
  - not clearly expressible

semantic rules
express how the computation of such attributes is related to the grammar rules of the language

like BNF for parsing

Attributes

정의:
- any property of a programming language construct
  - data type of a variable → static or dynamic
  - value of an expression → usually dynamic
  - location of a variable in memory → static or dynamic
  - obj code of a procedure → static
  - the number of significant digits in a number → static
- **binding** of the attribute
  - Process of computing an attribute and associating its computed value

static - execution 전에 attribute bind가 일어남
dynamic - execution 중에 bind가 일어남

Attribute grammars

- **Attribute grammar (semantic rules)**
  - a set of attributes and equations
  - Useful to describe the syntax-directed semantics
    semantic content of a program is closely related to its syntax
  - An attribute is computed to a nonterminal or a terminal

xj.aj = fij(x0.a1,...,x0.ak,x1.a1,...,x1.xk,....xn.a1,....xn.ak)
 xj.aj가 계산될려면 right-hand side에 있는 attribute가 존재해야 한다. (constraint)
 따라서, attribute 계산은 order가 존재 -> dependency graph

Dependency graph
 edge from xm.ak to xi.aj (depedency of xi.aj on xm.ak)
 xi.aj <- xm.ak

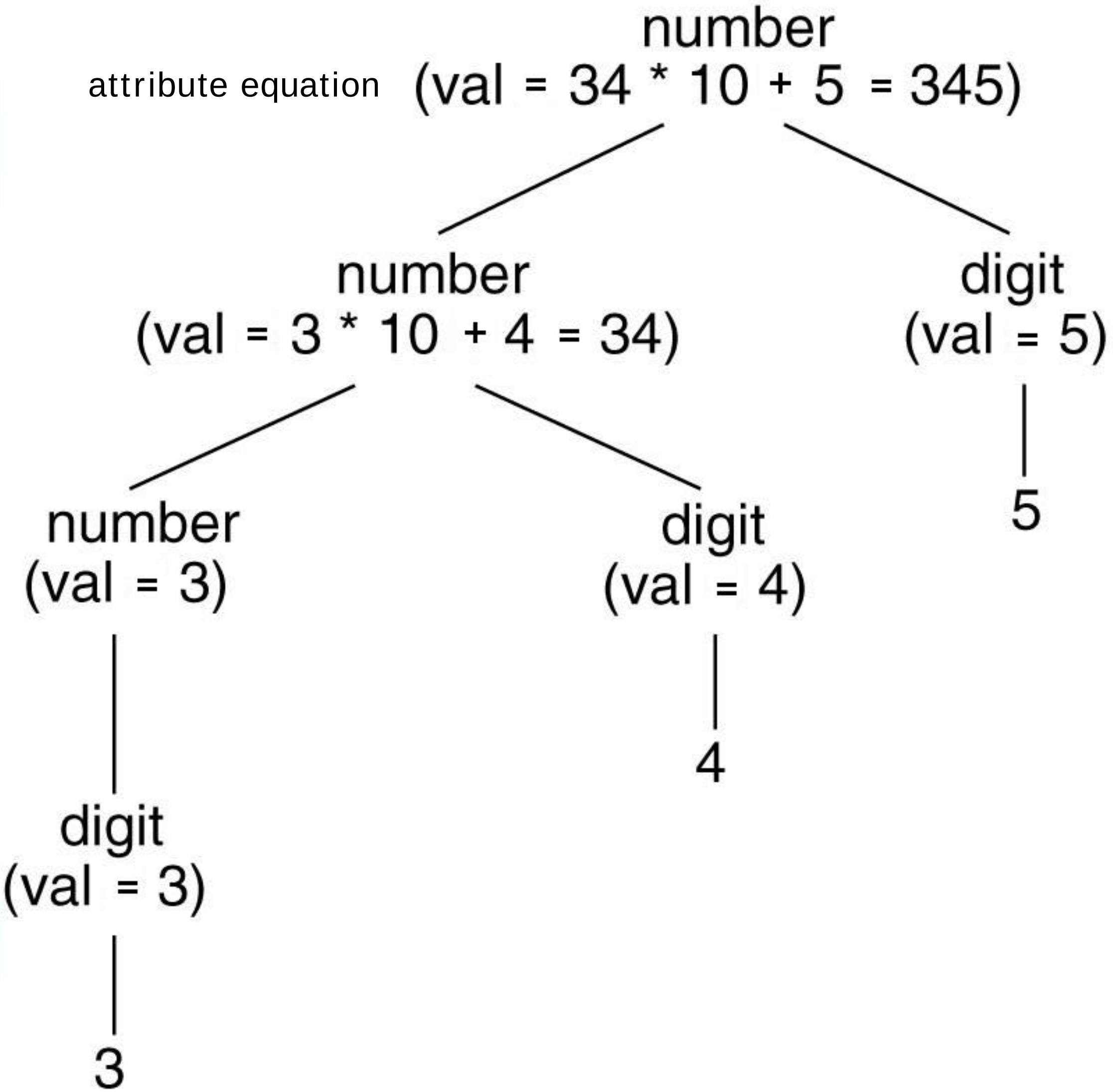
Example 6.1

- **Grammar**

$$number \rightarrow number\ digit \mid digit$$

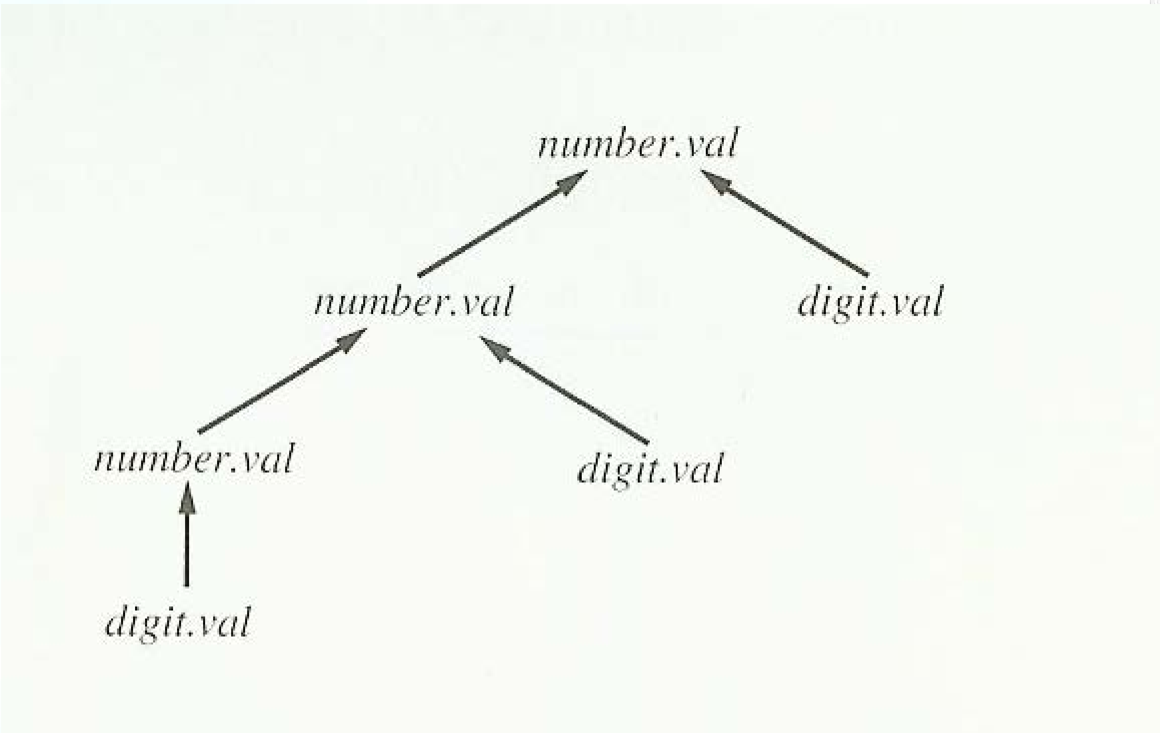
$$digit \rightarrow \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9}$$

Example 6.1


attribute equation
number
(val = 34 * 10 + 5 = 345)
number
(val = 3 * 10 + 4 = 34)
digit
(val = 5)
5
number
(val = 3)
digit
(val = 4)
4
digit
(val = 3)
3

Example 6.1

- Synthesized attribute



Example 6.1

attribute grammar

Grammar Rule	Semantic Rules
$number_1 \rightarrow number_2\ digit$	$number_1.val = number_2.val * 10 + digit.val$
$number \rightarrow digit$	$number.val = digit.val$
$digit \rightarrow \mathbf{0}$	$digit.val = 0$
$digit \rightarrow \mathbf{1}$	$digit.val = 1$
$digit \rightarrow \mathbf{2}$	$digit.val = 2$
$digit \rightarrow \mathbf{3}$	$digit.val = 3$
$digit \rightarrow \mathbf{4}$	$digit.val = 4$
$digit \rightarrow \mathbf{5}$	$digit.val = 5$
$digit \rightarrow \mathbf{6}$	$digit.val = 6$
$digit \rightarrow \mathbf{7}$	$digit.val = 7$
$digit \rightarrow \mathbf{8}$	$digit.val = 8$
$digit \rightarrow \mathbf{9}$	$digit.val = 9$

Example 6.2

- **Grammar**

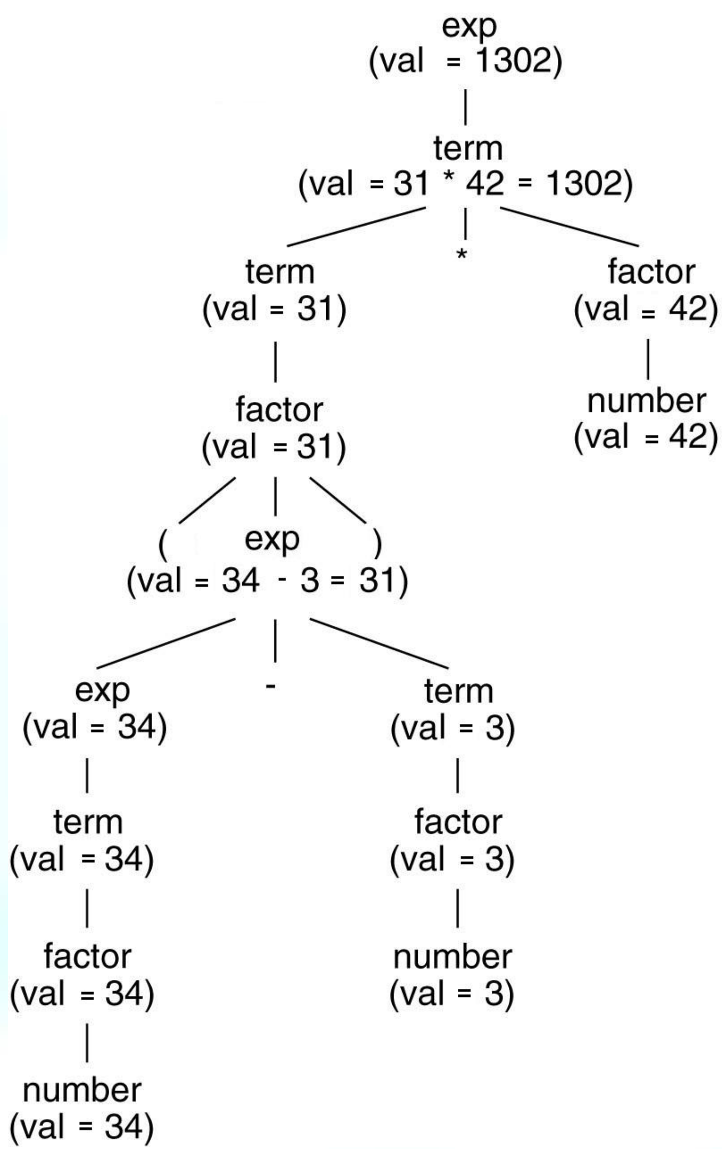
$$exp \rightarrow exp \ \mathbf{+} \ term \mid exp \ \mathbf{-} \ term \mid term$$

$$term \rightarrow term \ \mathbf{*} \ factor \mid factor$$

$$factor \rightarrow \mathbf{(} \ exp \ \mathbf{)} \mid \boldsymbol{number}$$

Example 6.2

(34 - 3)*42



Example 6.2

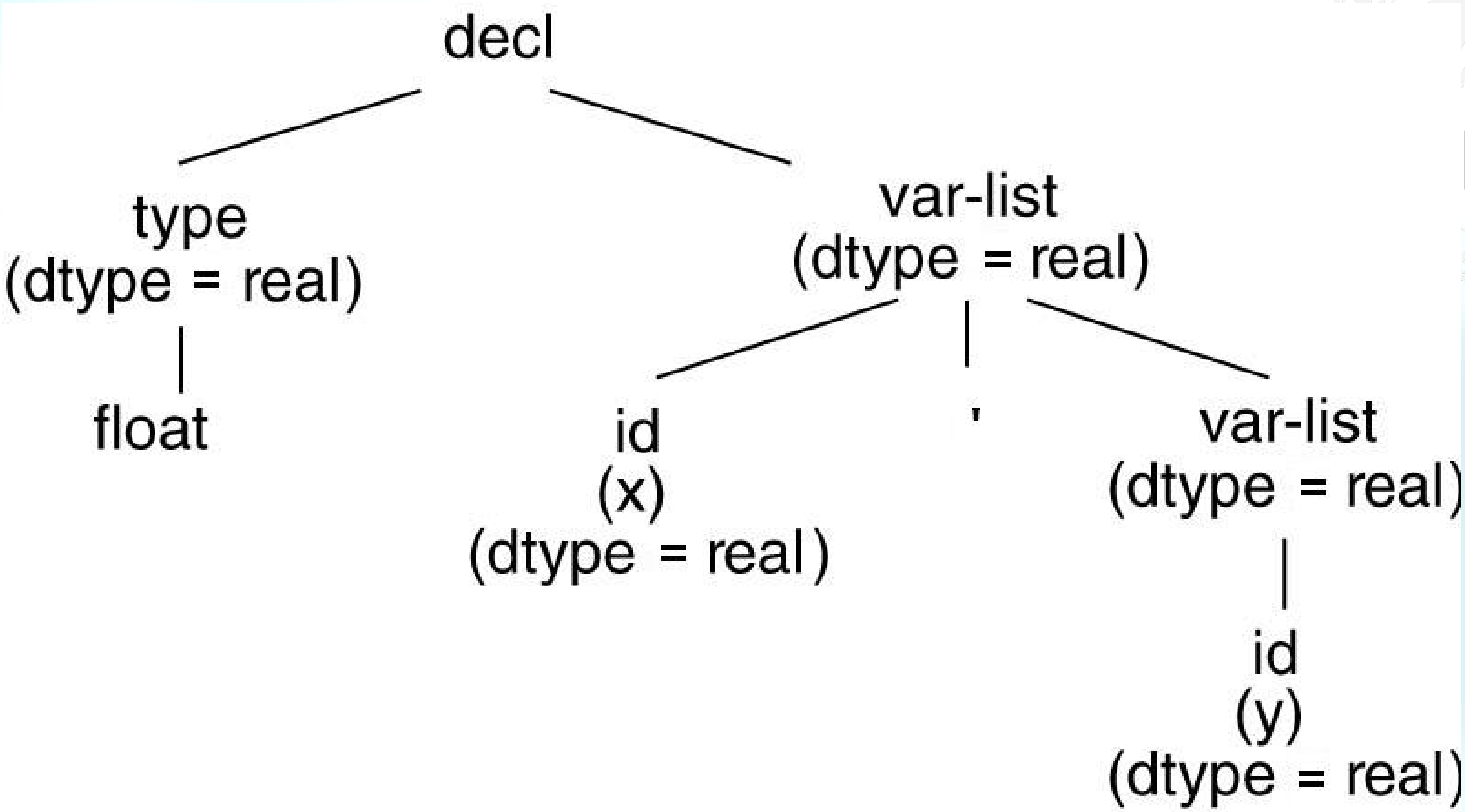
Grammar Rule	Semantic Rules
$exp_1 \rightarrow exp_2$ + $term$	$exp_1.val = exp_2.val + term.val$
$exp_1 \rightarrow exp_2$ - $term$	$exp_1.val = exp_2.val - term.val$
$exp \rightarrow term$	$exp.val = term.val$
$term_1 \rightarrow term_2$ * $factor$	$term_1.val = term_2.val * factor.val$
$term \rightarrow factor$	$term.val = factor.val$
$factor \rightarrow$ **(** exp **)**	$factor.val = exp.val$
$factor \rightarrow$ **number**	$factor.val =$ **number**.val

Example 6.3

- **Grammar**

$$\textit{decl} \rightarrow \textit{type var-list}$$
$$\textit{type} \rightarrow \textbf{int} \mid \textbf{float}$$
$$\textit{var-list} \rightarrow \textbf{\textit{id}}\textbf{,}\ \textit{var-list} \mid \textbf{\textit{id}}$$

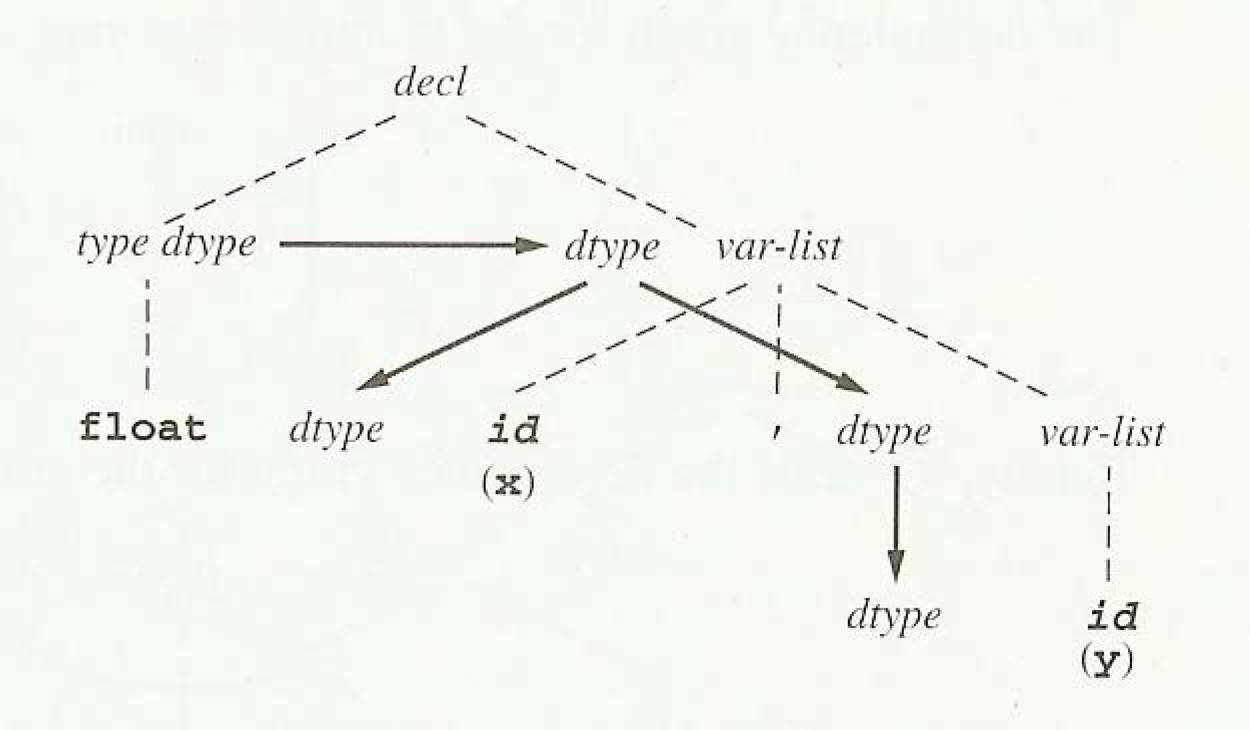
Example 6.3



decl
type
(dtype = real)
float
var-list
(dtype = real)
id
(x)
(dtype = real)
,
var-list
(dtype = real)
id
(y)
(dtype = real)

Example 6.3

- Inherited attribute



Example 6.3

Grammar Rule	Semantic Rules
$decl \rightarrow type\ val\text{-}list$	$var\text{-}list.dtype = type.dtype$
$type \rightarrow$ **int**	$type.dtype = integer$
$type \rightarrow$ **float**	$type.dtype = real$
$var\text{-}list_1 \rightarrow$ ***id,*** $var\text{-}list_2$	***id***$.dtype = var\text{-}list_1.dtype$
	$var\text{-}list_2.dtype = var\text{-}list_1.dtype$
$var\text{-}list \rightarrow$ ***id***	***id***$.dtype = var\text{-}list.dtype$

Example 6.4

- **Grammar**

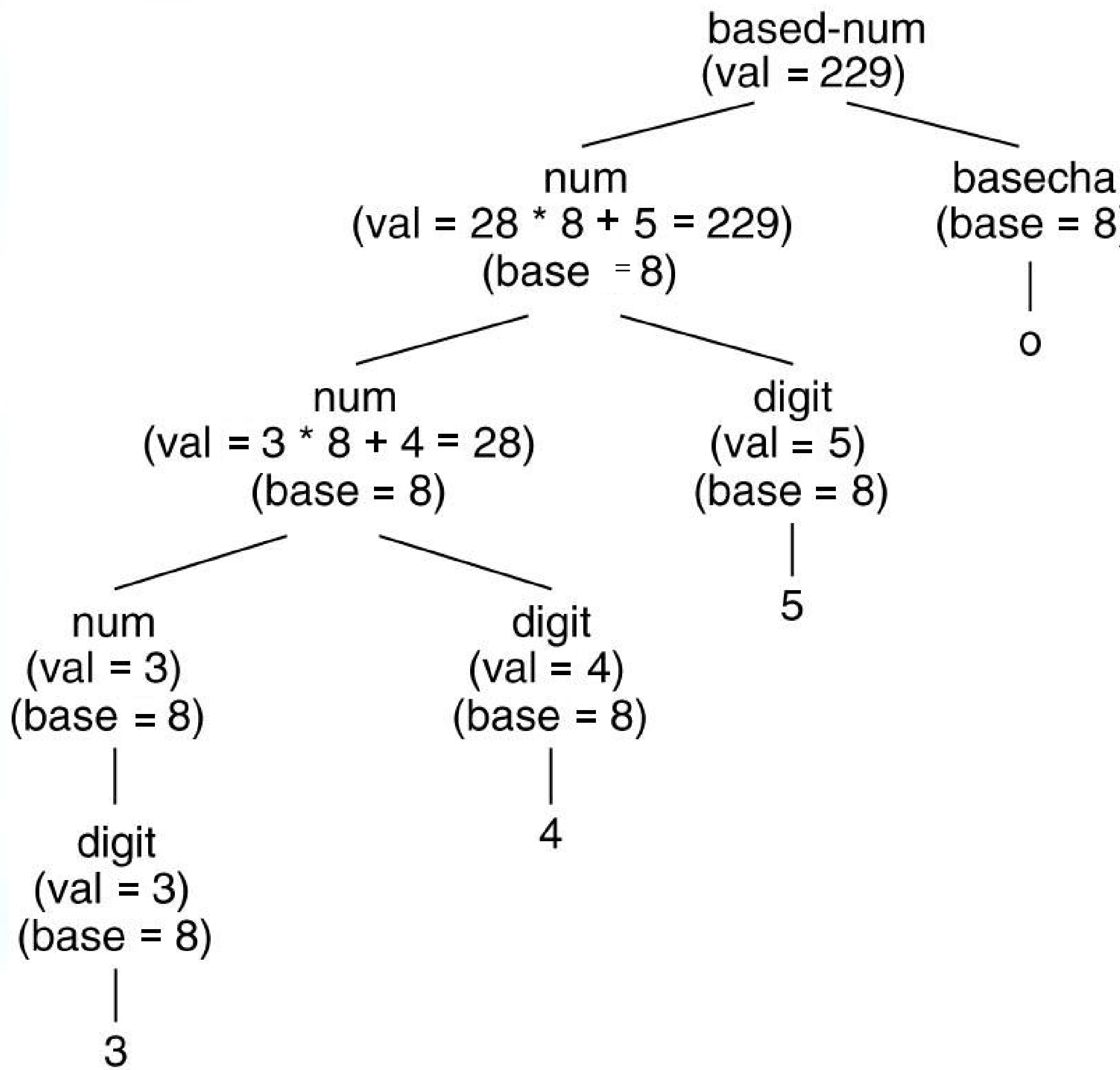
based-num → *num basechar*

basechar → **o** | **d**

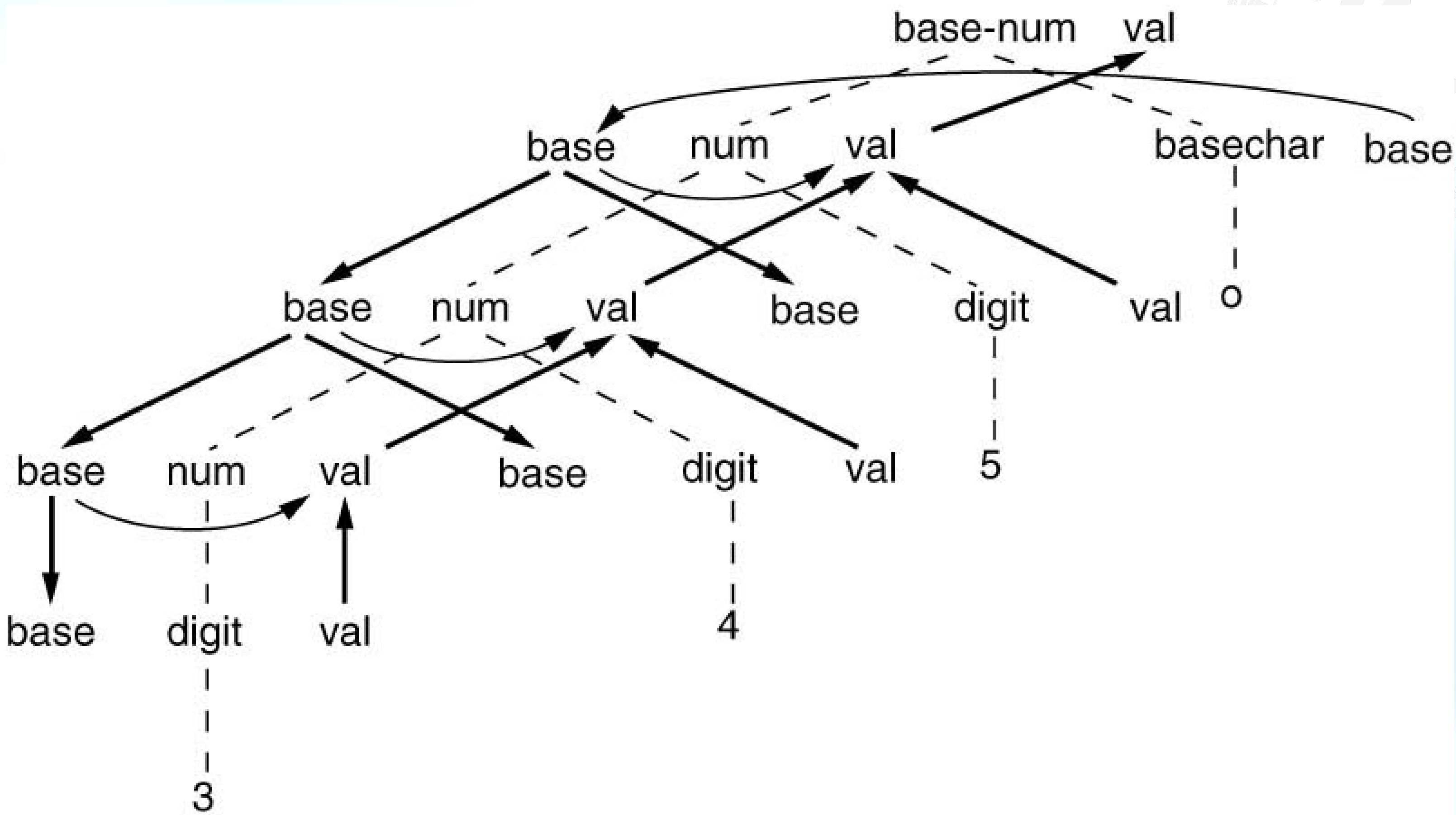
num → *num digit* | *digit*

digit → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Example 6.4


based-num
(val = 229)
num
(val = 28 * 8 + 5 = 229)
(base = 8)
basechar
(base = 8)
o
num
(val = 3 * 8 + 4 = 28)
(base = 8)
digit
(val = 5)
(base = 8)
5
num
(val = 3)
(base = 8)
digit
(val = 4)
(base = 8)
4
digit
(val = 3)
(base = 8)
3

Example 6.4

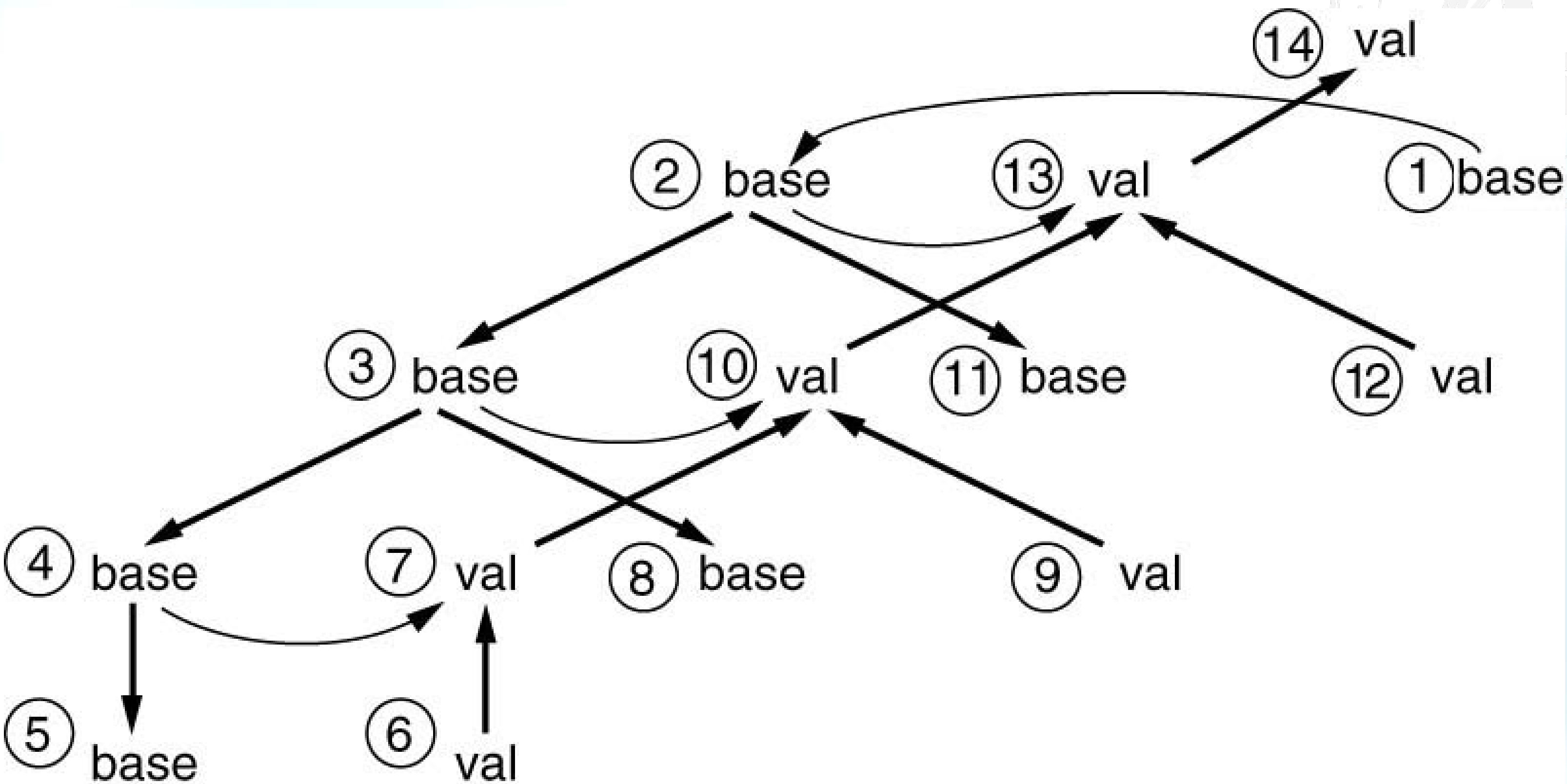


base-num val
base num val basechar base
base num val base digit val o
base num val base digit val 5
base digit val 4
3

Example 6.4

traversal order of the dependency graph = topological sort (DAGs(directed acyclic graphs))

root(predecessor가 없는 node) 의 경우 해당 정보로 (다른 attribute 정보없이) attribute가 계산 가능해야 한다. (ex 1,6,9,12 - root)



Example 6.4

Grammar Rule	Semantic Rules
based-num → *num basechar*	*based-num.val* = *num.val* *num.base* = *basechar.base*
basechar → **o**	*basechar.base* = 8
basechar → **d**	*basechar.base* = 10
$num_1 \rightarrow num_2\ digit$	$num_1.val$ = **if** *digit.val* = *error* **or** $num_2.val$ = *error* **then** *error* **else** $num_2.val * num_1.base + digit.val$ $num_2.base = num_1.base$ $digit.base = num_1.base$
num → *digit*	*num.val* = *digit.val* *digit.base* = *num.base*
digit → **0**	*digit.val* = 0
digit → **1**	*digit.val* = 1
. . .	. . .
digit → **7**	*digit.val* = 7
digit → **8**	*digit.val* = **if** *digit.base* = 8 **then** *error* **else** 8
digit → **9**	*digit.val* = **if** *digit.base* = 8 **then** *error* **else** 9

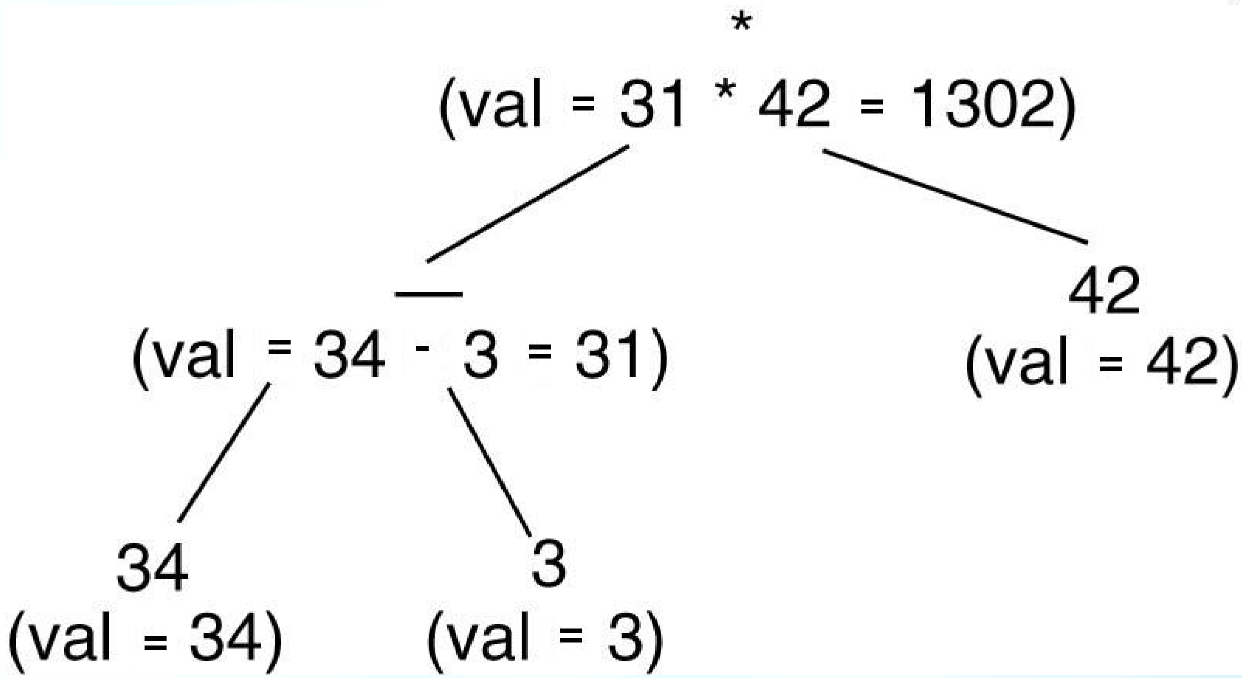
metalanguage
- the collection of expressions allowable in an attribute equation

Use of ambiguous grammar

parse에서 ambiguous가 해결되었기 때문에 semantic(attribute grammar)에서는 ambiguous에서 자유롭다.

Grammar Rule	Semantic Rules
$exp_1 \rightarrow exp_2 + exp_3$	$exp_1.val = exp_2.val + exp_3.val$
$exp_1 \rightarrow exp_2 - exp_3$	$exp_1.val = exp_2.val - exp_3.val$
$exp_1 \rightarrow exp_2 * exp_3$	$exp_1.val = exp_2.val * exp_3.val$
$exp_1 \rightarrow (exp_2)$	$exp_1.val = exp_2.val$
$exp \rightarrow \textbf{number}$	$exp.val = \textbf{number}.val$

Displaying attributes on a syntax tree



Syntax tree as an attribute

Grammar Rule	Semantic Rules
$exp_1 \rightarrow exp_2 + term$	$exp_1.tree =$ $mkOpNode\ (+, exp_2\ .tree, term.tree)$ use of function
$exp_1 \rightarrow exp_2 - term$	$exp_1.tree =$ $mkOpNode\ (-, exp_2\ .tree, term.tree)$
$exp \rightarrow term$	$exp.tree = term.tree$
$term_1 \rightarrow term_2 * factor$	$term_1.tree =$ $mkOpNode\ (*, term_2\ .tree, factor.tree)$
$term \rightarrow factor$	$term.tree = factor.tree$
$factor \rightarrow$ **(** exp **)**	$factor.tree = exp.tree$
$factor \rightarrow$ ***number***	$factor.tree =$ $mkNumNode$ (***number***$.lexval$)

extensions to attribute grammar
1. metalanguage
2. function

Synthesized and Inherited Attributes

- **synthesized** attributes A -> x1x2...xn
  A.a = f(x1.a1,...x1.ak,...xn.a1,...xn.ak)
  - An attribute is **synthesized** if all its dependencies point from child to parent in the parse tree.

S-attributed grammar - all attributes are synthesized

procedure *PostEval* (*T* : *treenode*);
begin
    for *each child C of T* do
        PostEval (*C*);
    compute all synthesized attributes of T ;
end;

attribute rules of an S-attributed grammar can be computed by a single bottom-up or postorder traversal of the tree

- **inherited** attributes
  - An attribute that is no synthesized is called an **inherited** attribute

can be computed by a preorder or combined preorder/inorder

Example 6.11

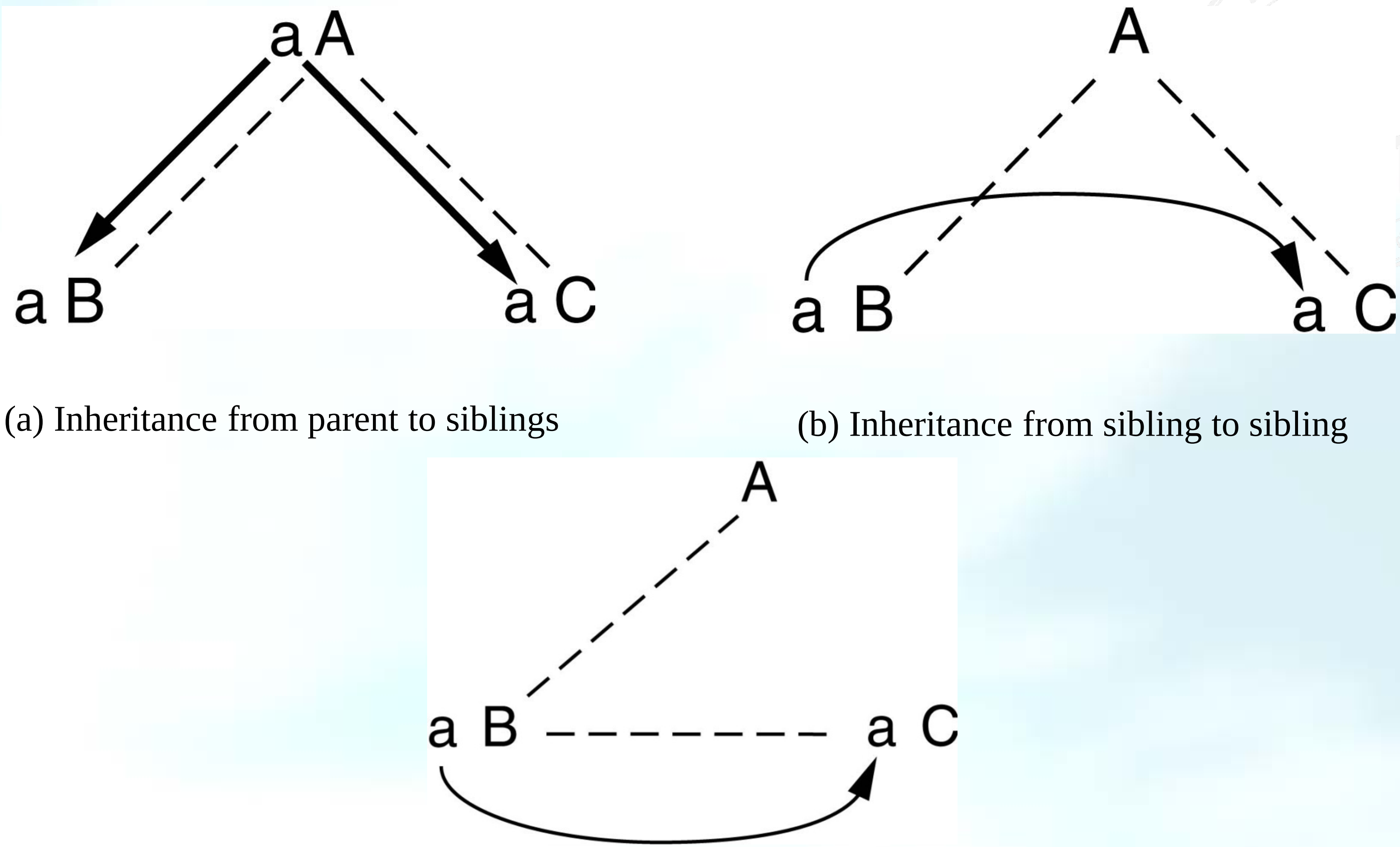
- Structure

```
typedef enum {Plus, Minus, Times} OpKind;
typedef enum {OpKind, ConstKind} ExpKind;
typedef struct streenode
            { ExpKind kind;
              OpKind op;
              struct streenode *lchild, *rchild;
               int val;
            } STreeNode;
typedef STreeNode *SyntaxTree;
```

Figure 6.8


```
void postEval (SyntaxTree t)
{  int temp;
   if (t->kind == OpKind)
  {  postEval(t->lchild);
     postEval(t->rchild);
     switch (t->op)
    {  case Plus:
           t->val = t->lchild->val + t->rchild->val;
           break;
       case Minus:
           t->val = t->lchild->val - t->rchild->val;
           break;
       case Times:
           t->val = t->lchild->val * t->rchild->val;
           break;
    } /* end switch */
  } /* end if */
} /* end postEval */
```

Figure 6.9



(a) Inheritance from parent to siblings

(b) Inheritance from sibling to sibling

(c) sibling inheritance via sibling pointers

```
procedure preEval (T : treenode);
begin
    for each child C of T do
        compute all inherited attributes of C ;
        PreEval (C );
end;
```

Example 6.12

- **Grammar**

$$decl \rightarrow type\ var\text{-}list$$

$$type \rightarrow \textbf{int} \mid \textbf{float}$$

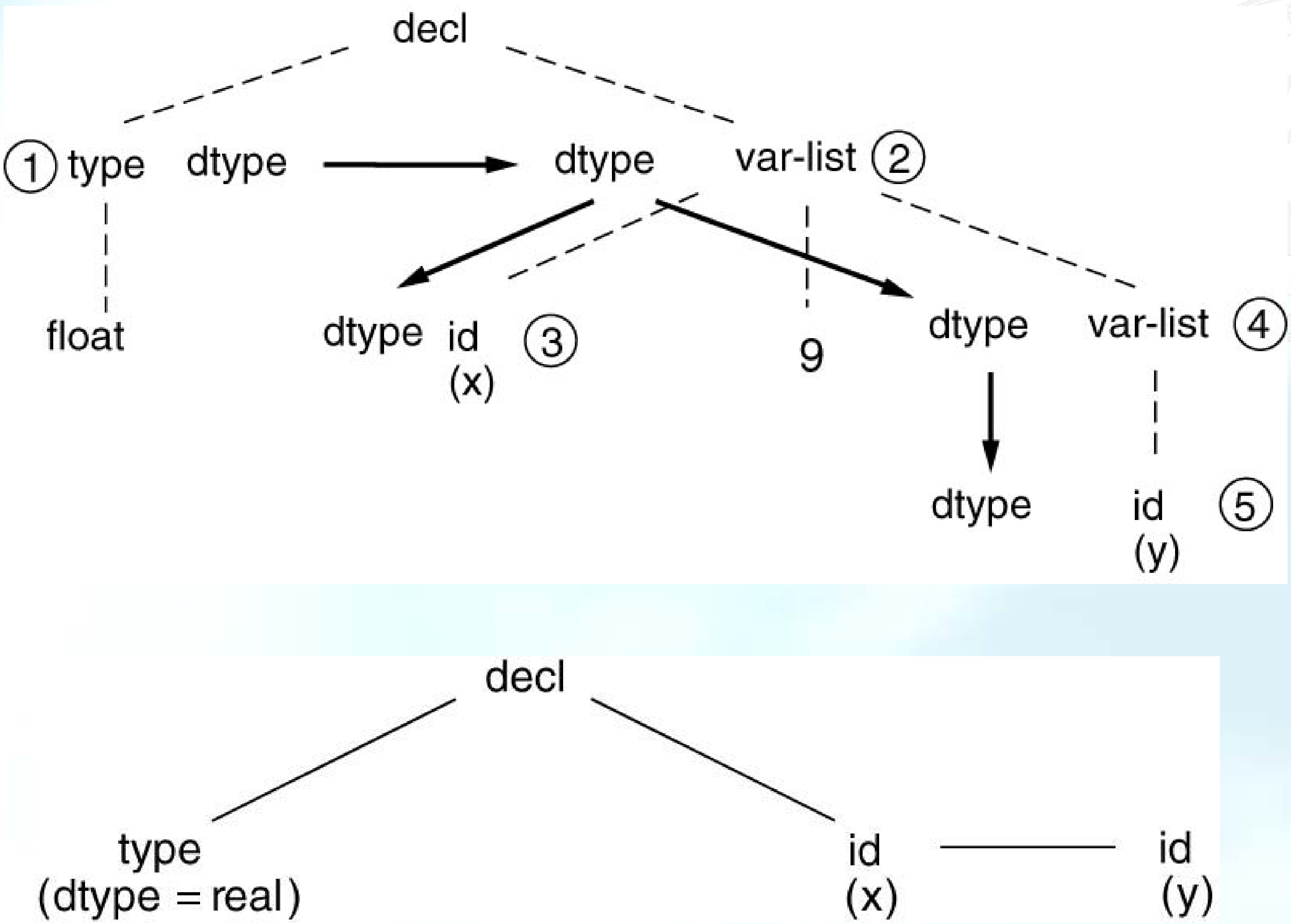
$$var\text{-}list \rightarrow \textbf{\textit{id}}\textbf{,}\ var\text{-}list \mid \textbf{\textit{id}}$$

Example 6.12

- **Pseudocode**

```
procedure EvalType ( T: treenode );
begin
  case nodekind of T of
  decl:
      EvalType(type child of T);
      Assign dtype of type child of T to var-list child of T;     inorder process
      EvalType(var-list child of T);
  type:
      if child of T = int then T.dtype := integer
      else T.dtype := real;
  var-list:
      assign T.dtype to first child of T;
      if third child of T is not nil then
         assign T.dtype to third child;                          preorder process
         EvalType(third child of T);
  end case;
end EvalType;
```

Figure 6.10



decl
1 type
dtype
dtype
var-list 2
float
dtype
id
(x)
3
9
dtype
var-list 4
dtype
id
(y)
5
decl
type
(dtype = real)
id
(x)
id
(y)

C code of Example 6.12


```
typedef enum {decl, type, id} nodekind;
typedef enum {integer, real} typekind;
typedef struct treeNode
   { nodekind kind;
      struct treeNode
         *lchild, *rchild, *sibling;
      typekind dtype;
      /* for type and id nodes */
      char *name;
      /* for id nodes only */
   } *SyntaxTree;
```

C code of Example 6.12


```
void evalType (SyntaxTree t)
{  switch (t->kind)
   {  case decl:
         t->rchild->dtype = t->lchild->dtype;
         evalType(t->rchild);
         break;
      case id:
         if (t->sibling != NULL)
         {  t->sibling->dtype = t->dtype;
            evalType(t->sibling);
         }
         break;
   }  /* end switch */
}  /* end evalType */
```

C code of Example 6.12


```
void evalType (SyntaxTree t)
{  if (t->kind == decl)
   {  SyntaxTree p = t->rchild;
      p->dtype = t->lchild->dtype;
      while (p->sibling != NULL)
      {  p->sibling->dtype = p->dtype;
         p = p->sibling;
      }
   }  /* end if */
}  /* end evalType */
```

Example 6.13

- **Grammar**

val - synthesized attribute
base - inherited attribute

based-num → *num basechar*

basechar → **o** | **d**

num → *num digit* | *digit*

digit → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Example 6.13

- **Pseudocode**

```
procedure EvalWithBase ( T: treenode );
begin
  case nodekind of T of
  based-num:
      EvalWithBase(right child of T);
      assign base of right child of T to base of left child;
      EvalWithBase(left child of T);
      assign val of left child of T to T.val;
  num:
      assign T.base to base of left child of T;
      EvalWithBase(left child of T);
      if right child of T is not nil then
       assign T.base to base of right child of T;
        EvalWithBase(right child of T);
         if vals of left and right children ≠ error then
             T.val := T.base*(val of left child) + val of right child
         else T.val := error;
      else T.val := val of left child;
  basechar:
      if child of T = o then T.base := 8
      else T.base :=10;
  digit:
      if T.base = 8 and child of T = 8 or 9 then T.val :=
       error
      else T.val := numval(child of T);
  end case;
end EvalWithBase;
```

<combinations of synthesized and inherited attribute>
if the synthesized attributes depend on the inherited attributed but the inherited attributes do not depend on any synthesized attribute

procedure *CombineEval* (*T* : *treenode*);
begin
    for *each child C of T* do
        compute all inherited attributes of C ;
        CombineEval (*C*);
    compute all synthesized attributes of T;
end;

example 6.14

- Grammar

  inherited attribute depends on synthesized attribute

  $$exp \rightarrow exp\ /\ exp \mid \underbrace{num}_{\text{integer}} \mid \underbrace{num.num}_{\text{floating-point}}$$

- Operations may be interpreted differently
- Three attributes
  - A synthesized Boolean attribute *isFloat* indicates if any part of an expression has a floating-point value
  - An inherited attribute *etype* int or float
  - A synthesized attribute *val*

depend on

exp 중에서 float 한개라도 있으면 전부 float 취급

table 6.7

Grammar Rule	Semantic Rules
$S \rightarrow exp$	$exp.etypr =$ **if** $exp.isFloat$ **then** $float$ **else** int $S.val = exp.val$
$exp_1 \rightarrow exp_2 \,/\, exp_3$	$exp_1.isFloat =$ $exp_2.isFloat$ **or** $exp_3.isFloat$ $exp_2.etype = exp_1.etype$ $exp_3.etype = exp_1.etype$ $exp_1.val =$ **if** $exp_1.etype = int$ **then** $exp_2.val$ **div** $exp_3.val$ **else** $exp_2.val \,/\, exp_3.val$
$exp \rightarrow$ ***num***	$exp.isFloat =$ **false** $exp.val =$ **if** $exp.etype = int$ **then** ***num***$.val$ **else** $Float($***num***$.val)$
$exp \rightarrow$ ***num.num***	$exp.isFloat =$ **true** $exp.val =$ ***num.num***$.val$

div - integer division
/ - floating divison

converts the integer value num into floating-point value

Notes

- The attributes in this example can be computed by two passes
  - First pass: computes the synthesized attribute *isFloat* by a postorder traversal
  - Second pass: computes both the inherited attributed *etype* and the synthesized attribute *val* in a combined preorder/posorder traversal