Chapter 6
Semantic Analysis

2022 Spring&Summer

Outline

- Attributes and attribute grammars
- Algorithms for attribute computation
- The symbol table
- Data types and type checking
- A semantic analyzer for the TINY language

6.1 Attributes and attribute grammars

- Attributes: any property of a programming language construct such as
  - The data type of a variable
  - The value of an expression
  - The location of a variable in memory
  - The object code of a procedure
  - The number of significant digits in a number
- Attributes may be fixed prior to the compilation process.
- Attributes may be only determinable during program execution.

6.1 Attributes and attribute grammars

- Binding of the attribute: the process of computing an attribute and associating its computed value with the language construct in question.
- Binding time: the time during the compilation/ execution process when the binding of an attribute occurs.

6.1 Attributes and attribute grammars

- Binding times of different attributes vary, and even the same attributes that can have quite different binding times from language to language.
- Static attributes/Dynamic attributes: Based on the difference of the binding time
    - be bound prior to execution
    - be bound during execution

6.1 Attributes and attribute grammars

- A type checker: is an important part of semantic analysis (in a language like C or Pascal)
- A type checker is an analyzer
  - Computes the data type attribute of all language entities for which data types are defined
  - Verifies that these types conform to the type rules of the language.

6.1 Attributes and attribute grammars

- Type checking = set of rules that ensure the type consistency of different constructs in the program

- Examples:
  - The type of a variable must match the type from its declaration
  - The operands of arithmetic expressions (+, *, -, /) must have integer types; the result has integer type
  - The operands of comparison expressions (==, !=) must have integer or string types; the result has boolean type

6.1 Attributes and attribute grammars

More examples:

- For each assignment statement, the type of the updated variable must match the type of the expression being assigned
- For each call statement foo(v1, …, vn), the type of each actual argument vi must match the type of the corresponding formal argument fi from the declaration of function foo
- The type of the return value must match the return type from the declaration of the function

6.1.1 Attribute grammars

- $X.a$: the value of a associated to X

  X is a grammar symbol and a is an attribute associated to X.
- Syntax-directed semantics: attributes are associated directly with the grammar symbols of the language.
- Given attributes $a_1, a_2, \ldots, a_k$, for each grammar rule $X_0 \rightarrow X_1\, X_2 \ldots X_n$ (X_0 is a nonterminal), the values of the attributes $X_i.a_j$ of each grammar symbol X_i are related to the values of the attributes of the other symbols in the rule.

6.1.1 Attribute grammars

- An attribute grammar for attributes $a_1, a_2, \ldots, a_k$ is the collection of all attribute equations or semantic rules of the following form, for all the grammar rules of the language.

$$X_i.a_j = f_{ij}(X_0.a_1, \ldots, X_0.a_k, X_1.a_1, \ldots, X_1.a_k, \ldots X_n.a_1, \ldots X_n.a_k)$$

f_{ij} is a mathematical function of its arguments

6.1.1 Attribute grammars

- Attribute grammars are written in tabular form as follows:

Grammar Rule	Semantic Rules
Rule 1	Associated attribute equations
…..	……….
………..	………….
Rule n	Associated attribute equations

6.1.1 Attribute grammars

Example 6.1:

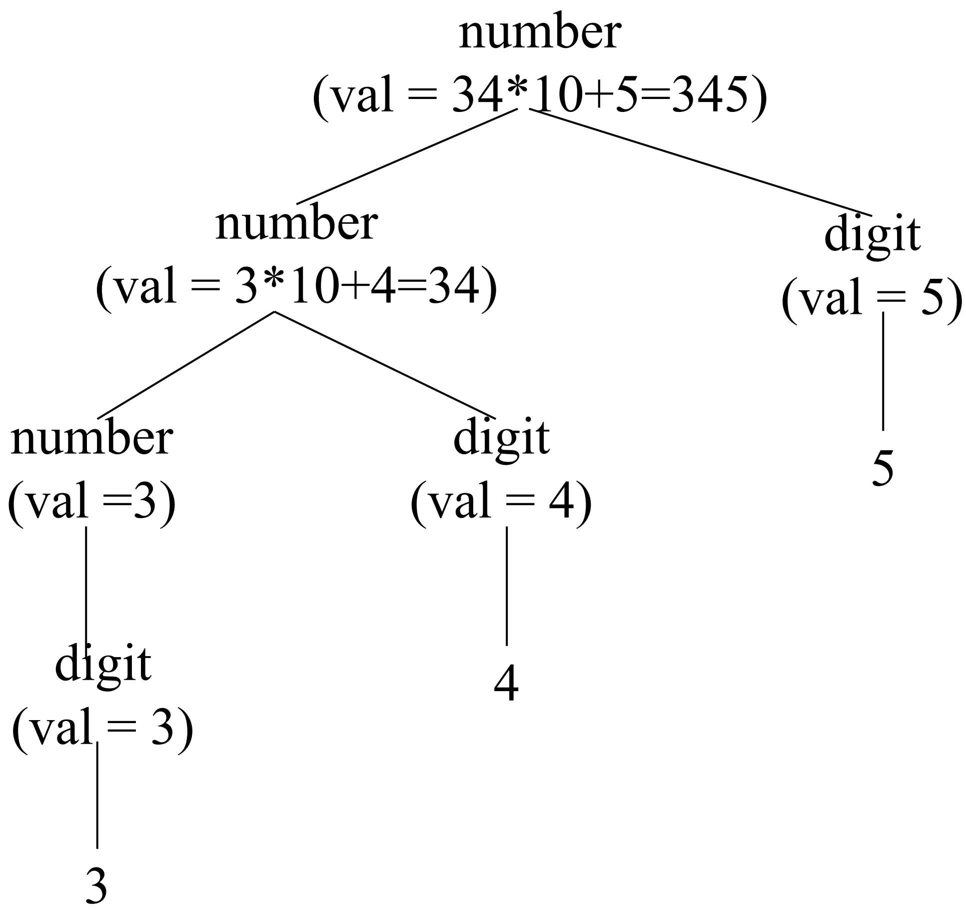
number $\rightarrow$ *number digit* | *digit*

digit $\rightarrow$ *0|1|2|3|4|5|6|7|8|9*

Grammar Rule	Semantic Rules
$number1 \rightarrow number2\ \ digit$	$number1.val = number2.val*10+digit.val$
$number \rightarrow digit$	$number.val = digit.val$
$digit \rightarrow 0$	$digit.val = 0$
$digit \rightarrow 1$	$digit.val = 1$
$digit \rightarrow 2$	$digit.val = 2$
$digit \rightarrow 3$	$digit.val = 3$
$digit \rightarrow 4$	$digit.val = 4$
$digit \rightarrow 5$	$digit.val = 5$
$digit \rightarrow 6$	$digit.val = 6$
$digit \rightarrow 7$	$digit.val = 7$
$digit \rightarrow 8$	$digit.val = 8$
$digit \rightarrow 9$	$digit.val = 9$

6.1.1 Attribute grammars

The parse tree showing attribute computations for the number *345* is given as follows



6.1.1 Attribute grammars

Example 6.2 :

exp → *exp* + *term* | *exp-term*|*term*

term → *term***factor* | *factor*

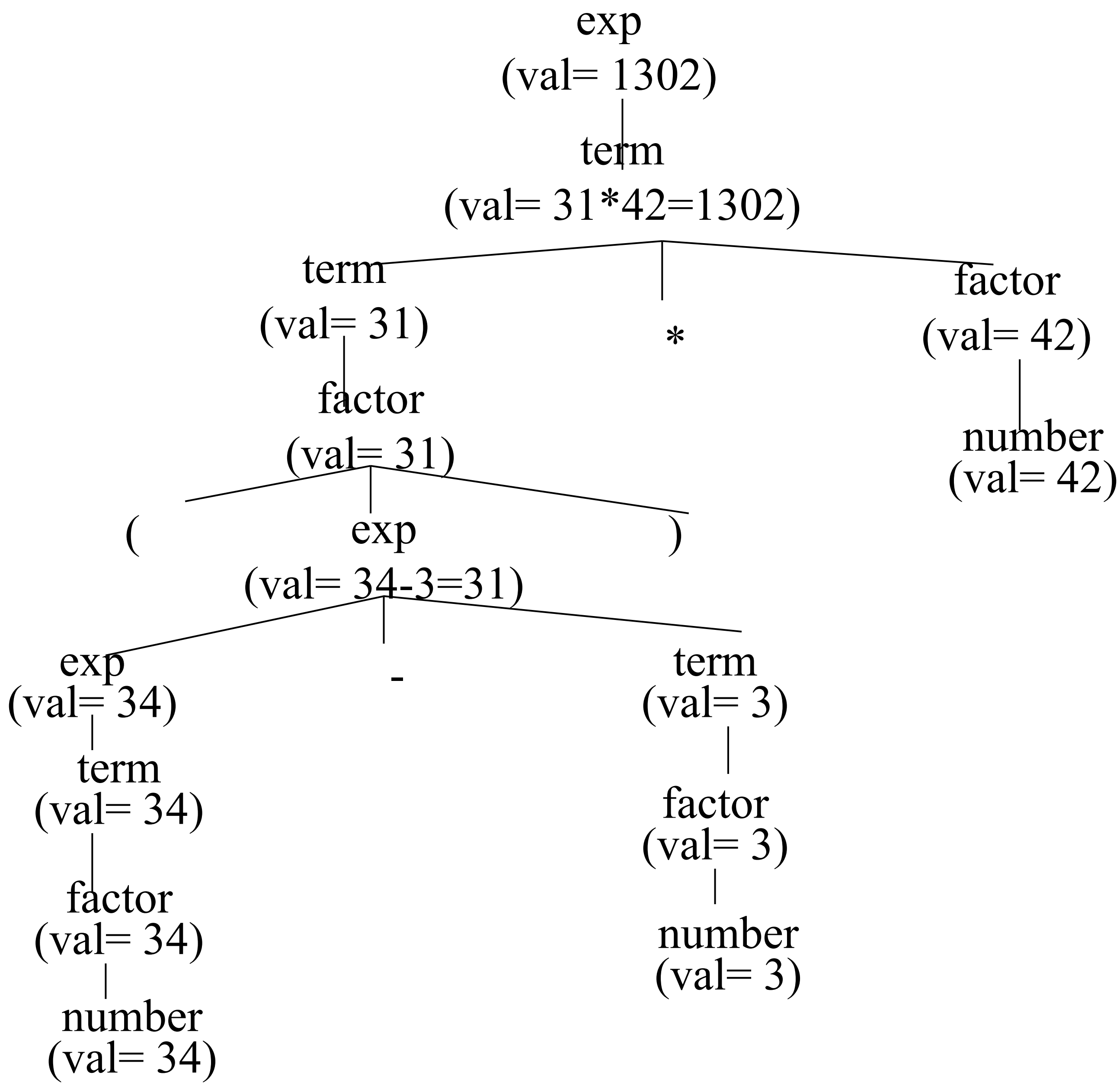
factor → (*exp*)|*number*

grammar Rule	semantic Rules
$exp1 \rightarrow exp2 + term$	$exp1.val = exp2.val + term.val$
$exp1 \rightarrow exp2 - term$	$exp1.val = exp2.val - term.val$
$exp1 \rightarrow term$	$exp1.val = term.val$
$term1 \rightarrow term2 * factor$	$term1.val = term2.val * factor.val$
$term \rightarrow factor$	$term.val = factor.val$
$factor \rightarrow (exp)$	$factor.val = exp.val$
$factor \rightarrow number$	$factor.val = number.val$

6.1.1 Attribute grammars

The computations implied by this attribute grammar by attaching equations to nodes in a parse tree is as follows. (Given the expression *(34-3)*42*)

6.1.1 Attribute grammars



6.1.1 Attribute grammars

Example 6.3 :

decl → *type var-list*

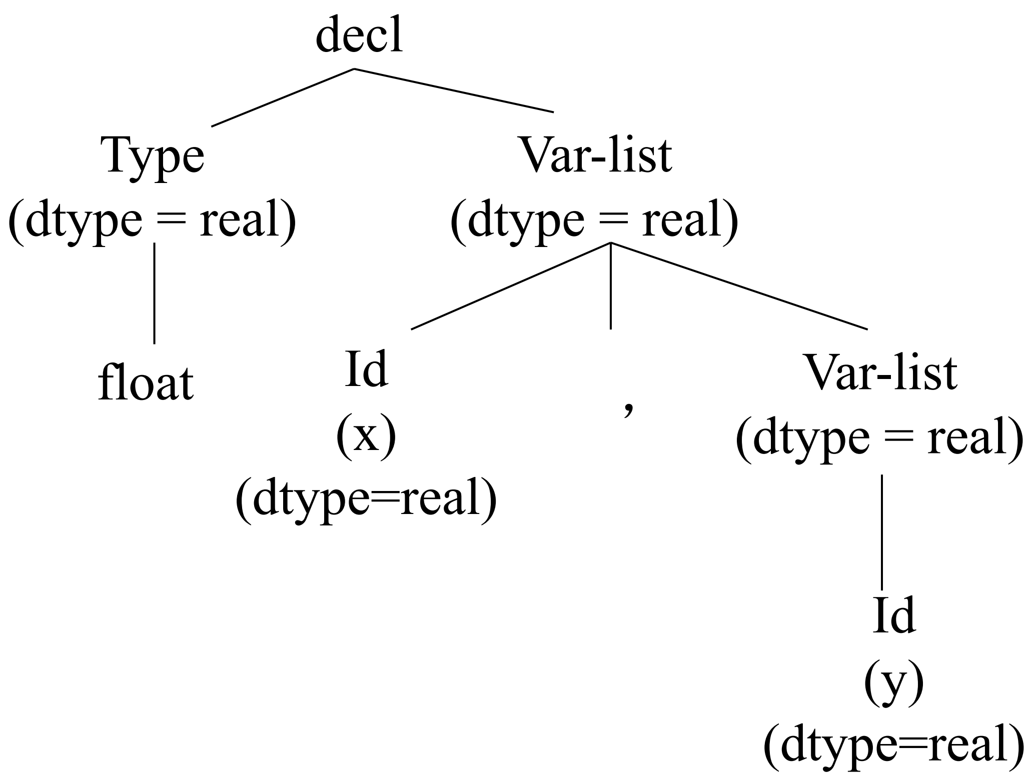
type → *int* | *float*

var-list → *id*, *var-list* | *id*

grammar Rule	semantic Rules
decl → *type var-list*	*var-list.dtype = type.dtype*
type → *int*	*type.dtype = integer*
type → *float*	*type.dtype = real*
var-list1 → *id, var-list2*	*id.dtype = var-list1.dtype* *var-list2.dtype = var-list1.dtype*
var-list → *id*	*id.dtype = var-list.dtype*

6.1.1 Attribute grammars

Parse tree for the string *float x,y* showing the *dtype* attribute as specified by the attribute grammar above is as follows



6.1.1 Attribute grammars

Example 6.4

based-num → *num basechar*

basechar → *o* | *d*

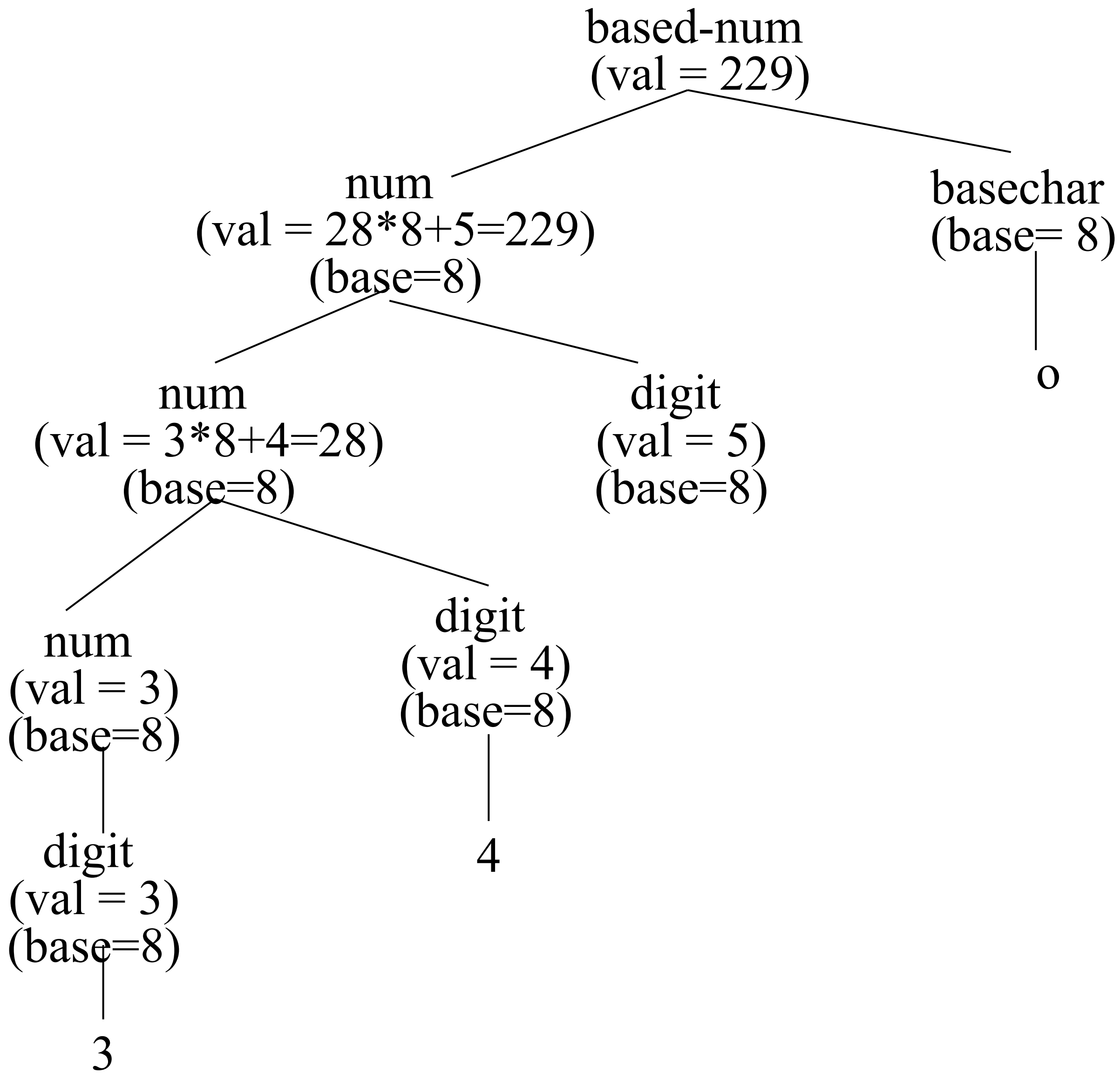
num → *num digit* | *digit*

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

6.1.1 Attribute grammars

grammar Rule	semantic Rules
based-num $\rightarrow$ *num basechar*	*based-num.val = num.val* *num.base = basechar.base*
basechar $\rightarrow$ *o*	*basechar.base = 8*
basechar $\rightarrow$ *d*	*basechar.base = 10*
num1 $\rightarrow$ *num2 digit*	*num1.val =* *If digit.val = error or num2.val = error* *Then error* *Else num2.val*num1.base+digit.val* *num2.base = num1.base* *digit.base = num1.base*
num $\rightarrow$ *digit*	*num.val = digit.val* *digit.base = num.base*
digit $\rightarrow$ *0*	*digit.val = 0*
digit $\rightarrow$ *1*	*digit.val = 1*
…..	…..
digit $\rightarrow$ *7*	*digit.val = 7*
digit $\rightarrow$ *8*	*digit.val = if digit.base = 8 then error rlse 8*
digit $\rightarrow$ *9*	*digit.val = if digit.base = 8 then error rlse 9*

6.1.1 Attribute grammars



6.1.2 Simplifications and Extensions to Attribute Grammars

- *Metalanguage* for the attribute grammar : the collection of expressions allowable in an attribute equation.
  - Here limited to *arithmetic*, *logical* and a few other kinds of *expressions*.
  - an *if-then-else* expression and occasionally a case or switch expression.
- *Functions* can be added to the *metalanguage* whose definitions may be given elsewhere.

  digit → *D* (D is understood to be one of the digits)

  digit.val = numval(D)

6.1.2 Simplifications and Extensions to Attribute Grammars

Simplifications :

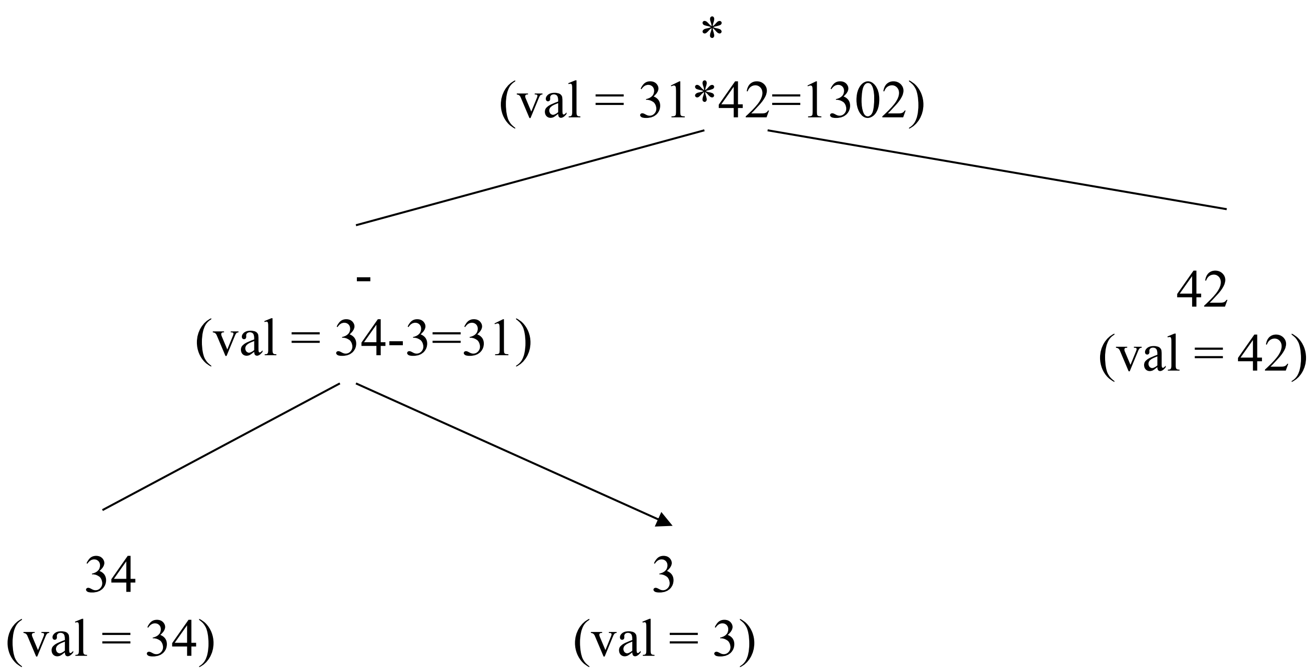
1. Using ambiguous grammar: (all ambiguity will have been dealt with at the parser stage)

exp → *exp* + *exp* | *exp* – *exp* | *exp* * *exp* | (*exp*) | *number*

grammar Rule	semantic Rules
$exp1 \rightarrow exp2 + exp3$	$exp1.val = exp2.val + exp3.val$
$exp1 \rightarrow exp2 - exp3$	$exp1.val = exp2.val - exp3.val$
$exp1 \rightarrow exp2 * exp3$	$exp1.val = exp2.val * exp3.val$
$exp1 \rightarrow (exp2)$	$exp1.val = exp2.val$
$exp \rightarrow number$	$exp.val = number.val$

6.1.2 Simplifications and Extensions to Attribute Grammars

2. Using abstract syntax tree instead of parse tree



6.1.2 Simplifications and Extensions to Attribute Grammars

Example 6.5 define an *abstract syntax tree* for simple integer arithmetic expressions by the attribute grammar as follows:

grammar Rule	semantic Rules
exp1 → *exp2 + term*	*exp1.tree = mkOpNode(+,exp2.tree , term.tree)*
exp1 → *exp2 - term*	*exp1.tree = mkOpNode(-,exp2.tree , term.tree)*
exp1 → *term*	*exp1.tree = term.tree*
term1 → *term2*factor*	*term1.tree= mkOpNode(*,term2.tree , factor.tree)*
term → *factor*	*term.tree = factor.tree*
factor → *(exp)*	*factor.tree = exp.tree*
factor → *number*	*factor.tree = mkNumNode(number.lexval)*

6.2 Algorithms for attribute computation

Purpose:

- Study the ways an attribute grammar can be used as basis for a compiler to compute and use the attributes defined by the equations of the attribute grammar.
- $X_i.a_j = f_{ij}(X_0.a_1, \ldots, X_0.a_k, X_1.a_l, \ldots, X_1.a_k, \ldots\ldots, X_n.a_1, \ldots X_n.a_k)$ is viewed as an assignment of the value of the functional expression on the right- hand side to the attribute $X_i.a_j$.

6.2.1 Dependency Graphs and Evaluation Order

- Each grammar rule choice has an *associated dependency graph*.

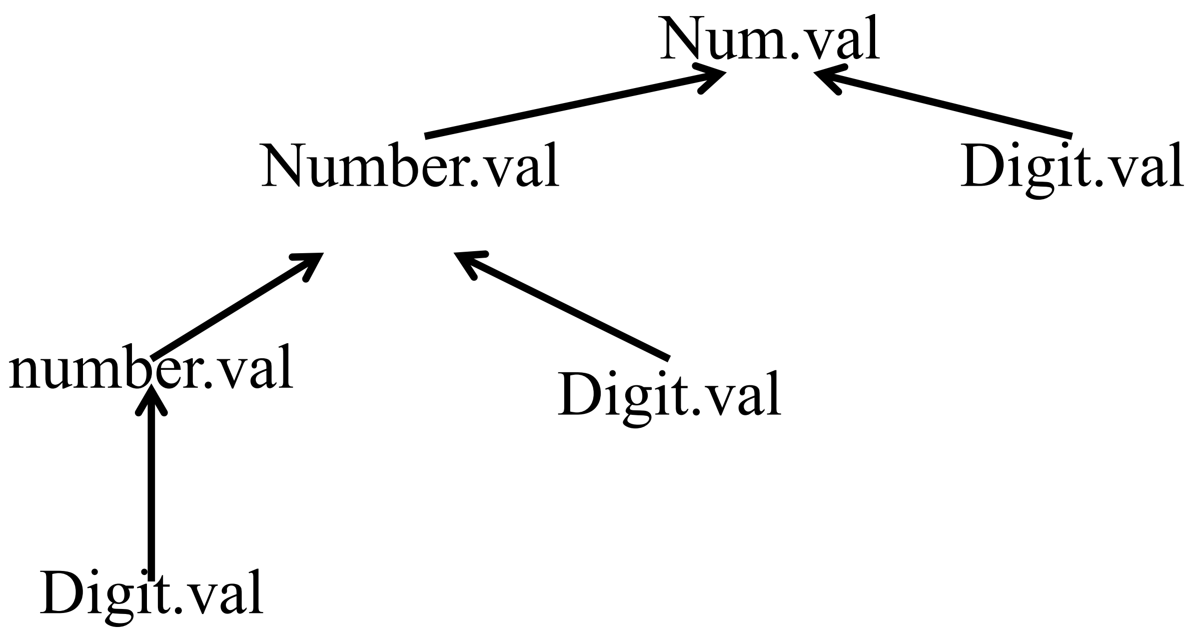
This graph has a node labeled by each attribute $X_i.a_j$ of each symbol in the grammar rule.

- Dependency graph of the string (sentence) is the union of the dependency graphs of the grammar rule choices representing each node(nonleaf) of the parse tree of the string.
- $X_i.a_j = f_{ij}(\ldots, X_m.a_k, \ldots)$

  An edge from each node $X_m.a_k$ to $X_i.a_j$ the node expressing the dependency of $X_i.a_j$ on $X_m.a_k$

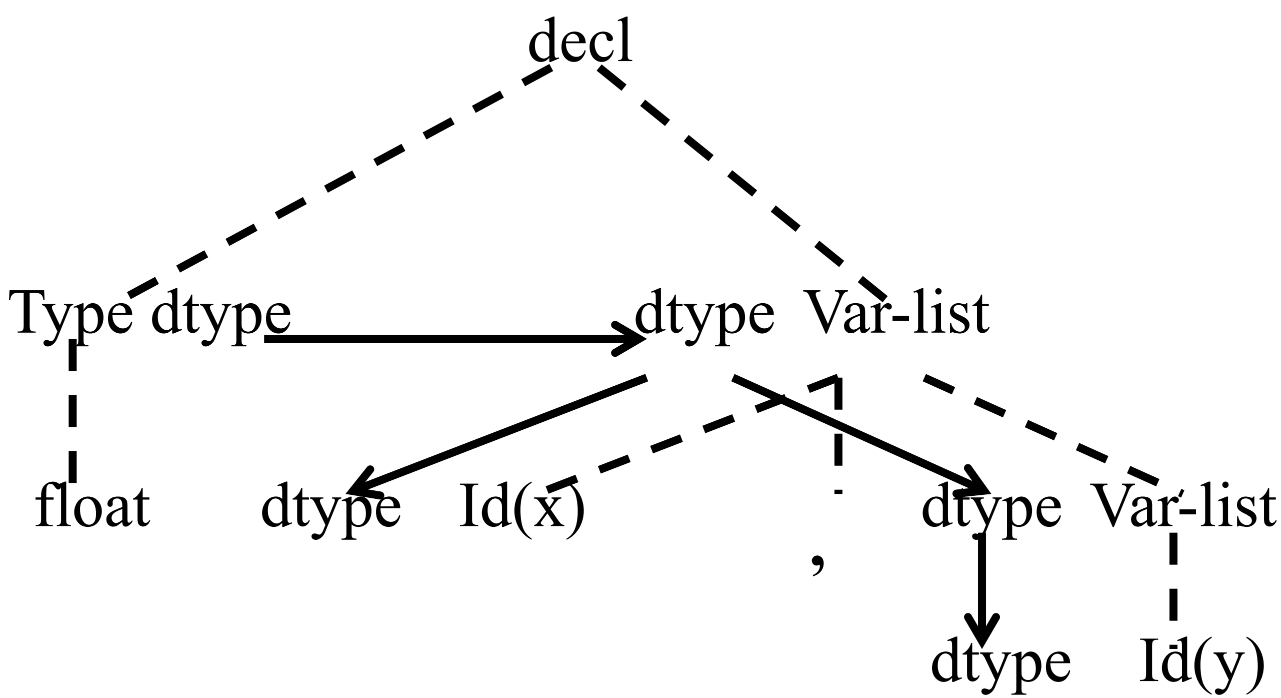
6.2.1 Dependency Graphs and Evaluation Order

The string *345* has the following dependency graph.



6.2.1 Dependency Graphs and Evaluation Order

decl → *type var-list*
type → *int* | *float*
var-list → *id,var-list* | *id*



6.2.1 Dependency Graphs and Evaluation Order

based-num → *num basechar*

num → *num digit*

num → *digit*

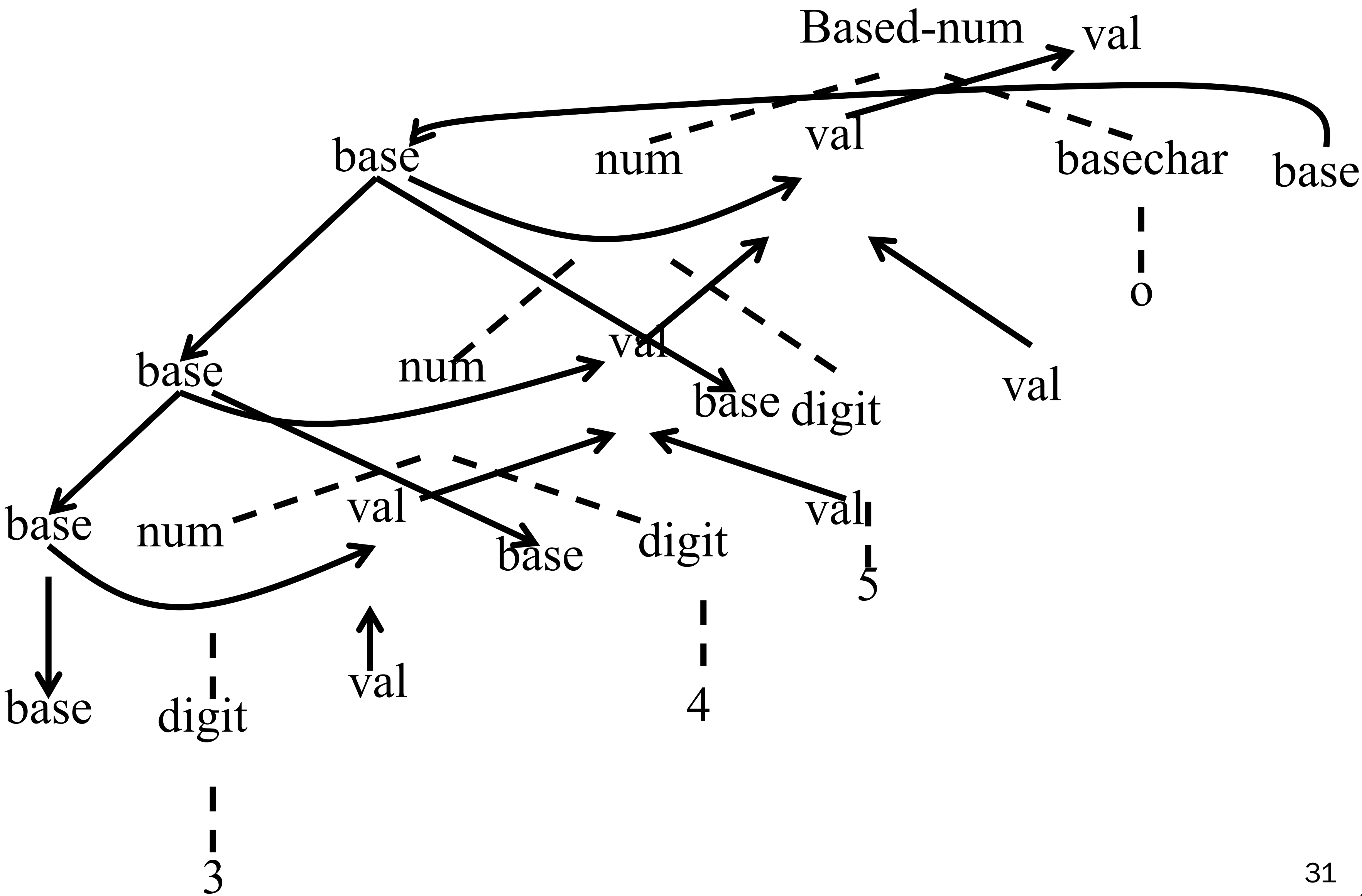
digit → *9*

…..

….

..

6.2.1 Dependency Graphs and Evaluation Order



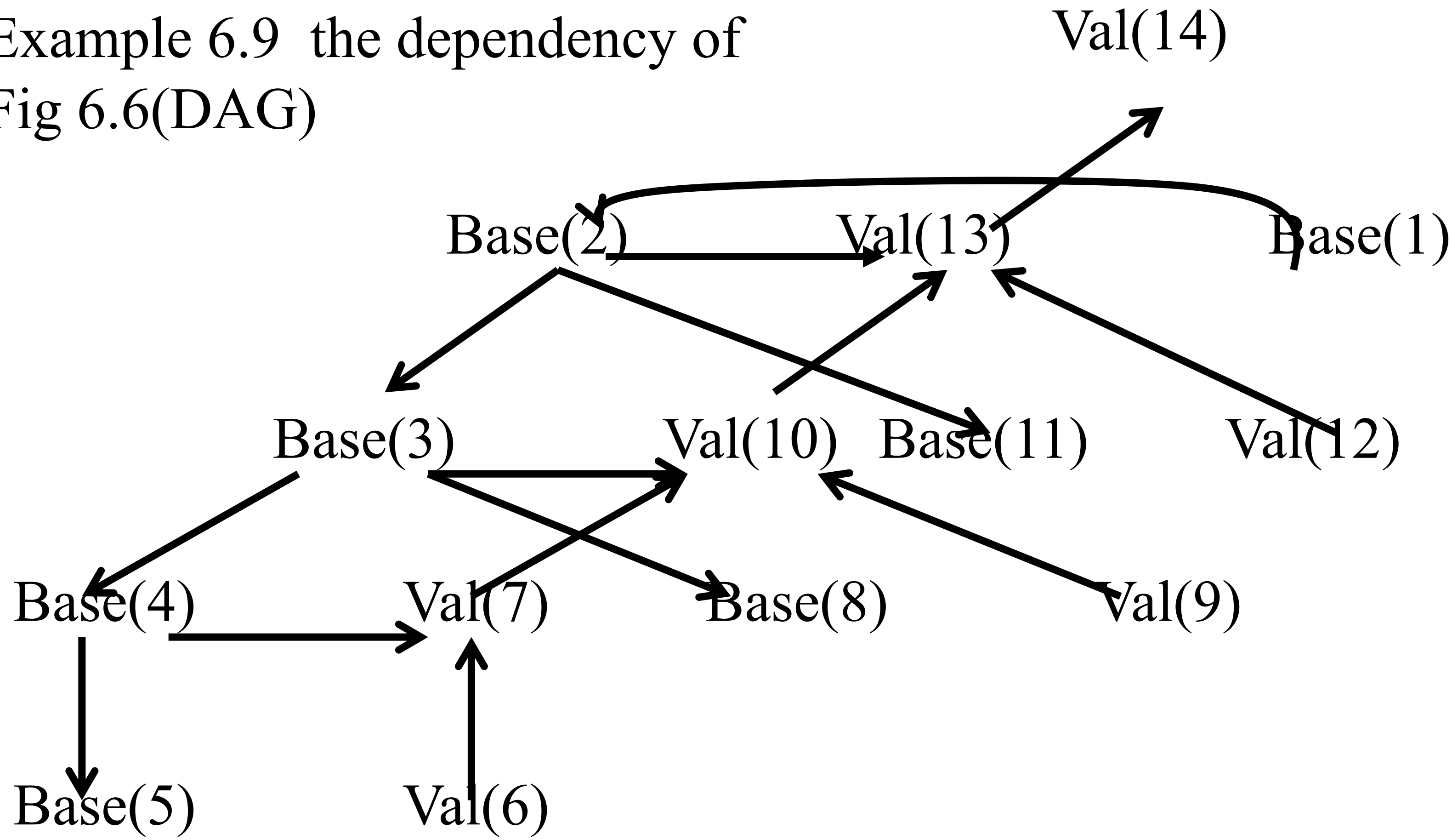
6.2.1 Dependency Graphs and Evaluation Order

Directed acyclic graphs(DAG)

- Algorithm must compute the attribute at each node in the dependency graph before it attempts to compute any successor attributes.
- A traversal order of the dependency graph that obeys this restriction is called a *topological sort*.
- The graph must be *acyclic*

6.2.1 Dependency Graphs and Evaluation Order

Example 6.9 the dependency of Fig 6.6(DAG)



Another topological sort is given by the order
12 6 9 1 2 11 3 8 4 5 7 10 13 14

6.2.1 Dependency Graphs and Evaluation Order

How attribute values are found at the roots of the graph

- Parse tree method: construction of the dependency graph is based on the specific parse tree at compile time., add complexity, and need circularity detective.
- Rule based method: fix an order for attribute evaluation at compiler construction time. It depends on an analysis of the attribute equations, or semantic rules.

6.2.2 Synthesized and inherited attributes

Classification of the attributes:

1. Synthesized attributes
2. Inherited attributes

6.2.2 Synthesized and inherited attributes

Synthesized attributes

- An attribute is *synthesized*
  - if all its dependencies point from child to parent in the parse tree.
  - Given a grammar rule $A \rightarrow X_1 X_2 \dots X_n$, the only associated attribute equation with an a on the left-hand side is of the form:
  - $A.a = f(X_1.a_1, \dots X_1.a_k, \dots X_n.a_1, \dots X_n.a_k)$
- S-attributed grammar:

  An attribute grammar in which all the attributes are *synthesized.*

6.2.2 Synthesized and inherited attributes

Synthesized attributes

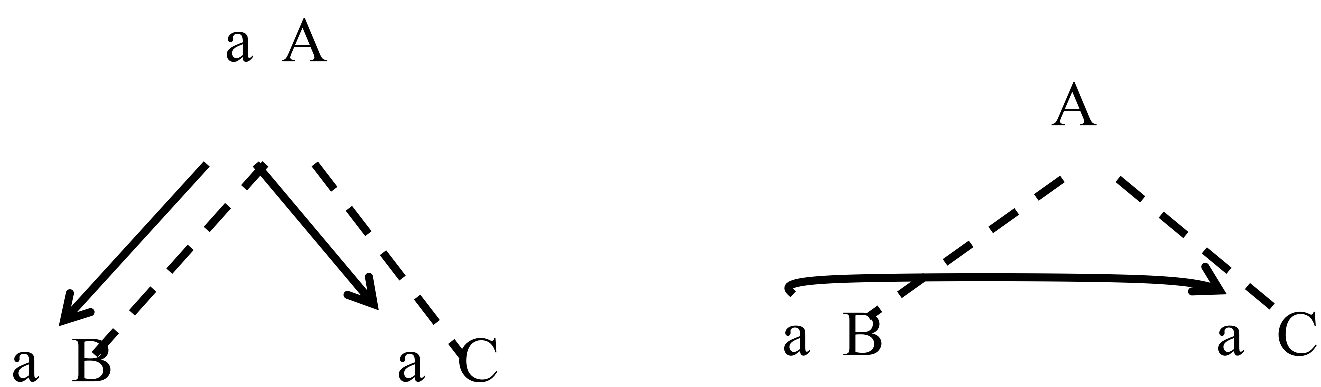
- The attribute values of an *S-attributed* grammar can be computed by a single bottom-up, or post-order, traversal of the parse or syntax tree.

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

6.2.2 Synthesized and inherited attributes

Inherited attributes

- An attribute that is not synthesized is called an *inherited* attribute.
- Such as *dtype* in example 6.3 and *base* in example 6.4.



(a) Inheritance from parent to siblings **(b) inheritance from sibling to sibling**

6.2.2 Synthesized and inherited attributes

Inherited attributes

- Inherited attributes : computed by a preorder traversal , or combined preorder/inorder traversal of the parse or syntax tree, represented by the following pseudocode:

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

6.2.2 Synthesized and inherited attributes

Inherited attributes

- The order in which the *inherited* attributes of the children are computed is important.
- It must adhere to any requirements of the dependencies.

6.2.2 Synthesized and inherited attributes

Example 6.12

- The grammar with the inherited attribute *dtype* and whose dependency graphs are given in example 6.7.

decl → *type var-list*

type → *int* | *float*

var-list → *id, var-list* | *id*

6.2.2 Synthesized and inherited attributes

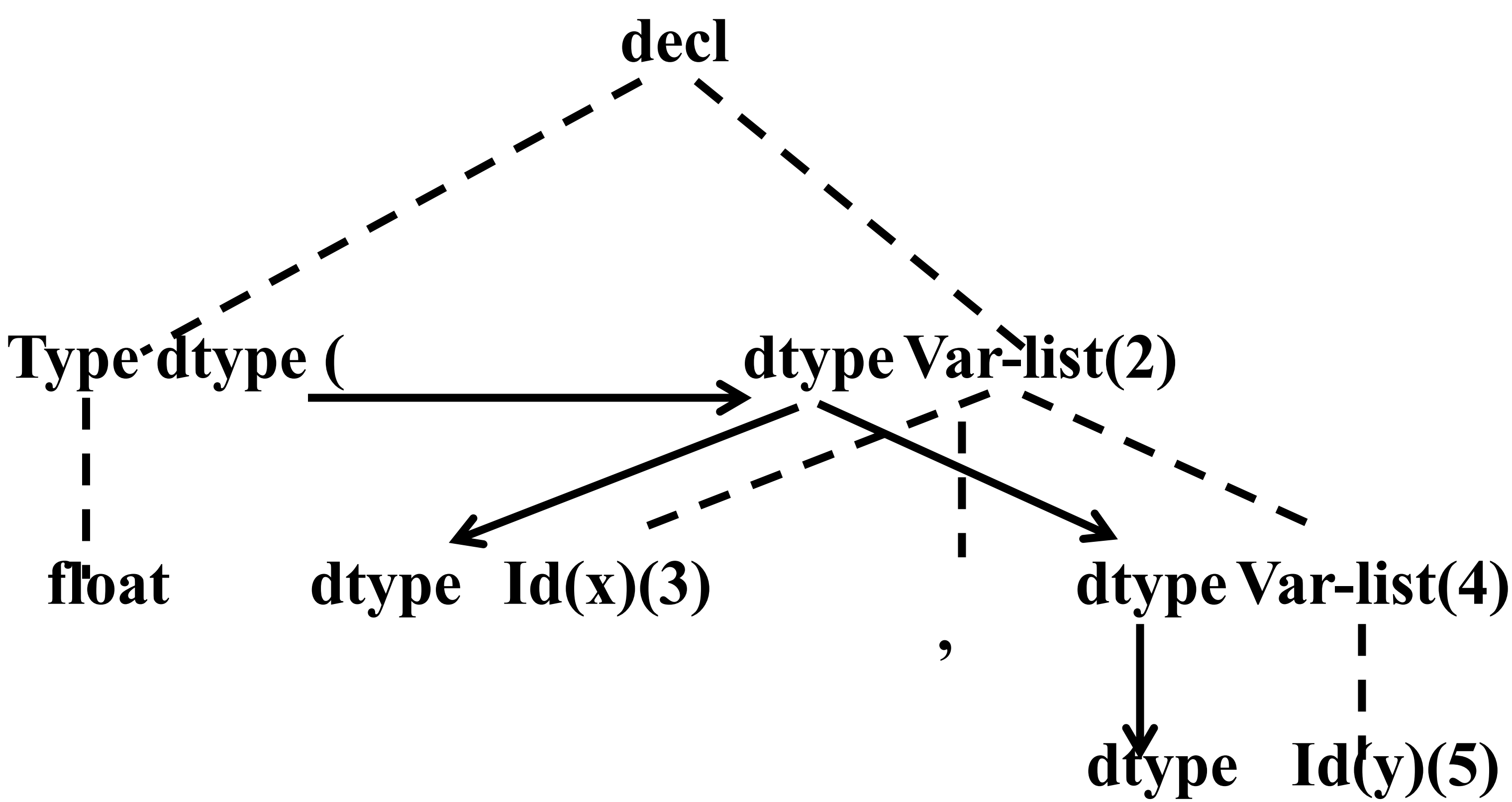
Example 6.12

```
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;
                  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;
                    evalType(third child of T);
      end case;
end EvalType;
```

6.2.2 Synthesized and inherited attributes

Example 6.12

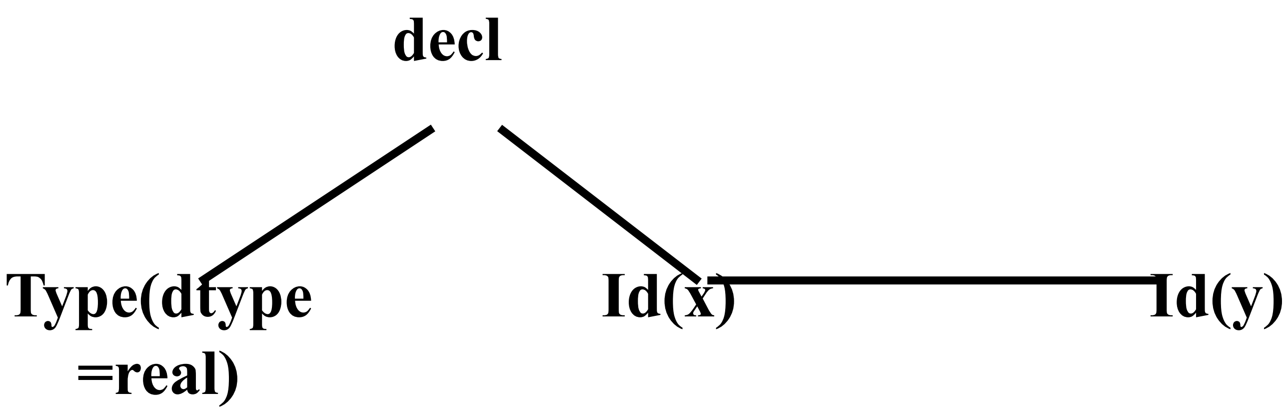
- Preorder and inorder operations are mixed.
- Inorder: *decl* node
- Preorder: *var-list* node



6.2.2 Synthesized and inherited attributes

Example 6.12

- If use the actual C code, assume that a syntax tree has been constructed, in which *var-list* is represented by a sibling list of *id* nodes as follows.



6.2.2 Synthesized and inherited attributes

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;
      char  *name;
     } *Syntaxtree;
```

6.2.2 Synthesized and inherited attributes

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;
    }
}
```

6.2.2 Synthesized and inherited attributes

Example 6.12

```
void  evaltype ( syntaxtree t )
{    if(t->kind = = decl)
     {    syntaxtree p = t->rchild;
          p->dtype = t->lchlild->dtype;
          while (p->sibling !=NULL)
          {p->sibling->dtype = p->dtype;
               p = p->sibling;
               }
     }
}
```

6.2.2 Synthesized and inherited attributes

Example 6.14

- The simple version of an expression grammar:

  exp → *exp* / *exp* | *num* | *num.num*

- Operations may be interpreted differently depending on whether they are *floating-point* or strictly *integer* operations.
- For instance:

  5/2/2.0 = 1.25

  5/2/2 = 1

6.2.2 Synthesized and inherited attributes

Example 6.14

- The attributes needed to express the corresponding semantic:

  isFloat : boolean, indicates if any part of an expression has a floating-point value (*synthesized*)

  etype: gives the type of each subexpression and depends on *isFloat* (*inherited*), here is int or float

  val : gives the numeric value of each subexpression , depends on *etype*.

6.2.2 Synthesized and inherited attributes

Grammar Rule	Semantic Rules
$S \rightarrow exp$	*exp.etype* = if *exp.isFloat* then *float* else *int* *S.val* = *exp.val*
$exp1 \rightarrow exp2/exp3$	*exp1.isFloat* = *exp2.isFloat* or *exp3.isFloat* *exp2.etype* = *exp1.etype* *exp3.etype* = *exp1.etype* *exp1.val* = if *exp1.etype* = int then *exp2.val* div *exp3.val* else *exp2.val/exp3.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*

6.2.3 Attributes as parameters and returned values

- Many attributes are the same or are only used temporarily to compute other attribute values, needn't be stored as fields in a syntax tree record structure.
- Inherited attributes

  be computed in preorder, often be treated as parameters of the call.
- Synthesized attributes

  be computed in postorder, often be treated as returned values of the call.

6.2.3 Attributes as parameters and returned values.

Example 6.15

- The recursive procedure *evalWithBase* of example 6.13
- turn *base* into a parameter and *val* into a returned value.

  based-num → *num basechar*

  basechar → *o*|*d*

  num → *num digit* | *digit*

6.2.3 Attributes as parameters and returned values.

Example 6.15

- To start the computation, one would have to make a call such as *EvalWithBase(rootnode, 0).*

```
function EvalWithBase( T: treenode;  base:integer): integer;
var temp, temp2 : integer;
begin
    case nodekind of T of
    based-num:
            temp := EvalWithBase(right child of T, base);
            return EvalWithBase(left child of T, temp);
    num:
            temp:= EvalWithBase(left child of T, base);
            if right child of T is not nil then
                temp2 := EvalWithBase(right child of T, base);
```

6.2.3 Attributes as parameters and returned values.

Example 6.15

```
            if temp != error and temp2 !=error then
                        return base*temp + temp2
            else return error;
          else return temp;
basechar:
          if child of T = o then return 8
          else return 10;
digit:
          if base = 8 and child of T = 8 or 9 then return error
          else return numval(child of T);
      end case;
end EvalWithBase;
```

6.2.3 Attributes as parameters and returned values.

Example 6.15

```
function EvalBasedNum(T: treenode) : integer;
(/*only called on root node */)
begin
    return EvalNum(left child of T, EvalBase(right child of
    T));
end ;

function EvalBase(T: treenode) : integer;
(/*only called on basechar node*/)
begin
    if child of T = o then return 8
    else return 10;
end
```

6.2.3 Attributes as parameters and returned values.

```
function EvalNum(T:treenode;  base: integer) : integer;
var temp, temp2: integer;
begin
  case nodekond of T of
     num:
        temp:= EvalWithBase(left child of T, base);
        if right child of T is not nil then
          temp2 := EvalWithBase(right child of T, base);
          if temp != error and temp2 !=error then
                    return base*temp + temp2
              else return error;
       else return temp;
     digit:
       if base = 8 and child of T = 8 or 9 then return error
          else return numval(child of T);
     end case;
end.
```

6.2.4 The use of external data structures to store attributes values.

Applicability:

- Not suitable to the method of *parameters* and *returned values*
- Particularly when the attribute values have significant structure and may be needed at arbitrary points during translation.
- Not reasonable to be stored in the syntax tree nodes.

6.2.4 The use of external data structures to store attributes values.

Ways:

- External data structures : table, graphs and other data structures. one of the prime examples is the symbol table.
- Replace attribute equations by calls to procedures representing operations on the appropriate data structure used to maintain the attribute values.

6.2.5 The computation of attributes during parsing.

- Attributes that computed successfully at the same time as the parsing stage depends on the power and properties of the parsing method employed.
- All the major parsing methods process the input program from left to right (LL, or LR) .
- Require the attribute be capable of evaluation by a left-to-right traversal of the parse tree (*synthesized* attributes will always be OK).

6.2.5 The computation of attributes during parsing

L-attributed Definition:

- An attribute grammar for attribute $a_1, \ldots, a_k$ is *L-attributed* if , for each inherited attribute a_j and each grammar rule:

  $X_0 \rightarrow X_1 X_2 \ldots X_n$

  The associated equations for a_j are all of the form:

  $X_i.a_j = f_{ij}(X_0.a_1, , X_0.a_k, X_1.a_1, \ldots, X_1.a_k, .X_{i-1}.a_1, \ldots X_{i-1}.a_k)$
- *S-attributed grammar* is *L-attributed* grammar.

6.2.5 The computation of attributes during parsing.

L-attributed

Given an *L-attributed* grammar in which the *inherited* attributes do not depend on the *synthesized* attributes:

1、Top-down parser: a recursive-descent parser can evaluate all the attributes by turning the *inherited* attributes into *parameters* and *synthesized* attributes into *returned values*.

2、Bottom-up parser: LR parsers are suited to handling primarily *synthesized* attributes, but are difficult for *inherited* attributes.

Computing *synthesized* attributes during LR parsing.

Value stack: store *synthesized* attributes, be manipulated in parallel with the parsing stack.

6.2.5 The computation of attributes during parsing.

Computing synthesized attributes during LR parsing

$	3*4+5$	Shift	$	
$n	*4+5$	Reduce E→n	$n	E.val = n.val
$E	*4+5$	Shift	$3	
$E*	4+5$	Shift	$3*	
$E*n	+5$	Reduce E→n	$3*n	E.val =n.val
$E*E	+5$	Reduce E→E*E	$3*4	E1.val=E2.val*E3. val
$E	+5$	Shift	$12	
$E+	5$	Shift	$12+	
$E+n	$	Reduce E→n	$12+n	E.val = n.val
$E+E	$	Reduce E→E+E	$12+5	E1.val=E2.val +E3.val
$E	$		$17	

6.2.5 The computation of attributes during parsing.

Inheriting a previously computed synthesized attributes during LR parsing

An action associated to a nonterminal in the right-hand side of a rule can make use of *synthesized* attributes of the symbols to the left of it in the rule.

- For instance:

  $A \rightarrow B\ C$

  $C.i = f(B.s)$ s is a *synthesized* attribute.

6.2.5 The computation of attributes during parsing

- The question can be settled through a ε–production as follows

Grammar Rule	Semantic Rules
$A \rightarrow BDC$	
$B \rightarrow \ldots$	{computer B.s}
$D \rightarrow \varepsilon$	saved_i= f(valstack[top])
$C \rightarrow \ldots$	{now saved_i is available}

- In Yacc this process is made easier. The action of storing the computed attribute is simply written at the place in the rule where it is to be scheduled:

  A: B { saved_i=f($1);} *C;*

6.2.5 The computation of attributes during parsing

The following attribute grammar satisfy the above request.

Grammar Rule	Semantic Rules
decl → *type var-list*	*var-list.dtype = type.dtype*
type → *int*	*type.dtype = integer*
type → *float*	*type.dtype = real*
var-list1 → *var-list2,id*	*insert(id.name, var-list1.dtype)* *var-list2.dtype = var-list1.dtype*
var-list → *id*	*insert(id.name, var-list.dtype)*

6.2.5 The computation of attributes during parsing

Problems

1. Require the programmer to directly access the value stack during a parse

   This may be risky in automatically generated parsers.

2. Only works if the position of the previously computed attribute is predictable from the grammar.

   The best technique for dealing with inherited attributes in LR parsing:

   Use external data structures, to hold *inherited* attribute values and to add *ε_production* or embedded actions as in Yacc (may add parsing conflicts).

6.2.6 The dependence of attributes computation on the syntax

- Modifications to the grammar that do not change the legal strings of the language

  Make the computation of attributes simpler or more complex.

- The properties of attributes depend heavily on the structure of the grammar.

6.2.6 The dependence of attributes computation on the syntax

Theorem

Given an attribute grammar , all inherited attributes can be changed into synthesized attributes by suitable modification of the grammar, without changing the language of the grammar.

(From Knuth [1968]).

6.2.6 The dependence of attributes computation on the syntax

Example 6.18

- An inherited attribute can be turned into a synthesized attribute by modification of the grammar.
- Consider the grammar as follows:

  decl → *type var-list*

  type → *int|float*

  var-list → *id,var-list|id*

- The *dtype* attribute is inherited. Rewrite the grammar as follows

  decl → *var-list id*

  var-list → *var-list id* , | *type*

  type → *int* | *float*

6.2.6 The dependence of attributes computation on the syntax

- Turned the inherit attribute into synthesized attribute as follows:

Grammar Rule	Semantic Rules
decl → *var-list id*	*id.dtype = var-list.dtype*
var-list1 → *var-list2 id*	*varlist1.dtype = varlist2.dtype* *id.dtype = varlist2.dtype*
var-list → *type*	*var-list.dtype = type.dtype*
type → *int*	*type.dtype = integer*
type → *float*	*type.dtype = real*