Compiler Principle and Technology

Prof. Dongming LU
Apr. 15th, 2015

6. Semantic Analysis

- **Semantic Analysis Phase**
  - **Purpose**: compute additional information needed for compilation that is beyond the capabilities of Context-Free Grammars and Standard Parsing Algorithms
  - **Static semantic analysis:** Take place prior to execution (Such as building a symbol table、performing type inference and type checking)
- **Classification**
  - Analysis of a program required by the rules of the programming language to establish its correctness and to guarantee proper execution
  - Analysis performed by a compiler to enhance the efficiency of execution of the translated program

- **Description** of the static semantic analysis
  - **Attribute grammar**
    - identify **attributes** of language entities that must be computed
    - and to write **attribute equations** or **semantic rules** that express how the computation of such attributes is related to the grammar rules of the language
  - Which is most useful for languages that obey the principle of **Syntax-Directed Semantics**
  - **Abstract syntax** as represented by an abstract syntax tree

- **Implementation** of the static semantic analysis:
  - Not as clearly expressible as parsing algorithms because of the addition problem caused by the timing of the analysis during the compilation process
  - Multi-pass (more common) or single pass lead to totally different process

- **Emphasis:**
  - Attributes and attribute grammars
  - Algorithms for attribute computation
  - The symbol table
  - Data types and type checking

Contents

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.
  - Attribute equations is turned into computation rules
  - $X_i.a_j = f_{ij}(X_0.a_1,\ldots,X_0.a_k, \ldots, X_1.a_l, \ldots, X_{n-1}.a_1, \ldots X_n.a_k)$
  - Viewed as an assignment of the value of the functional expression on the right- hand side to the attribute $X_i.a_j$.

- The attribute equations indicate the **order constraints on the computation** of the attributes.
  - Attribute at the right-hand side must be computed before that at the left-hand side.
  - The constraints is represented by directed graphs — **dependency graphs.**

6.2.1 Dependency graphs and evaluation order

- **Dependency graph** of the string:

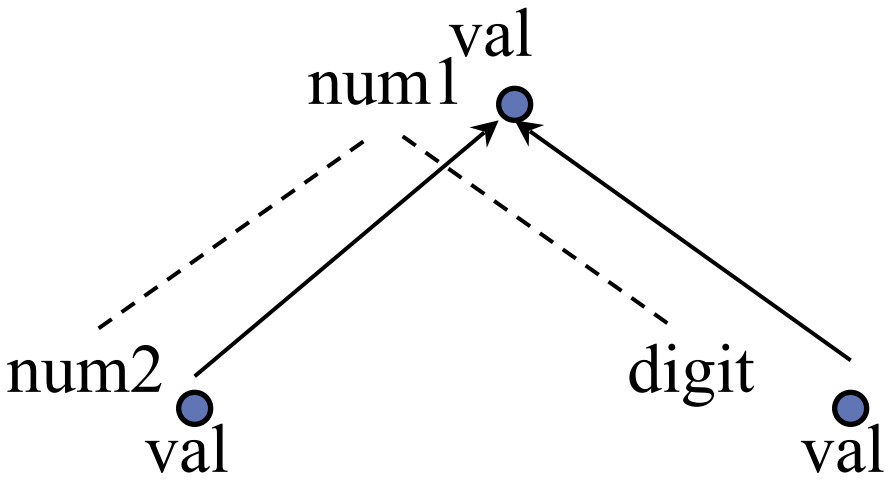
  The **union of the dependency graphs** of the grammar rule choices representing each node (nonleaf) of the parse tree of the string

  - $X_i.a_j = f_{ij}(\ldots, X_m.a_k, \ldots)$
  - 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 $X_m.a_k$.

Example 6.6 Consider the grammar of Example 6.1, with the attribute grammar as given in tab 6.1. for each symbol there is only one node in each dependency graph, corresponding to its val attribute

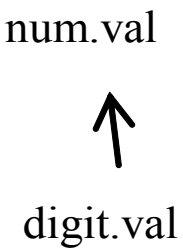
Grammar rule	attribute equation
number1→number2 digit	number1.val = number2.val * 10 +digit.val

The dependency graph for this grammar rule choice is

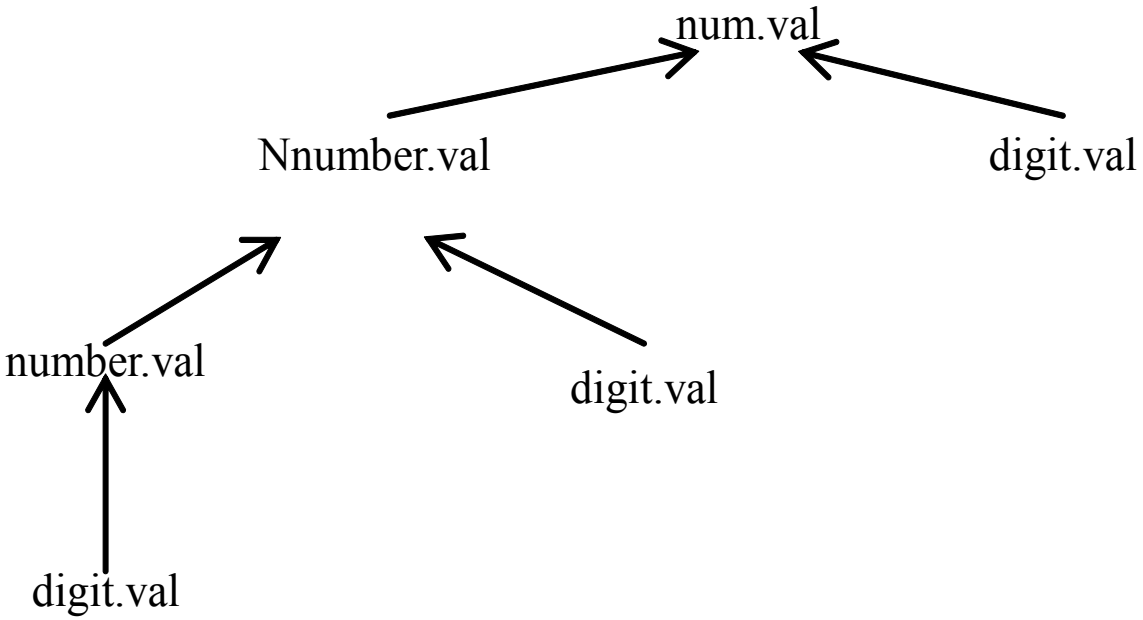


The subscripts for repeated symbols will be omitted

number→ digit number.val = digit.val



The string 345 has the following dependency graph.



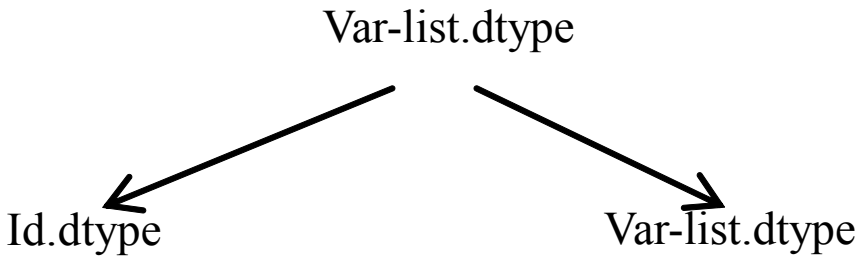
Example 6.7 Consider the grammar of example 6.3

var-list1 →id, varlist2

id.dtype = var-list1.dtype

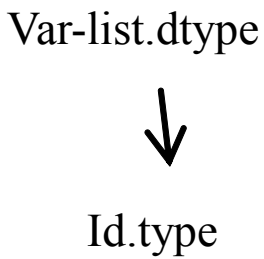
var-list2.dtype = var-list1.dtype

And the dependency graph



Similarly, var-list →id

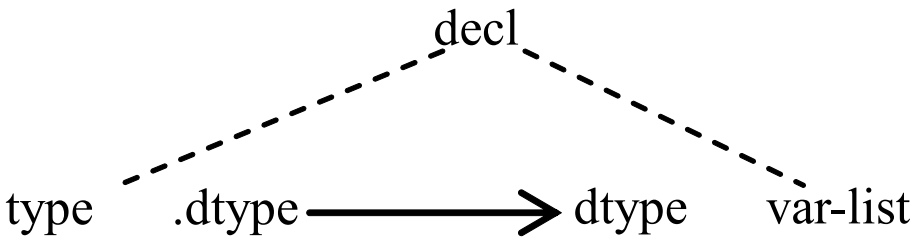
respond to



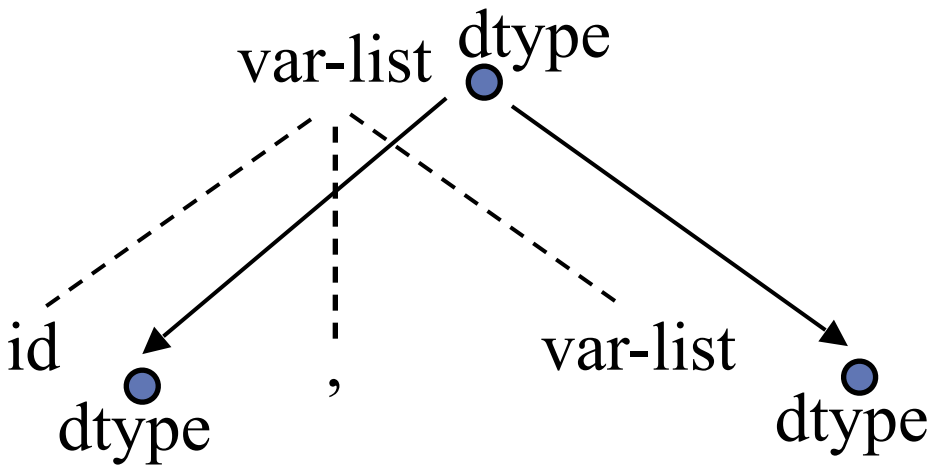
decl→type varlist

type.dtype ——> var-list.dtype

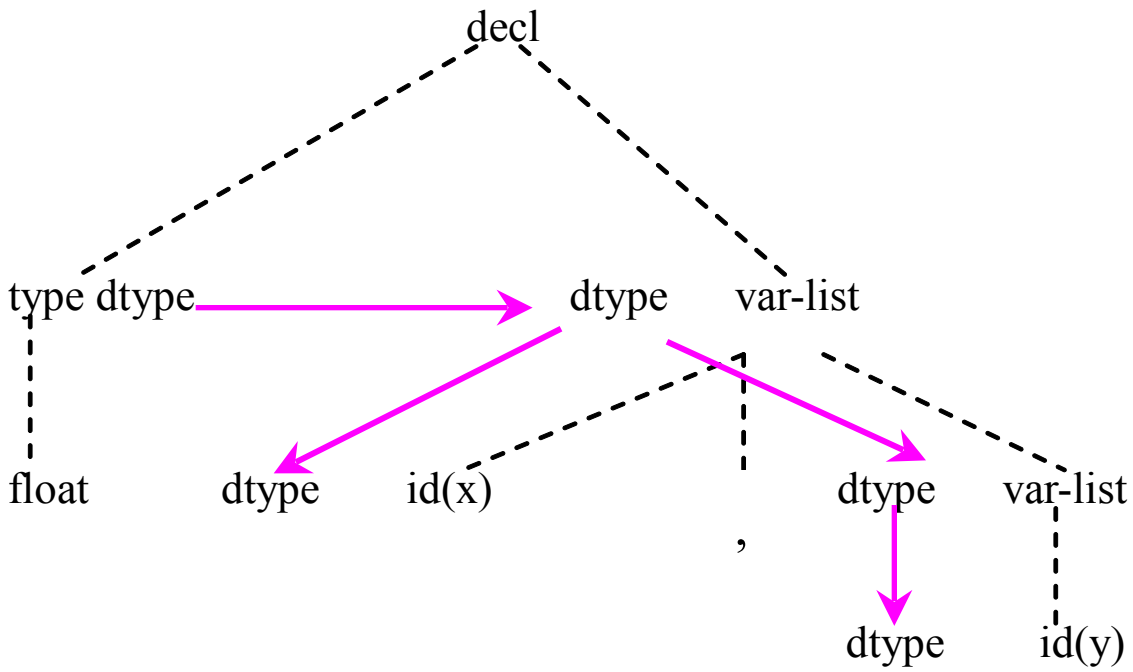
It can also be drawn as:



So, the first graph in this example can be drawn as :



Finally, the dependency graph for the string float x, y is



Example 6.8 Consider the grammar of based number of example 6.4

based-num→num basechar

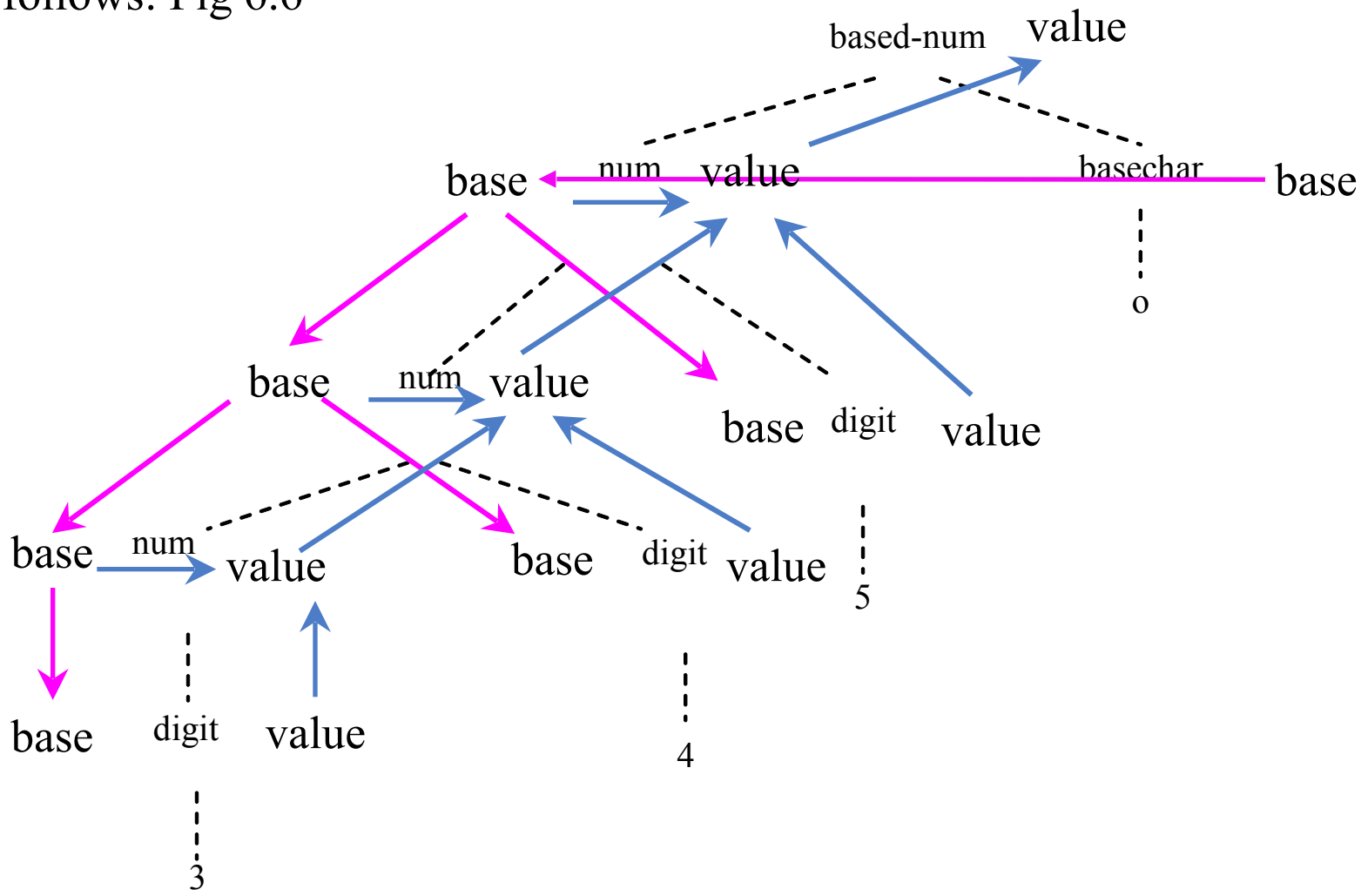
num→num digit

num→digit

digit→9

…

For string 345o, the corresponding dependency graph is as follows: Fig 6.6



- Any 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**
  - Requiring that the graph must be **acyclic**, such graphs are called **directed acyclic graphs** Or **DAG**s

Example 6.9

The dependency of Fig 6.6 is a DAG, which gives as follows:

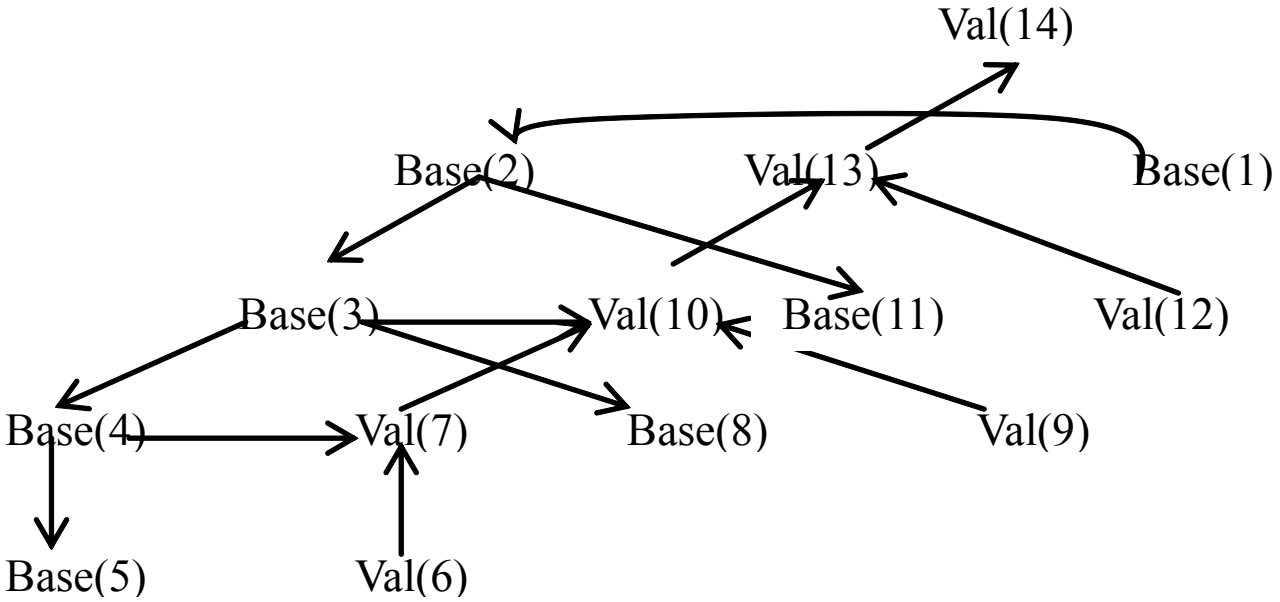


Fig 6.7

Another topological sort is given by the order

12 6 9 1 2 11 3 8 4 5 7 10 13 14

- **One question:**
  - How attribute values are found at the roots of the graph (root of the graph mean that has no predecessors)
  - Attribute values at these nodes is often in the form of tokens, and should be computed before any other attribute values
- **Parse tree method**:
  - Construction of the dependency graph is based on the specific parse tree at compile time, add complexity, and need circularity detective, the method fails when there is a cycle
- **Rule based method**:
  - Fix an order for attribute evaluation at compiler construction time

6.2.2 Synthesized and Inherited Attributes

- **Classification** of the attributes:
  1. Synthesized attributes
  2. Inherited attributes
- **Definition:**
  - An attribute is **synthesized** if all its dependencies point form child to parent in the parse tree.
  - Equivalently, an attribute a is synthesized if, given a grammar rule $A \rightarrow X_1 X_2 \ldots X_n$, the only associated attribute equation with an '**a**' on the left-hand side is of the form:
    - $A._a = f(X_1.a_1, \ldots X_1.a_k, \ldots X_n.a_1, \ldots X_n.a_k)$

- **An S-attributed grammar**
  - An attribute grammar in which all the attributes are synthesized
- 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.**Just as follows:

```
Procedure PostEval (T : treenode);
Begin
    For each child C of T do
    PostEval(C);
    Compute all synthesized attributes of T;
End
```

- **Example 6.11** Consider the attribute grammar for simple arithmetic expressions (example 6.2), with the synthesized attribute val
- Given the following structure for a syntax tree:

```
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;
```

The PostEval pseudocode would translate into the C code as follows:

```
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;
  }
  }
}
```

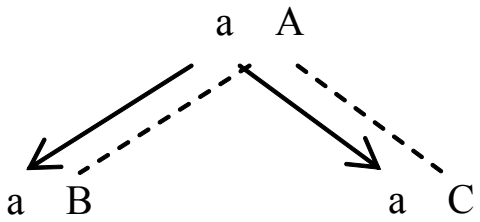
Definition:

An attribute that is not synthesized is called an **inherited** attribute.

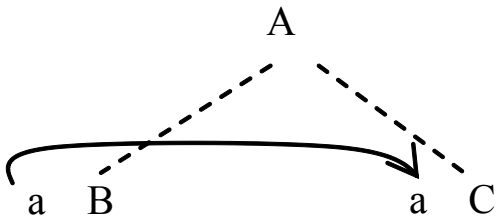
Such as dtype in the example below:

Decl → type var-list
Type→int|float
Var-list→id,var-list|id

Two basic kinds of dependency of inherited attributes:

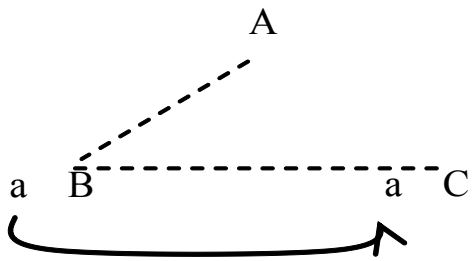


(a) Inheritance form parent to siblings



(b) Inheritance from sibling to sibling

If some structures in a syntax tree are implemented via sibling pointers, then sibling inheritance can proceed directly along a sibling chain, as follows:



(c) Sibling inheritance via sibling pointers

Inherited attributes can be 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    T;
        PreEval(C);
End;
```

- The order in which the inherited attributes of the children are computed is important
  - Adhere to any requirements of the dependencies

- **Example 6.12** Consider the grammar with the inherited attribute dtype and its dependency graphs as illustrated

  Decl → type var-list

  Type→int|float

  Var-list→id,var-list|id

```
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;
```

In the procedure above, preorder and inorder operations are mixed.

Inorder: decl node

Preorder: var-list node

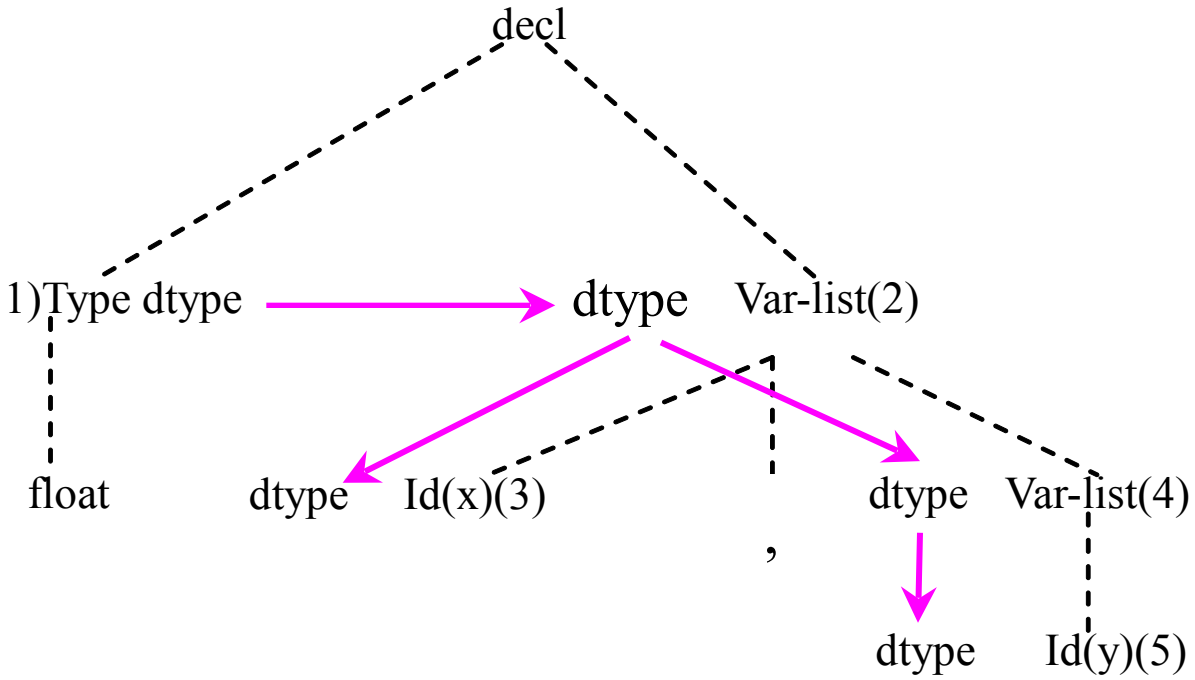
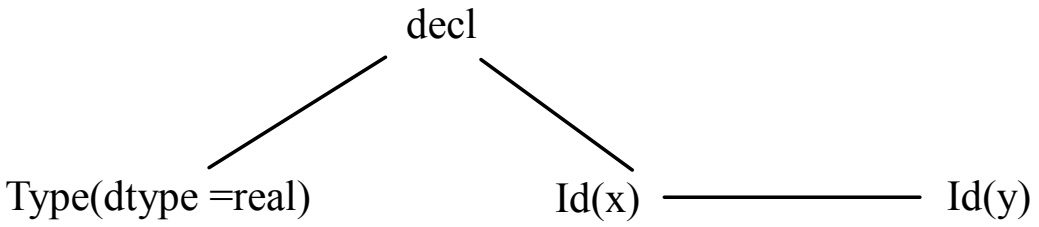


Fig 6.10

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**.



The syntax tree structure is :

```
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;
```

The EvalType procedure now has the corresponding C code:

```
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;
  }
}
```

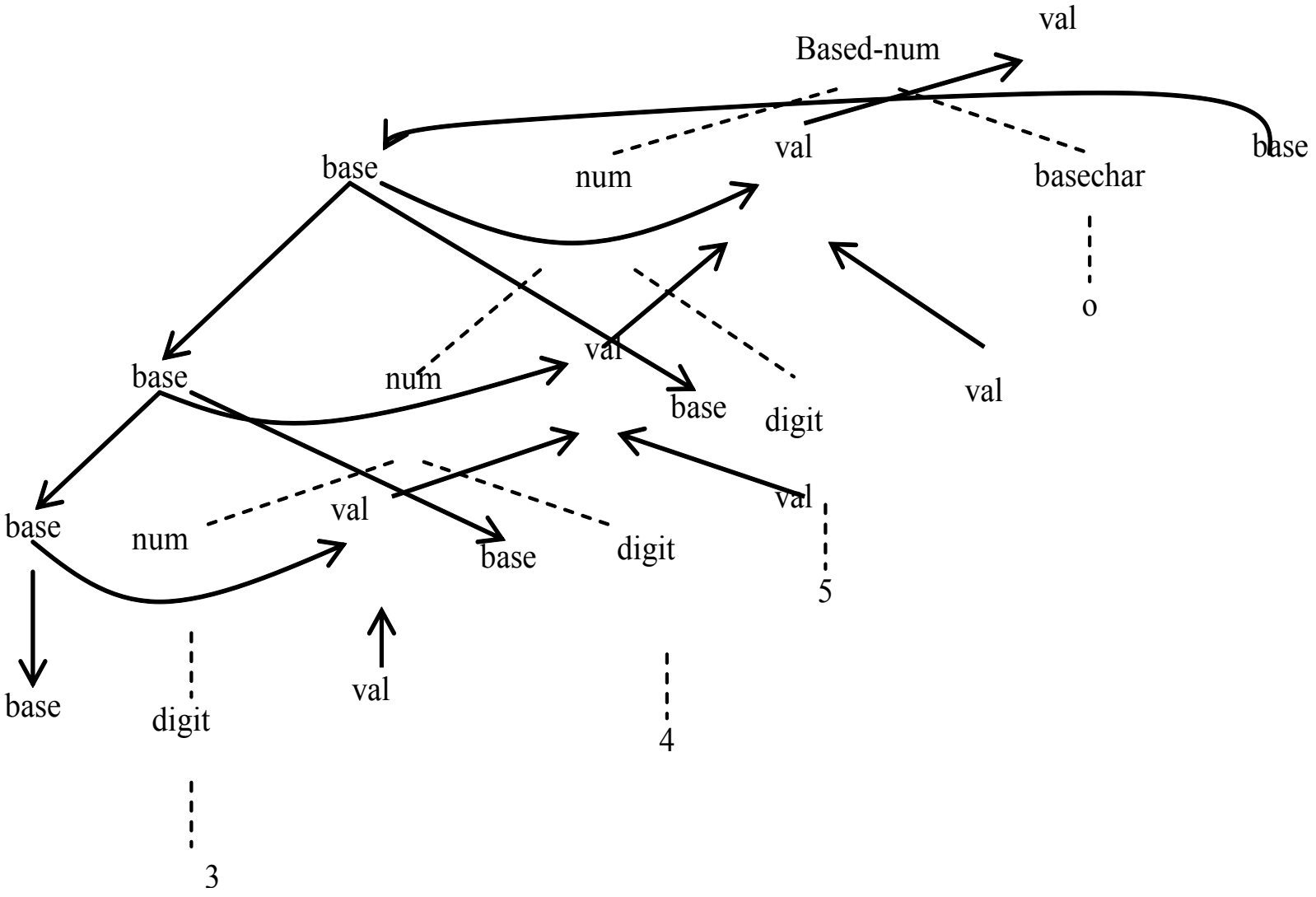
The code above can be simplified to the following non-recursive procedure:

```
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;
            }
            }
}
```

- **Example 6.13** Consider the following grammar which has the **inherited attribute base**.

  Based-num → num basechar
  Basechar → o|d
  Num → num digit | digit
  Digit → 0|1|2|3|4|5|6|7|8|9

- Features:
  - Two attributes include synthesized attribute val and inherited attribute base, on which val depends;
  - The base attribute is inherited from the right child to the left child of a based-num.



Base is computed in preorder and val in postorder during a single pass as follows.

```
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;
        EvalWith Base(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;
```

- To compute all the attributes **in a single pass over the parse or syntax tree,** the **inherited attributes shouldn't depend on any synthesized attributes**
- The synthesized attributes could depend on the inherited attributes (as well as other synthesized attributes).

```
Procedure CombinedEval(T:treenode);
Begin
  For each child C of T do
        Compute all inherited attributes of C;
        CombinedEval(C);
  Compute all synthesized attributes of T.
End;
```

- When inherited attributes depend on synthesized attributes, it requires more than one pass over the parse or syntax tree

- **Example 6.14** Consider the following 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

- The attributes to express 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.

Grammar Rule	Semantic Rules
S→exp	exp.etype = if exp.isFloat then float else int S.val = exp.val
Exp1→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→num	exp.isFloat = false exp.val = if exp.etype = int then num.val else Float(num.val)
exp→num.num	exp.isFloat = true exp.val = num.num.val

Tab. 6.7

The attributes can be computed by two passes over the parse or syntax tree.
First pass: synthesized attribute isFloat, by postorder traversal
Second pass: inherited attribute etype and synthesized attribute val in a combined traversal (pre/post order).

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:**
  - Computed in preorder, often be treated as **parameters of the call**
- **Synthesized attributes:**
  - Computed in postorder, often be treated as **returned values of the call**

- **Example 6.15** Consider the recursive procedure EvalWithBase of example 6.13, we can turn base into a parameter and val into a returned val.

```
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,0);
          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);
                  If temp != error and temp2 !=error then
                          Return base*temp + temp2
                  Else return error;
          Else return temp;
```

```
Basechar:
        If child of T = 0 then retur 8
        Else return 10;
Digit:
        If base = 8 and child of T = 8 or 9 then return error
        Else return numbal(child of T);
End case;
End EvalWithBase;
```

- To start the computation, one would have to make a call such as

  EvalWithBase(rootnode, 0)
- To distinguish three cases, it's more national to write three separate procedures as follows:

```
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 = 0 then return 8
            else return 10;
      end
```

```
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 numbal(child of T);
  End case;
End.
```

6.2.4 The Use of External Data Structures to Store Attributes Values

- **Applicability:**
  - When not suitable to the method of parameters and returned values
    - The attribute values have significant structure
    - May be needed at arbitrary points during translation
  - and not reasonable to be stored in the syntax tree nodes.
- **Ways:**
  - Use external data structures such as table, graphs and other data structures to store the corresponding attribute values;
    - 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

```
function EvaWithBase(T:treenode) : integer;
var temp, temp2: integer;
begin
  case nodekond of T of
  based-num:
        SetBase( right child of T);
        Return EvalWithBase(left child of T);
  Num:
        Temp:= EvalWithBase(left child of T);
        If right child of T is not nil then
              Temp2 := EvalWithBase(right child of T);
              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.
```

```
Procedure SetBase(T:treenode);
Begin
  If child of T = 0 then base :=8
  Else base :=10
End
```

We can also change the semantic rules to reflect the use of the nonlocal base variable.

Grammar Rule	Semantic Rules
Based-num→num basechar	based-num.val = num.bval
Basechar →o	base = 8
Basechar→ d	base = 10
Num1→num2 digit	num1.val = If digit.val = error or num2.val = error Then error Else num2.val* base+digit.val
Etc.	Etc.

Base is no longer an attribute, the semantic rules no longer form an attribute grammar as before.

Example 6.17 consider the attribute grammar of simple declarations given in example 6.12 , treat the dtype as a nonlocal variable, use symbol table to store the dtype values.

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

Procedure insert(name:string; dtype: typekind) is a function which store the identifier name together with its declared data type into the symbol table.

The corresponding pseudocode for an attribute evaluation procedure EvalType is as follows:

```
Procedure EvalType(T: treenode);
Begin
        Case nodekind of T of
        Decl:
                EvalType(type child of T)
                EvalType(var-list child of T)
    Type:
            If child of T = int then dtype := integer;
            Else dtype :=real;
    Var-list:
            Insert (name of first child of T, dtype)
            If third child of T is not nil then
              EvalType(third child of T);
    End case
End
```

6.2.5 The Computation of Attributes during Parsing

- What extent attributes can be 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)
- **Definition:**
  - An attribute grammar for attribute $a_1, \ldots, a_k$ is **L-attributed** if, for each inherited attribute *aj* and each grammar rule:

$$X_0 \rightarrow X_1 X_2 \ldots X_n$$

- The associated equations for aj are all of the form:
  - *Xi.aj = fij(X0.a1,…,X0.ak, …, X1.al, …, Xi-1.a1, …Xi-1.ak)*
  - That is, **the value of *aj* at *Xi* can only depend on attribute of the symbols *X0*, … *Xi-1* that occur to the left of *Xi* in the grammar rule**

- S-attributed grammar is L-attributed grammar.

- Given an L-attributed grammar in which the inherited attributes do not depend on the synthesized attributes:
  - 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.
  - Bottom-up parser: LR parsers are suited to handling primarily synthesized attributes, but are difficult for inherited attributes.

1) Computing synthesized attributes during LR parsing

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

Parsing stack	input	parsing action	value stack	semantic action
1	$	3*4+5$	Shift	$	
2	$n	*4+5$	Reduce E→n	$n	E.val = n.val
3	$E	*4+5$	Shift	$3	
4	$E*	4+5$	Shift	$3*	
5	$E*n	+5$	Reduce E→n	$3*n	E.val =n.val
6	$E*E	+5$	Reduce E→E*E	$3*4	E1.val=E2.val*E3.val
7	$E	+5$	Shift	$12	
8	$E+	5$	Shift	$12+	
9	$E+n	$	Reduce E→n	$12+n	E.val = n.val
10	$E+E	$	Reduce E→E+E	$12+5	E1.val=E2.val +E3.val
11	$E	$		$17	

1) 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→ B C

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

The question can be settled through an ε –production as follows

Grammar Rule	Semantic Rules
A→BDC	
B→…	{computer B.s}
D→ε	saved_i= f(valstack[top])
C→…	{now saved_i is available}

The strategy can be OK without ε -production when the position of a previously computed synthesized attribute in the value stack can always be predicted.

For instance: 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→id,var-list2	insert(id.name, dtype), var-list2.dtype = var-list1.dtype
var-list→id	insert(id.name, dtype)

it can be computed without ε -production as follows.

Grammar Rule	Semantic Rules
decl→type var-list	
type→int	type.dtype = integer
type→ float	type.dtype = real
var-list1→id,var-list2	insert(id.name, valstack[top-3]),
var-list→id	insert(id.name, valstack[top-1])

Problems:

1. require the programmer to directly access the value stack during a parse, is risky in automatically generated parsers;
2. only works if the position of the previously computed attribute is predictable from the grammar.

By far the best technique for dealing with inherited attributes in LR parsing is to use external data structures, to hold inherited attribute values and to add ε _production (may add parsing conflicts.

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

Example 6.18 how 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, we can rewrite the grammar as follows

Decl → var-list id

Var-list→ var-list id, | type

Type → int | float

We can then truned 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 = varlist1.dtype
var-list→type	var-list.dtype = type.dtype
type→int	type.dtype = integer
type→ float	type.dtype = real

The change in the grammar affects the parse tree and the dtype attribute computation as follows. String float x, y

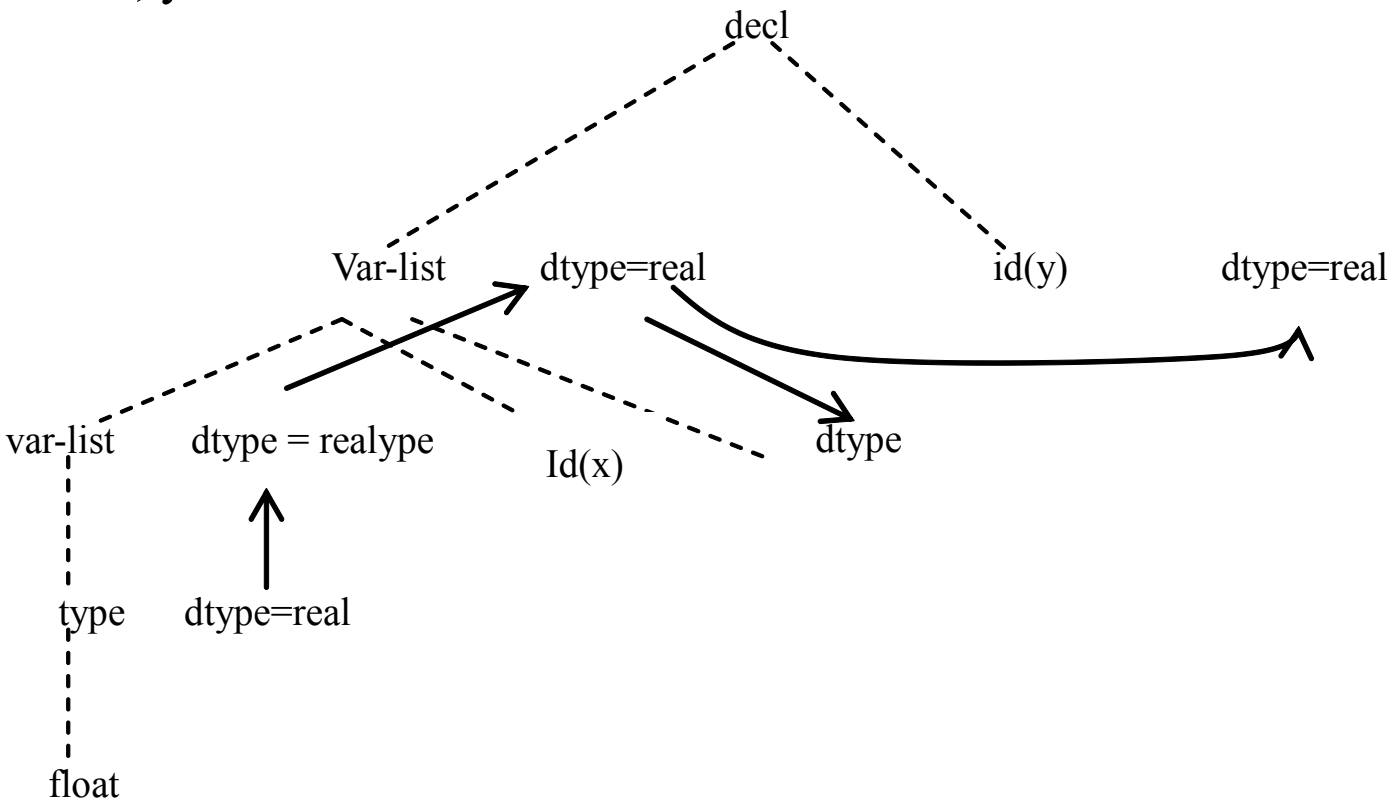


Fig 6.11

End of Part Two

THANKS