Chapter 6 Semantic Analysis

- Study Goals:
- Master：
  Dependency graphs, Algorithms for attribute computation
- Understand:
  Attribute grammar, Synthesized and Inherited attributes, S-attributed grammar, L-attributed grammar, Symbol table
- Know:

Overview of Semantic Analysis

1 Semantic
2 Semantic Analysis
3Typical realization of semantic analysis
4 Method of Semantic Analysis

1 Semantic

- Information that is closely related to the eventual meaning of the program being translated
- Two kinds of semantic

a) Static semantic is defined statically, which can be determined prior to execution.
   compiler performs static semantic analysis

b) Dynamic semantic can only be determined when execution

2 Semantic Analysis

The analysis of a program required by the rules of the programming language to establish its correctness and to guarantee proper execution

Typical examples are:

a) Static Type Checking:

- Whether the types of operands of a operator are equal?
- Whether the types of the left and right hand side of assignment are equal?
- Whether the types of formal parameters are equal to corresponding real parameters?
- Whether the type of index of array is proper?
- Whether the type of return value is equal to the type of function in definition?

b) Others:

- Has V been declared to be a variable of array type for “V[E]” ?
- Has V been declared to be a variable of record type for “V.i”? Is i a field name of the record?
- Whether an identifier used has been declared? whether an identifier is declared only once?

3 Typical Realization of Semantic Analysis

- Building a symbol table to keep track of the meaning of names established in declarations
- Performing type inference and type checking on expressions and statements to determine their correctness within the type rule of the language

4 Method of Semantic Analysis

- Description

  Attribute grammar is used to describe the semantic
- Implementation

  Syntax-directed semantics analysis

  Semantic content of a program is closely related to its syntax

- Attribute Grammar

  Attribute grammar includes a set of attributes and attribute equations
- Attributes are properties of language entities that must be computed
- Attribute equations (or semantic rules) express how the computation of such attributes is related to the grammar rules of the language

6.1 Attributes and Attribute Grammars

1 Attribute

- Definition
- An attribute is any property of a programming language construct
- Typical examples of attributes are:
  - The data type of a variable
  - The value of an expression
  - The location of a variable in memory
  - The object code of a procedure

2 Attribute Grammars

- Attribute
- Attributes are associated directly with the grammar symbols (terminals and nonterminals)
- If X is a grammar symbol ,and *a* is an attribute associated to X, then the value of *a* associated to X is written as X.*a*

Attribute Equation(or Semantic Rule)

- Given a collection of attributes $a_1 , . . . , a_k$
- For each grammar rule $X_0 \to X_1\ X_2 . . . X_n$, 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
- Attribute equation has the form

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

where f_{ij} is a mathematical function of its arguments

Attribute Grammar

- An attribute grammar for the attributes a1,…,ak is the collection of all attribute equations, for all the grammar rules of the language
- Typically, attribute grammars are written in tabular form

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

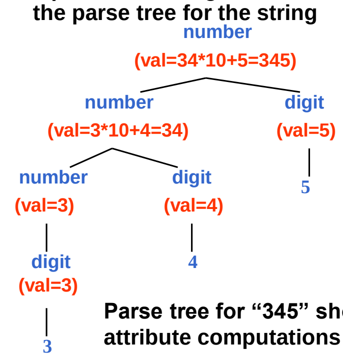
Example 1

Attribute grammar for unsigned numbers

Attribute of a number is its value

Grammar rule	Semantic Rule
number1->number2 digit	number1.val=number2.val*10+digit.val
number->digit	number.val=digit.val
digit->0	digit.val=0
…	…
digit->9	digit.val=9

The meaning of the attribute equations for a particular string can be visualizing using the parse tree for the string



Parse tree for “345” showing attribute computations

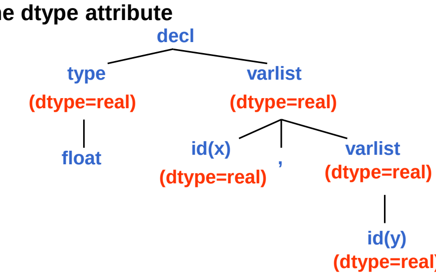
Example 2

Attribute grammar for variable declarations

Attribute of the variable is data type

Grammar rule	Semantic Rules
decl->type varlist	varlist.dtype=type.dtype
type->int	type.dtype=integer
type->float	type.dtype=real
varlist1->id,varlist2	id.dtype=varlist1.dtype varlist2.dtype =varlist1.dtype
var-list->id	id.dtype=varlist.dtype

Parse tree for the string “float x,y” showing the dtype attribute



Example 3

Attribute grammar for type checking of expression

Attribute of expression is its type

Grammar rule	Semantic Rules
$E \to T^1 + T^2$	$T^1.t = int$ AND $T^2.t = int$
$E \to T^1$ or T^2	$T^1.t = bool$ AND $T^2.t = bool$
$T \to num$	$T.t := int$
$T \to true$	$T.t := bool$
$T \to false$	$T.t := bool$

- The kinds of expressions that can appear in attribute equations
  - Arithmetic ,logical,and a few other kinds of expressions
  - If-then-else expression, a case or switch expression
  - Functions whose definitions may be given elsewhere as a supplement to the attribute grammar

6.2 Algorithms for Attribute Computation

- The ways of turning the attribute equations into computation rules
  - Attributes are computed after parse tree has been constructed by a parser(6.2.2)
  - Attributes are computed at the same time as the parsing stage(6.2.3)

- The problem of implementing an algorithm corresponding to an attribute grammar
  - Attribute grammar is an abstract specification, where the attribute equations can be written in arbitrary order without affecting their validity, they don't specify the order of attribute computing

- The problem consists primarily in finding an order for the evaluation and assignment of attributes that ensures that all attribute values used in each computation are available when each computation is performed

Attribute equations themselves indicate the order constraints on the computation of the attributes. We will use dependency graphs to make the order constraints explicit

6.2.1 Dependency Graphs and Evaluation Order

- Dependency Graph
  - Given an attribute grammar, each grammar rule has an associated dependency graph
  - Each attribute $X_i.a_j$ of each symbol corresponds to a node
  - For each attribute equation $X_i.a_j=f_{ij}(\ldots,X_m.a_k,\ldots)$ there is an edge from each node $X_m.a_k$ in the right-hand side to the node $X_i.a_j$(expressing the dependency of $X_i.a_j$ on $X_m.a_k$)

– The dependency graph of a legal string generated by the context-free grammar is the union of the dependency graphs of the grammar rule choices representing each node of the parse tree of the string

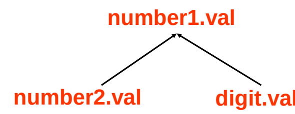
Example 1

Attribute grammar for unsigned numbers

The dependency graph for grammar rule:

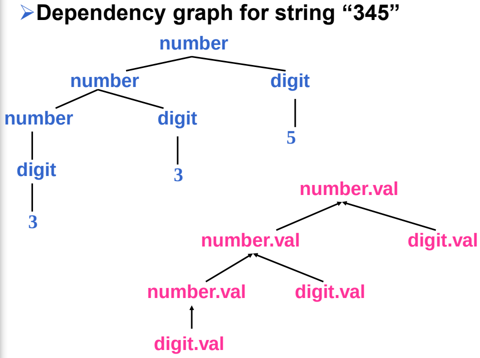
- $number_1$->$number_2$ digit

{number1.val=number2.val*10+digit.val}



- number->digit {number.val=digit.val}

- Dependency graph for string “345”



Example 2

Attribute grammar for variable declarations

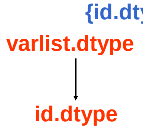
The dependency graph for grammar rule:

- varlist1->id,varlist2

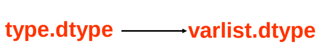
{id.dtype=varlist1.dtype

varlist2.dtype=varlist1.dtype}

- varlist->id {id.dtype=varlist.dtype}

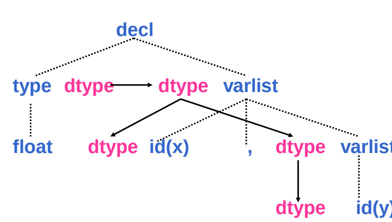


- decl->type varlist {varlist.dtype=type.dtype}



We often draw the dependency graph superimposed over a parse tree segment corresponding to the grammar rule

- The dependency graph for string “float x,y”



6.2.2 Synthesized and Inherited Attributes

- Attribute evaluation depends on an explicit or implicit traversal of the parse tree
- Different kinds of traversals vary in power in terms of the kinds of attribute dependencies that can be handled
- We must classify attributes by the kinds of dependencies they exhibit
  - Synthesized attribute
  - Inherited attribute

1 Synthesized Attribute

- Definition
- An attribute is synthesized if all its dependencies point from child to parent in the parse tree.
- Equivalently, an attribute *a* is synthesized if, given a grammar rule A->X1X2…Xn, the only associated attribute equation with an *a* on the left-hand side is of the form A.a=f(x1.a1,..X1.ak,…,Xn.a1,…Xn.ak)

Example 1

Attribute grammar for unsigned numbers

Grammar rule	Semantic Rule
number1->number2 digit	number1.val=number2.val*10+digit.val
number->digit	number.val=digit.val
digit->0	digit.val=0
…	…
digit->9	digit.val=9

The *val* attribute is synthesized

Example 2

Attribute grammar for simple integer arithmetic expression

Grammar rule	Semantic Rules
$E \to E^1 + T$	$E.val := E^1.val + T.val$
$E \to T$	$E.val := T.val$
$T \to T^1 * F$	$T.val := T^1.val * F.val$
$T \to F$	$T.val := F.val$
$F \to (E)$	$F.val := E.val$
$F \to num$	$F.val := num.val$

The *val* attribute is synthesized

- **S-attributed grammar**

  An attribute grammar in which all attributes are synthesized is called an S-attributed grammar

Example

- Attribute grammar for unsigned numbers
- Attribute grammar for simple integer arithmetic expression

are both S-attributed grammars

- The Evaluation of Synthesized Attributes
- Given that a parse tree or syntax tree has been constructed by a parser
- The synthesized attribute values can be computed by a single bottom-up, or postorder traversal of the tree

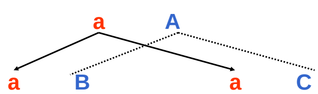
- Express this by the following code

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

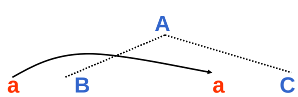
2 Inherited Attribute

- Definition
  - An attribute that is not synthesized is called an inherited attribute
  - Inherited attributes have dependencies that flow either from parent to children in the parse tree or from sibling to sibling

- Two basic kinds of dependency of inherited attributes



(a)Inheritance from parent to siblings



(b)Inheritance from sibling to sibling

Example 1

Attribute grammar for variable declarations

Grammar rule	Semantic Rules
decl->type varlist	varlist.dtype=type.dtype
type->int	type.dtype=integer
type->float	type.dtype=real
varlist1->id,varlist2	id.dtype=varlist1.dtype varlist2.dtype =varlist1.dtype
var-list->id	id.dtype=varlist.dtype

The *dtype* attribute is inherited

- The Evaluation of Inherited Attributes
  - Inherited attributes can be computed by a preorder traversal, or combined preorder/inorder traversal of the parse tree or syntax tree
  - Express this by the following code

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

Note:

- Unlike synthesized attributes, the order in which the inherited attributes of the children are computed is important
- Since inherited attributes may have dependencies among the attributes of the children

Example

Attribute grammar for variable declarations

Grammar rule	Semantic Rules
decl->type varlist	varlist.dtype=type.dtype
type->int	type.dtype=integer
type->float	type.dtype=real
varlist1->id,varlist2	id.dtype=varlist1.dtype varlist2.dtype=varlist1.dtype
var-list->id	id.dtype=varlist.dtype

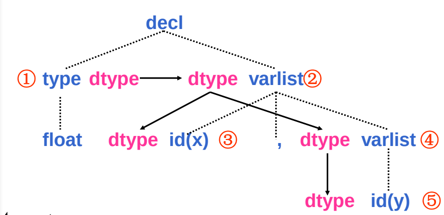
- Assume that a parse tree has been explicitly constructed from the grammar
- A recursive procedure that computes the *dtype* attribute at all required nodes

```
procedure EvalType(T:treenode);
begin
  case nodekind of T of
  decl: //decl->type varlist {varlist.dtype=type.dtype}
      EvalType(type child of T);
      vallist.dtype=type.dtype;
      EvalType(varlist child of T);
  type:     //type->int          {type.dtype=integer}
            //type-> float       {type.dtype=real}
      if child of T=int then T.dtype:=integer
      else T.dtype:=real;
```

```
  varlist:   //varlist->id,varlist{id.dtype=varlist1.dtype
                                   varlist2.dtype=varlist1.dtype}
             //varlist->id{id.dtype=varlist.dtype}
      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;
```

- The parse tree for the string “float x,y” together with the dependency graph for the *dtype* attribute

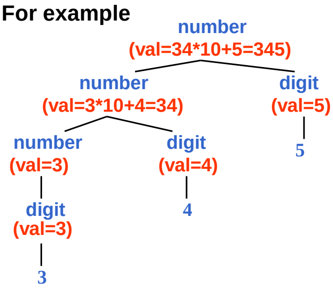
We number the nodes to show the traversal order



3 Storage of Attribute Values

1. Attribute values are stored as fields in the syntax tree nodes

   For example



- If many of the attribute values are the same or are only used temporarily to computer other attribute values
- It makes little sense to use space in the syntax tree to store attribute values at each node

For example

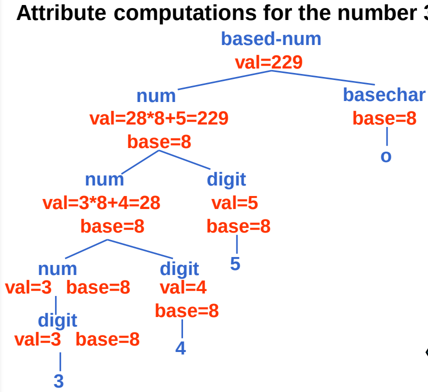
Attribute grammar for octal and decimal numbers

Attributes are value and base

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
num1->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

Grammar rule	Semantic Rules
num->digit	num.val=digit.val digit.base=num.base
digit->0	digit.val=0
…	
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

Attribute computations for the number 345o



2. Attributes as Parameters and Returned Values

- A single recursive traversal procedure that computes inherited attributes in preorder and synthesized attributes in postorder can, in fact,
- pass the inherited attribute values as parameters to recursive calls on children
- and receive synthesized attribute values as returned values of those same calls

For example: attribute grammar for octal and decimal numbers

The base is an inherited attribute and the val is a synthesized attribute

The recursive function for attribute computation is

```
function EvalWithBase(T:treenode;base:int):int
var temp,temp2:int;
begin
    case nodekind of T of
    base-num: //based-num->num basechar
        temp:=EvalWithBase(right child of T);
        return EvalWithBase(left child of T,temp);
```

```
num: //num1->num2 digit      num->digit
    temp:=EvalWithBase(left child of T,base);
    if right child of T is not nill 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: //basechar->o | d
    if child of T=o then return 8
    else return 10;
digit: //digit->0 |1 |…|9
    if base=8 and child of T=8 or 9 then error
    else return numval(child of T);
end case;
end EvalWithBase
```

3. The use of External Data Structures to Store Attribute Values

- When attribute values have significant structure and may be needed at arbitrary points during translation
- Data structures such as lookup tables, graphs and other structures may be useful to obtain the correct behavior and accessibility of attribute values
- One of the prime data structure is the symbol table, which stores attributes associated to declared constants, variables, and procedures in a program

6.2.3 The Computation of Attributes During Parsing

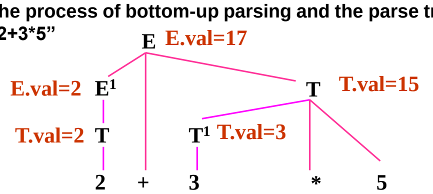
Attributes can be computed at the same time as the parsing stage, without waiting to perform further passes over the source code by recursive traversals of the syntax tree

Example

Attribute grammar for arithmetic expression

1) $E \rightarrow E^1 + T$ { E.val := E^1.val +T.val }
2) $E \rightarrow T$ { E.val := T.val }
3) $T \rightarrow T^1$*number { T.val := T^1.val * number.val }
4) $T \rightarrow$ number { T.val := number.val }

The process of bottom-up parsing and the parse tree for "2+3*5"



When the parsing completes, the attribute value is also computed

1 The Need for Attributes to be Computed During the Parse

- Syntax tree itself is a synthesized attribute that must be constructed during the parse
- Compilers that perform one-pass translation compute all attributes during the parse

2 Which Attributes can be Computed During the Parse?

- It depends on the power and properties of the parsing method employed.
- All the major parsing methods process the input program form left to right
- This is equivalent to the requirement that the attributes be capable of evaluation by a left-to-right traversal of the parse tree
- For synthesized attributes this is not a restriction, since the children of a node can be processed in arbitrary order

- But, for inherited attributes, this means that there may be no "backward" dependencies(dependencies pointing from right to left in the parse tree) in the dependency graph
- Attribute grammars that do satisfy this property are called L-attributed(Left to right)

- **L-attributed Grammar**
  - **Definition**

An attribute grammar for attributes a1,..,ak is L-attributed if, for each inherited attribute aj and each grammar rule

X0->X1X2…Xn

the associated equations for a_j are all of the form

$X_i.a_j=f_{ij}(X_0.a_1,\ldots,X_0.a_k,X_1.a_1,\ldots X_1.a_k,\ldots X_{i-1}.a_1,\ldots X_{i-1}.a_k)$

- **That is,the value of a_j at X_i can only depend on attributes of the symbols $X_0,\ldots,X_{i-1}$ that occur to the left of X_i in the grammar rule**
- **As a special case, an S-attributed grammar is L-attributed**

3 The Computation of Attribute During the Parse

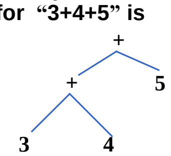
- **Recursive-descent parsers can evaluate all the attributes by turning the inherited attributes into parameters and synthesized attributes into returned values**
- **LR parsers are suited to handling primarily synthesized attributes.**

- **Attributes computation in recursive-descent parser**

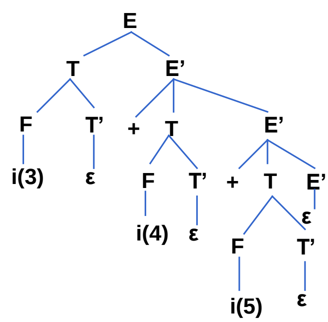
Example

E→TE'
E'→+TE'│ε
T→FT'
T'→*FT'│ε
F→(E)│i

The syntax tree for "3+4+5" is



The parse tree for "3+4+5" is



Grammar rule	Semantic Rules
E->TE'	E.tree=E'.tree E'.left=T.tree
E'_1->+TE'_2	E'_1.tree=E'_2.tree E'_2.left=mkOpNode(+,E'_1.left,T.tree)
E'->ε	E'.tree=E'.left
T->FT'	T.tree=T'.tree T'.left=F.tree
T'_1->*FT'_2	T'_1.tree=T'_2.tree T'_2.left=mkOpNode(*,T'_1.left,F.tree)
T'->ε	T'.tree=T'.left
F->(E)	F.tree=E.tree
F->i	F.tree=mkNumNode(i.lexval)

```
//E→TE'    {E.tree=E'.tree    E'.left=T.tree}
function E:syntaxTree;
var temp:syntaxTree;
begin
    temp:=T;
    return E'(temp);
end;
```

```
function E'(treesofar:syntaxTree):syntaxTree
var temp: syntaxTree;
begin
  if TOKEN='+' then //E'1->+TE'2      {E'1.tree=E'2.tree
     begin              E'2.left=mkOpNode(+,E'1.left,T.tree)}
         temp:=makeOpNode('+');
         match('+');
         leftChild(temp):=treesofar;
         rightChild(temp):=T;
         return E'(temp)
     end;
  else            //E'->ε       {E'.tree=E'.left}
     begin
         if TOKEN≠')' and TOKEN≠'$' then ERROR;
         return treesofar;
     end;
end;
```

Computing Synthesized Attributes During LR Parsing

- Adds a value stack in which the synthesized attributes are stored
- The value stack will be manipulated in parallel with the parsing stack
- As reductions occur on the parsing stack, computations occur on the value stack according to the attribute equations
- Shifts are viewed as pushing the token values on both the parsing stack and the value stack

Example :Attribute grammar for arithmetic expression

1) L →E
{print(E.val)}
2) E→E^1+T
{ E.val :=E^1.val +T.val }
3) E→T
{ E.val :=T.val }
4) T→T^1*num
{T.val :=T^1.val * num.val }
5) T→num
{ T.val :=num.val }

	ACTION				GOTO	
	n	+	*	#	E	T
0	S_3				1	2
1		S_4		acc		
2		r3	S_5	r_3		3
3		r_5	r_5	r_5		
4	S_3					7
5	S_6					
6		r_4	r_4	r_4		
7		r_2	S_5	r_2		

The parsing process of “2+3*5”

	paring stack	value stack	input	Action	GOTO
0	$0	$	2+3*5$	S_3	
1	$0 2 3	$2	+3*5$	r_5	2
2	$ 0 T 2	$2	+3*5$	r_3	1
3	$ 0 E 1	$2	+3*5$	S_4	
4	$ 0 E 1 + 4	$2+	3*5$	S_3	
5	$ 0 E 1 + 4 3 3	$2+3	*5$	r_5	7
6	$ 0 E 1 + 4 T 7	$2+3	*5$	S_5	
7	$ 0 E 1 + 4 T 7 *5	$2+3*	5$	S_6	
8	$ 0 E 1 + 4 T 7 *556	$2+3*5	$	r_4	7
9	$ 0 E 1 + 4 T 7	$2+15	$	r_2	1
10	$ 0 E 1	$17	$	acc	

Technique for dealing with inherited attributes in LR parsing

- Inheriting a previously computed synthesized attribute 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 Example

Production A->BC

- C has an inherited attribute i that depends on the synthesized attribute s of B: C.i=f(B.s)
- C.i can be stored in a variable prior to the recognition of C by introducing an ε-production between B and C that schedules the storing of the top of the value stack

Grammar rule	Semantic Rules
A->BDC	
B->…	{compute B.s}
D-> ε	saved_i=f(valstack[top])
C->…	{now saved_i is available}

For Example

Grammar rule	Semantic Rules
E->TE’	E.tree=E’.tree E’.left=T.tree
E'_1->+TE’$_2$	E'_1.tree=E'_2.tree E'_2.left=mkOpNode(+,E'_1.left,T.tree)
…	

E’.left is an inherited attribute that depends on the synthesized attribute T.tree

Grammar rule	Semantic Rules
E->TXE’	E.tree=E’.tree E’.left=T.tree
X-> ε	saved_T=valstack[top]
…	

- Technique for dealing with inherited attributes in LR parsing is
- To use external data structures, such as a symbol table or nonlocal variables, to hold inherited attribute values
- And to add ε-productions to allow for changes to these data structures to occur at appropriate moments

6.3 The Symbol Table

- **Function of Symbol Table**

  Symbol table stores attributes associated to declared constants, variables and procedures in a program

- **The principal symbol table operations:**
- **Insert**

  It is used to store the information provided by name declarations when processing these declarations
- **Lookup**

  It is needed to retrieve the information associated to a name when that name is used in the associated code
- **Delete**

  It is needed to remove the information provided by a declaration when that declaration no longer applies

- **What information needs to be stored in the symbol table:**
- Data type information
- Information on region of applicability(scope)
- Information on eventual location in memory

Example :Declaration part of a program

CONST A=35 B

VAR C,D E

PROC

V

Information of the symb

Variables need to be allocated memory locations, they are viewed simply as integer indices that are incremented each time a new variable is encountered

NAME:A	Kind :CONSTANT	VAL:35		
NAME:B	Kind :CONSTANT	VAL:49		
NAME:C	Kind :VARIBALE	LEVEL:LEV	ADR: DX	
NAME:D	Kind :VARIBALE	LEVEL:LEV	ADR: DX+1	
NAME:E	Kind :VARIBALE	LEVEL:LEV	ADR: DX+2	
NAME:P	Kind :PROCEDUR	LEVEL:LEV	ADR:	SIZE:4
NAME:G	Kind :VARIBALE	LEVEL:LEV+1	ADR: DX	
…	…	…	…	

6.4 Semantic Analysis of a Program

1. Declarations

- Typically, the information in declarations is inserted into a symbol table for later lookup during the translation of other parts of the program
- Assume that inset(id.name,dtype) is a procedure that insets an identifier into the symbol table and associates a type to it
- Attribute grammar is as follow:

Grammar rule	Semantic Rules
decl->type varlist	varlist.dtype=type.dtype
type->int	type.dtype=integer
type->float	type.dtype=real
varlist1->id,varlist2	insert(id.name,varlist1.dtype) varlist2.dtype =varlist1.dtype
var-list->id	insert(id.name,varlist.dtype)

2. Statements

- Semantic analysis of statements is mainly type checking (the use of type information to ensure that each part of a program makes sense under the type rules of the language)
- A simple grammar to illustrate semantic analysis

  stmt->id:=exp

  stmt->if exp then stmt

  exp-> exp1+exp2

  exp->exp1 or exp2

  exp-> id

- Attributes and Procedures used in attribute grammar

  we assume the availability of a symbol table that contains variable names and associated types

  Attribute:

  - name of an identifier
  - type of grammar symbol

  Procedures:

  - lookup(id.name), which returns the associated type of a name if it has already in the symbol table, otherwise returns nil
  - error, which reports semantic errors

- Attribute grammar for semantic analysis of simple grammar

Grammar rule	Semantic Rules
exp->exp1+exp2	if exp1.type≠integer or exp2.type≠integer then error else exp.type=integer
exp->exp1 or exp2	if exp1.type ≠boolean or exp2.type ≠boolean then error else exp.type=boolean
exp->id	t=lookup(id.name) if t ≠nil then exp.type=t else error

Grammar rule	Semantic Rules
stmt->id:=exp	t=lookup(id.name) if t=nil then error else if t ≠ exp.type then error
stmt->if exp then stmt	if exp.type ≠ boolean then error