Semantic Analysis

Scope Checking

Two topics

- Scoping
- Symbol table interface
- Type checking: updates

Symbol table – a reminder

- Stores info on attributes (type, role, etc.) for every object defined in the compiled program

- Info is collected and entered when an object declaration is analyzed

- Info is retrieved when object's use is analyzed (in expressions and commands)

Familiar semantic mistake

- Duplicate declaration of the same name

  void main() {int a, b; real a; … }

- **Forbidden**: two entries with identical names in the same symbol table

- When object is used, it is searched in the table by its name;

  $\geq$ 1 entries with same name cause a **conflict**

Another familiar concept - local objects

- **Allow** objects with same name

- But: in different **scopes**
  (e.g. in different functions)

Scope (תחום הגדרה)

Section of program (block) in which use of a declared object (e.g. variable) is valid

Syntax:

- Block is enclosed by special start and finish delimiters

  examples: { … } **begin … end**

Scope (תחום הגדרה)

- Blocks can be nested

  example:

```
{
    int a;
    a = 1;
    {
        int b;
        real c;
        c = 2.3;
        b = a + 4
      } ;
      a = a *3
}
```

- Block is a type of command – can appear in any place where command is expected

Scope (תחום הגדרה)

Semantics (where a declared object is known / visible?):

- ***global***: can be used / visible anywhere in the program
- ***local***: visible within certain section only

```
int a = 9;      /* global a */
real b = 3.14;  /* global b */
void my(int b)  /* parameter  b  is local in the function */
{
    int a = 8;    /* local a */
    real c;       /* local c */
     ...
    a = a + 5;            /*  a  gets value 14?  or 13?   Which a is meant here ?   */
    c = b * 2.4;
  }
```

Challenges

Given a use of object X, how to know which declaration of X is relevant?

```
int a = 9;
real b = 3.14;
void my(int b)
{
   int a = 8;
   real c;
    ...
   a = a + 5;          /*  a  gets value 14?  or 13?   Which  a  is meant here ?   */
   c = b * 2.4;
 }
```

In most programming languages: use innermost declared *a*.

So, in our case it is the local *a* ; hence *a* gets value 13.

Challenges

- How to implement scope checking ?

- How to avoid conflicts in symbol table ?

Solution

- Individual symbol table for each scope / block.

- Tables are organized in a hierarchical way,
  to reflect the hierarchy of the nested blocks

- When block BL is entered:
  - a new table is created for BL; it becomes the current table
  - objects declared in BL are stored in BL's table

- When leaving block BL:
  - return to father-block of BL
  - its table becomes the current one

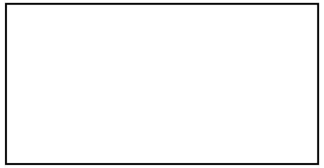
Scoping rules by example

$P \rightarrow L \ S$
$L \rightarrow L \ D$
$L \rightarrow \varepsilon$
$D \rightarrow T \ N \ ;$
$N \rightarrow N \ , \ id$
$N \rightarrow id$
$T \rightarrow real$
$T \rightarrow int$
$S \rightarrow S \ C$
$S \rightarrow \varepsilon$
$C \rightarrow id := E \ ;$
$C \rightarrow \{L \ S\}$
$E \rightarrow E + id$
$E \rightarrow id$

Sample program and hierarchy of scopes

```
int x, y;
x := y ;
{
  real x, w ;
  x := x + w ;
}
y := x + y ;
{
  int y , w ;
  x : = w + y ;
    {
      int a ;
      a := x ;
    }
}
```

Sample program and hierarchy of scopes



Sample program and hierarchy of scopes

```
int x, y;
```

int x int y

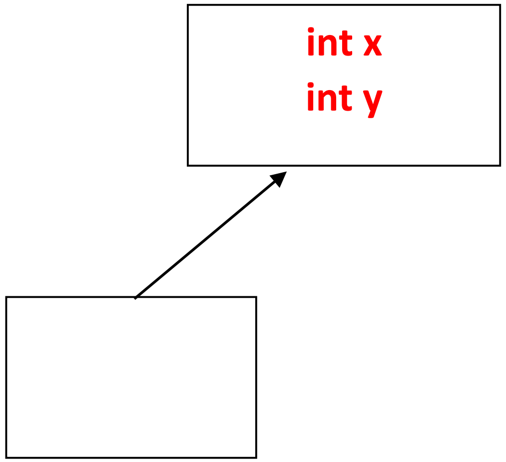
Sample program and hierarchy of scopes

```
int x, y;
x := y ;
```

int x
int y

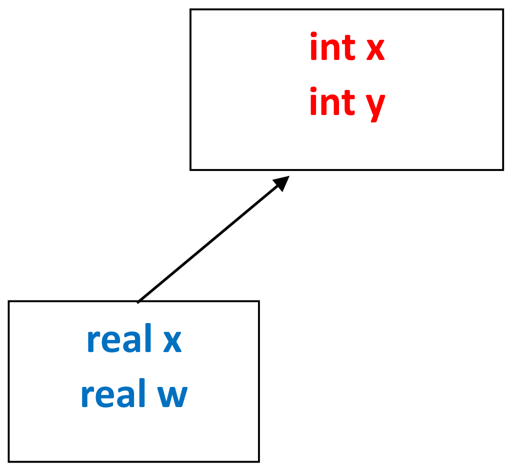
Sample program and hierarchy of scopes

```
int x, y;
x := y ;
{
```



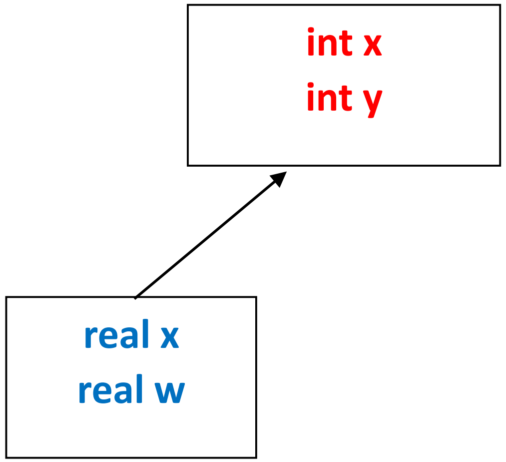
Sample program and hierarchy of scopes

```
int x, y;
x := y ;
{
  real x, w ;
```



Sample program and hierarchy of scopes

```
int x, y;
x := y ;
{
  real x, w ;
  x := x + w ;
```



Sample program and hierarchy of scopes

```
int x, y;
x := y ;
{
  real x, w ;
  x := x + w ;
}
```

int x
int y

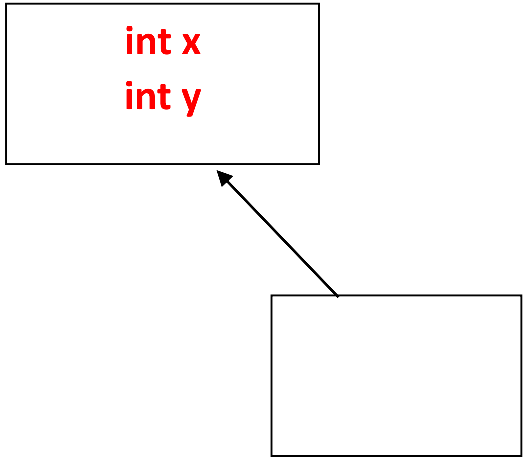
Sample program and hierarchy of scopes

```
int x, y;
x := y ;
{
  real x, w ;
  x := x + w ;
}
y := x + y ;
```

int x
int y

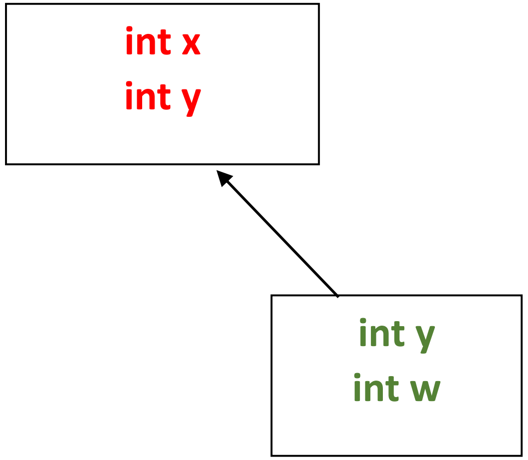
Sample program and hierarchy of scopes

```
int x, y;
x := y ;
{
  real x, w ;
  x := x + w ;
}
y := x + y ;
{
```



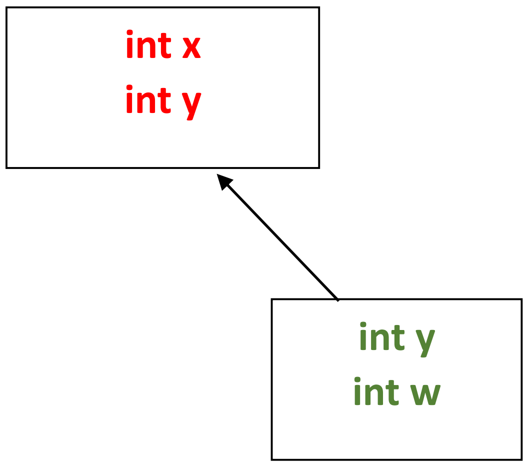
Sample program and hierarchy of scopes

```
int x, y;
x := y ;
{
  real x, w ;
  x := x + w ;
}
y := x + y ;
{
  int y , w ;
```



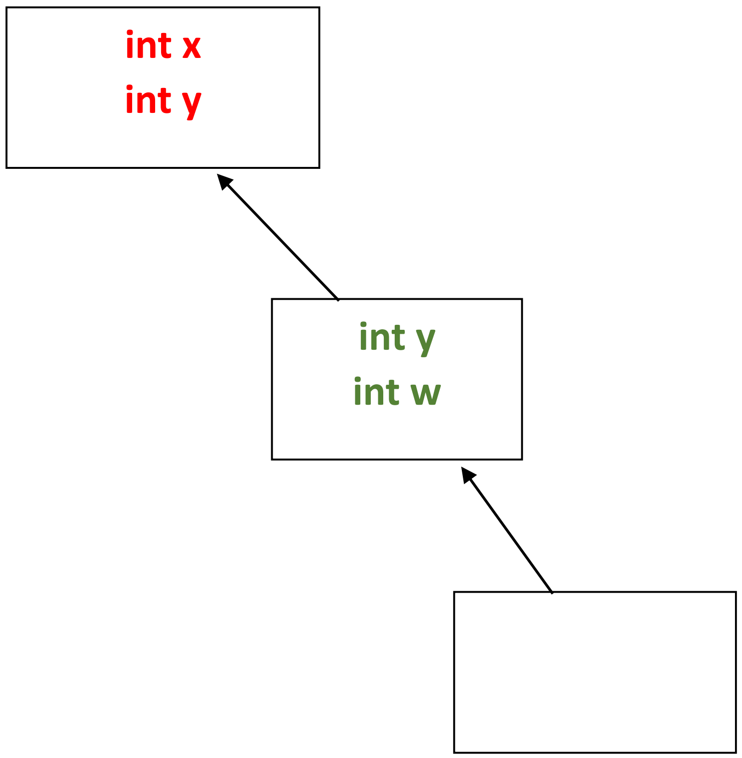
Sample program and hierarchy of scopes

```
int x, y;
x := y ;
{
  real x, w ;
  x := x + w ;
}
y := x + y ;
{
  int y , w ;
  x : = w + y ;
```



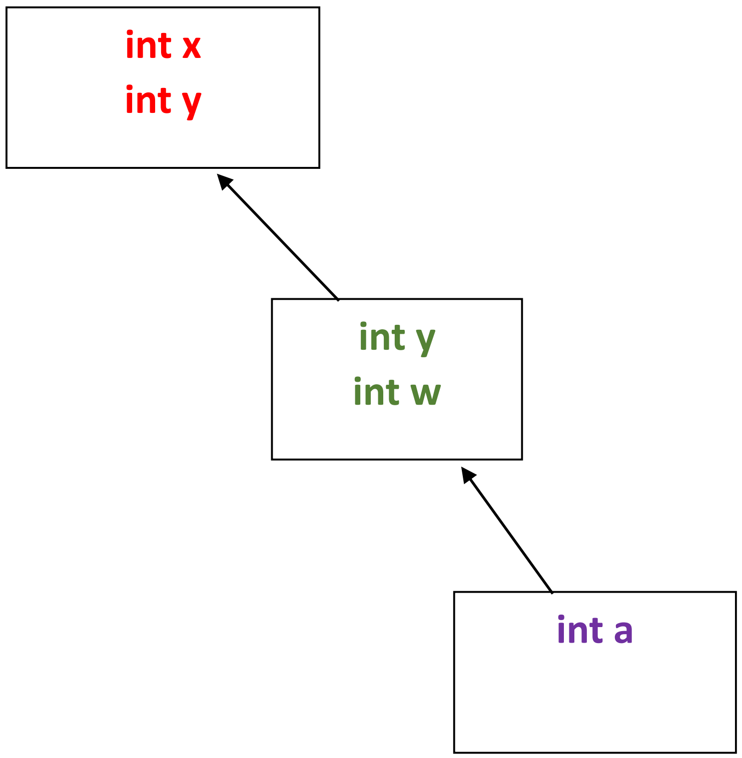
Sample program and hierarchy of scopes

```
int x, y;
x := y ;
{
  real x, w ;
  x := x + w ;
}
y := x + y ;
{
  int y , w ;
  x : = w + y ;
    {
```



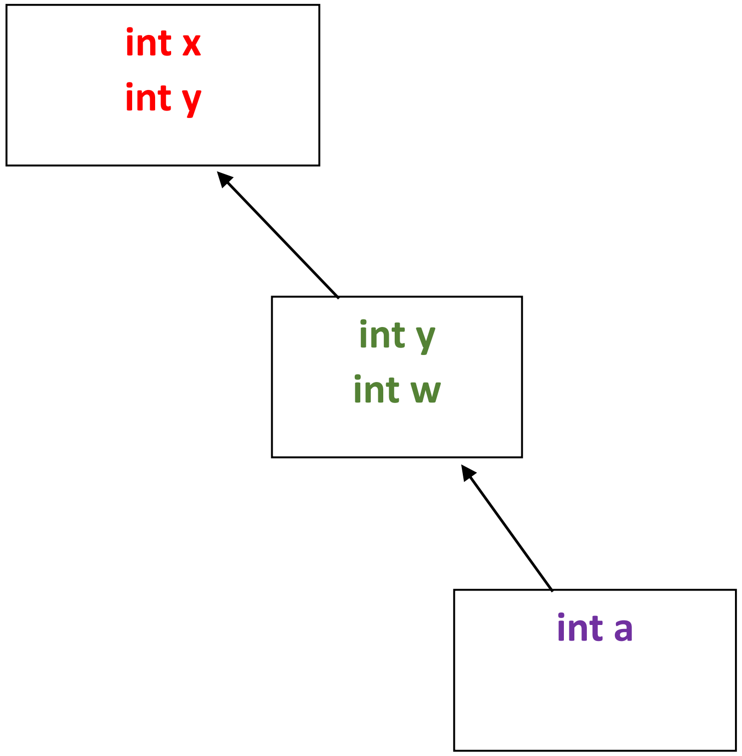
Sample program and hierarchy of scopes

```
int x, y;
x := y ;
{
  real x, w ;
  x := x + w ;
}
y := x + y ;
{
  int y , w ;
  x : = w + y ;
   {
     int a ;
```



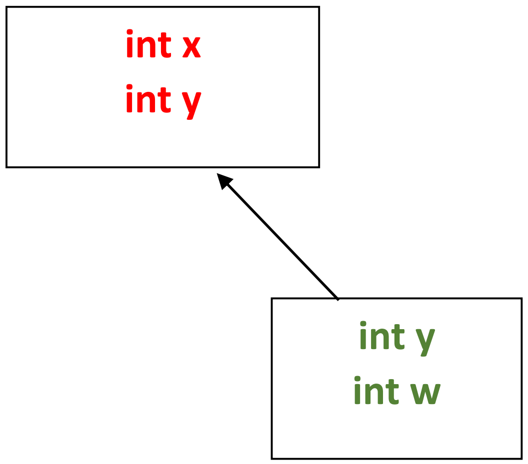
Sample program and hierarchy of scopes

```
int x, y;
x := y ;
{
  real x, w ;
  x := x + w ;
}
y := x + y ;
{
  int y , w ;
  x : = w + y ;
    {
      int a ;
      a := x ;
```



Sample program and hierarchy of scopes

```
int x, y;
x := y ;
{
  real x, w ;
  x := x + w ;
}
y := x + y ;
{
  int y , w ;
  x : = w + y ;
   {
     int a ;
     a := x ;
   }
```



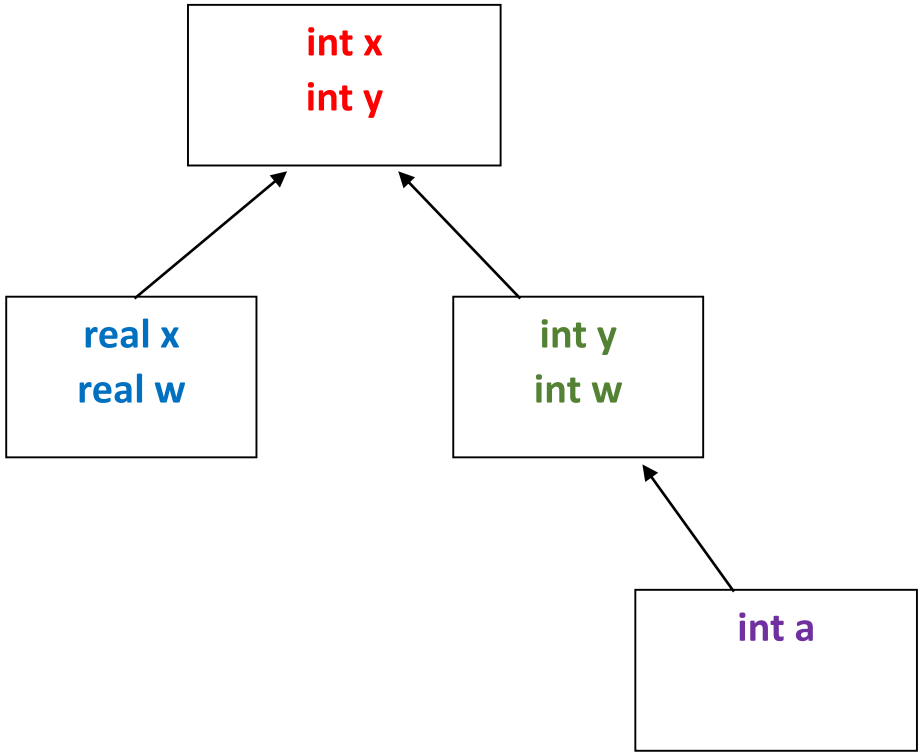
Sample program and hierarchy of scopes

```
int x, y;
x := y ;
{
  real x, w ;
  x := x + w ;
}
y := x + y ;
{
  int y , w ;
  x : = w + y ;
   {
     int a ;
     a := x ;
   }
}
```

int x
int y

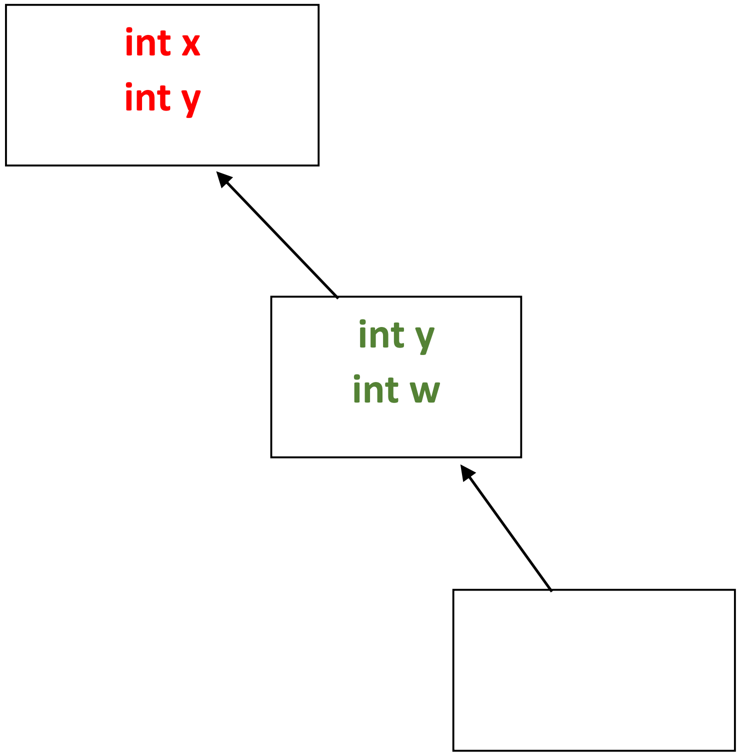
Sample program and hierarchy of scopes

```
int x, y;
x := y ;
{
  real x, w ;
  x := x + w ;
}
y := x + y ;
{
  int y , w ;
  x : = w + y ;
  {
    int a ;
    a := x ;
  }
}
```



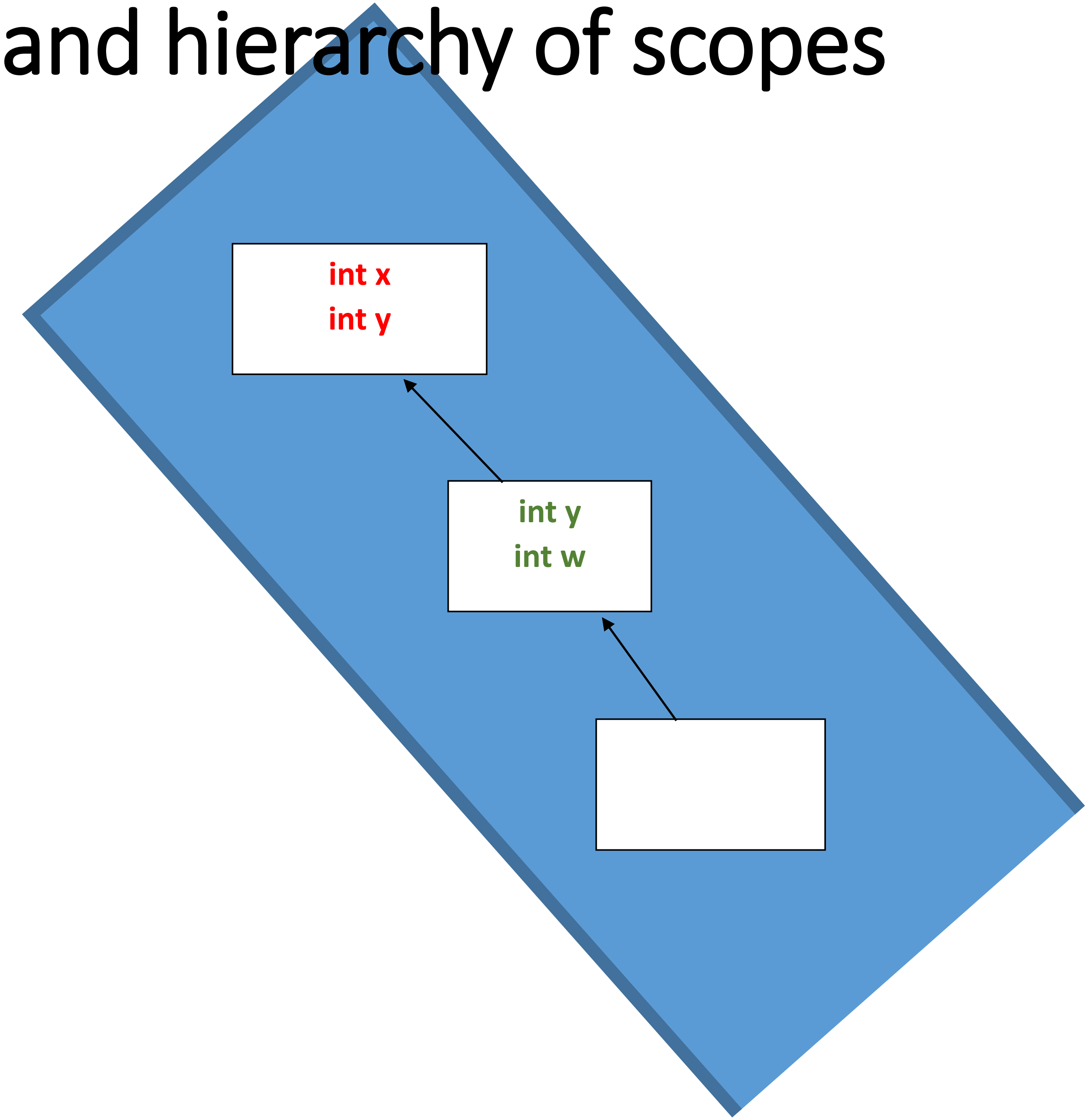
Sample program and hierarchy of scopes

```
int x, y;
x := y ;
{
  real x, w ;
  x := x + w ;
}
y := x + y ;
{
  int y , w ;
  x : = w + y ;
    {
```



Sample program and hierarchy of scopes

```
int x, y;
x := y ;
{
  real x, w ;
  x := x + w ;
}
y := x + y ;
{
  int y , w ;
  x : = w + y ;
   {
```



During the analysis: stack of tables is used

- <u>When block BL is entered:</u>

  - a new table is created for BL, and placed on top of the stack

- <u>When leaving block BL:</u>

  - return to father-block of BL

  - pop BL’s table from the stack

Scoping rules – summary

When object X is defined in scope BL:

- it is inserted into the table of that scope BL
- requires a local search in BL's table

  (to avoid duplicated definition of X in BL)

When object X is used in scope BL:

- its declaration is searched in the hierarchy of scopes
- requires a hierarchical search

  starts in BL's table; if not found – continue to the table of BL's father (deeper into the stack), etc.

Symbol table interface

- For efficiency, implemented using HASH TABLES…
- The drawings of “tables” is abstract…

FATHER	
NAME	TYPE
x	int
y	int

```c
table_ptr make_table (table_ptr current_table)
{
  table_ptr tab;
  tab = (table_ptr)malloc(sizeof(table));
  tab -> father = current_table;
  return tab;
}
```

- called when a nested block is entered:

  c_table = make_table(c_table)
- creates symbol table for this block
- links it to the table of the current block

c_table

FATHER	
NAME	TYPE
x	int
y	int

int x, y ;

table_ptr **make_table** (table_ptr current_table)

- called when a nested block is entered:

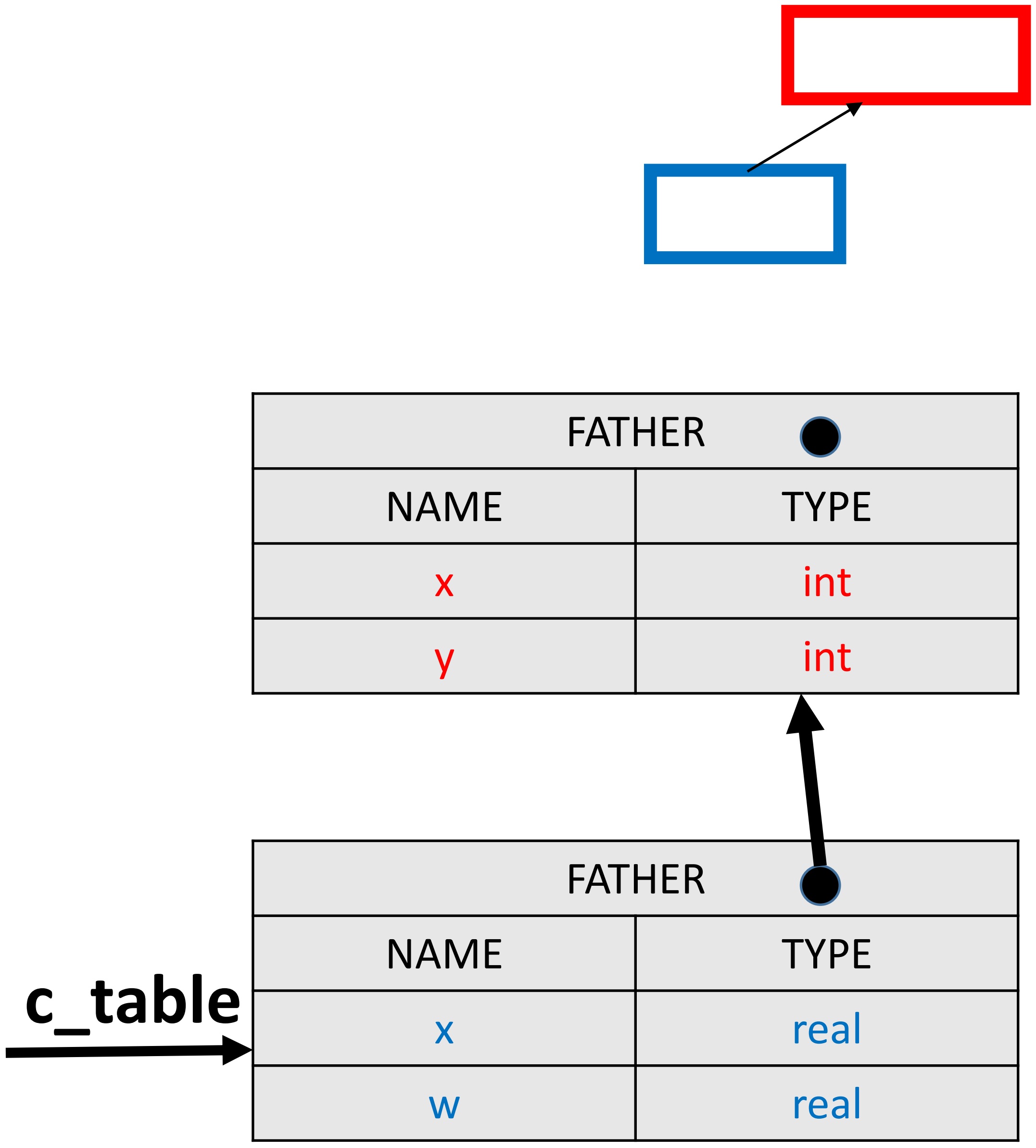
  c_table = make_table(c_table)
- creates symbol table for this block
- links it to the table of the current block

int x, y ;

x := y ;

{

    real x, w ;



```
table_ptr pop_table (table_ptr current_table)
{
  return (current_table -> father);
}
```

called when a block is exited:

c_table = pop_table(c_table)

returns to the father-block of the current block

```
int x, y ;
x := y ;
{
   real x, w ;
   x := x + w ;
}
y := x + y ;
```

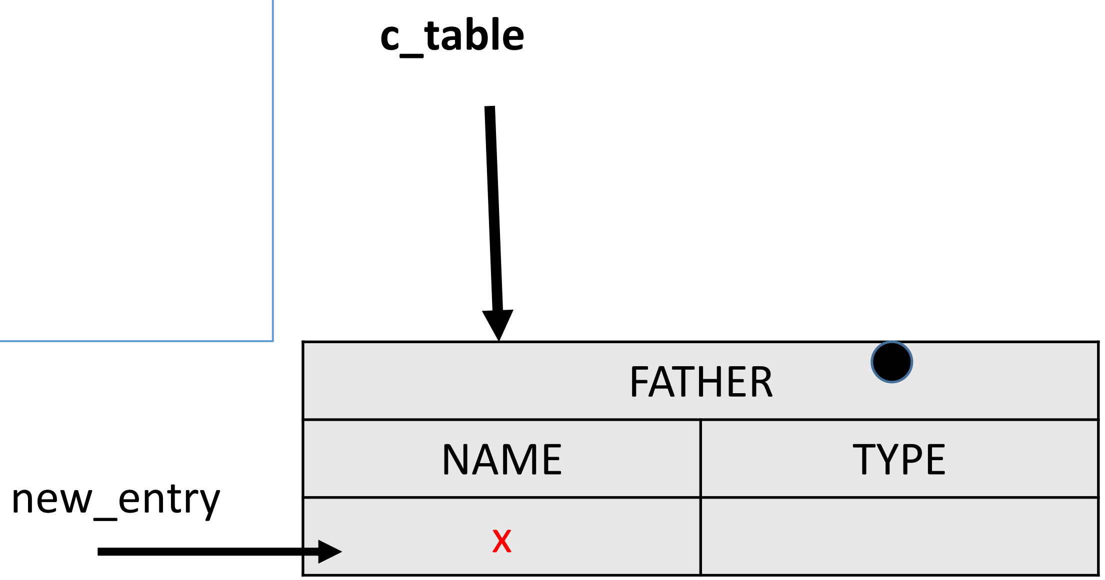
c_table

FATHER	
NAME	TYPE
x	int
y	int

```
table_entry insert (table_ptr current_table , char* id_name)
{
  table_entry etr;
  etr = lookup(table_ptr current_table , char* id_name)    /* local search! */
  if (etr != NULL)
     { ERROR ("Duplicated declaration of %s" , id_name) ; return NULL }
  else return (create_new_entry(current_table));
}
```

called when a variable declaration is processed
if entry for id_name already exist in current_table
  then ERROR ; returns NULL
  else creates a new entry in the table for id_name;
      returns pointer to this entry
Example: new_entry = insert(c_table, "x")

int x, y ;



table_entry **lookup** (table_ptr current_table , char* id_name)

- performs a local search of id_name in the current_table

  (checks whether id_name is already declared in the current block)
- returns pointer to the found entry, or NULL if the name is not found in that table

int x, y ;

<u>Example:</u>

lookup (c_table, “y”)

this call returns NULL

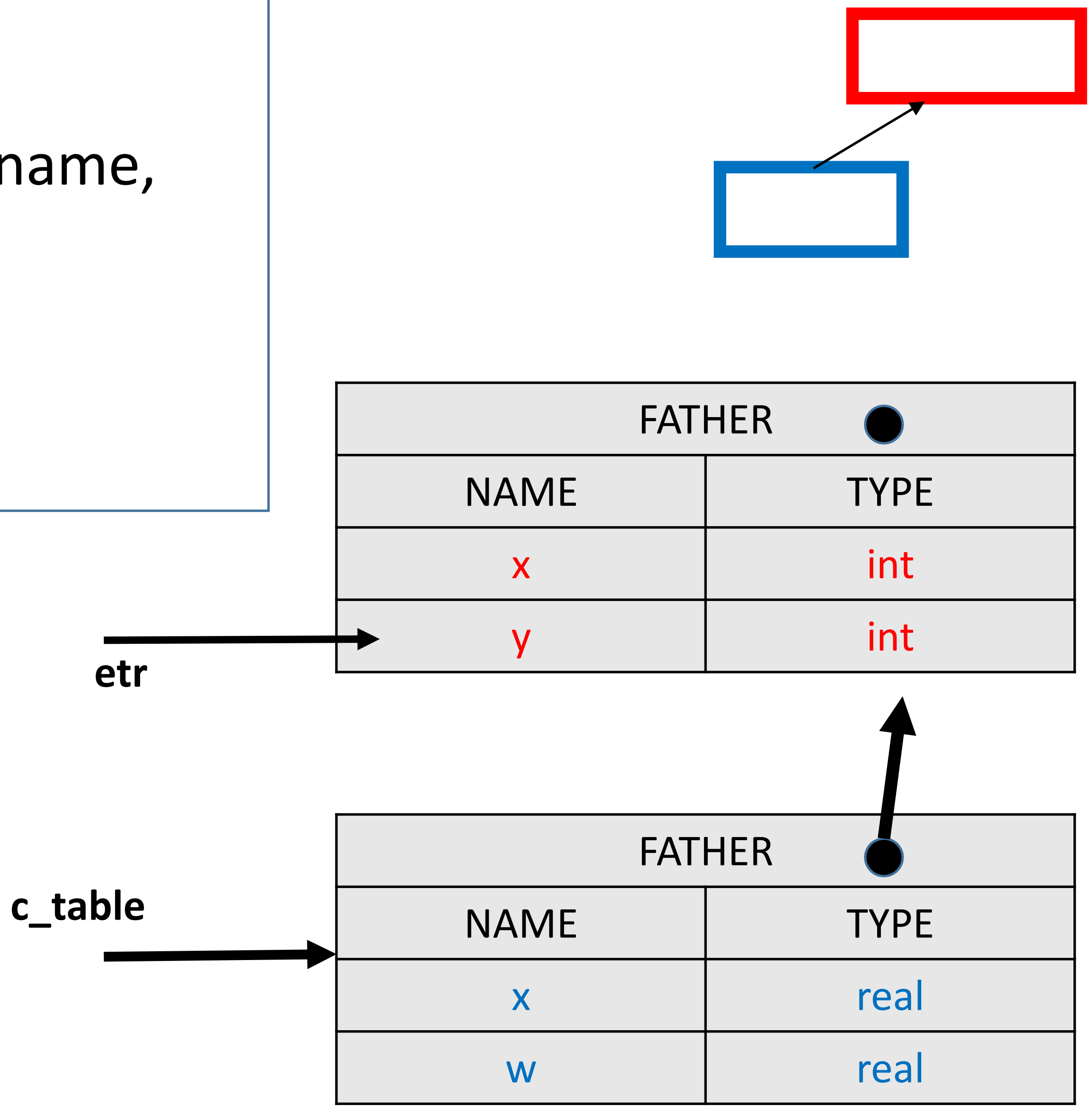
c_table →

FATHER	
NAME	TYPE
x	int

table_entry **find** (table_ptr current_table , char* id_name)

- called when id use is found
- allows to check whether the id is declared
- for this, performs a hierarchical search of id_name, starting from the current_table
- returns the found entry pointer, or NULL (if id_name is undeclared)

<u>Example:</u> etr = find(c_table , “y”)



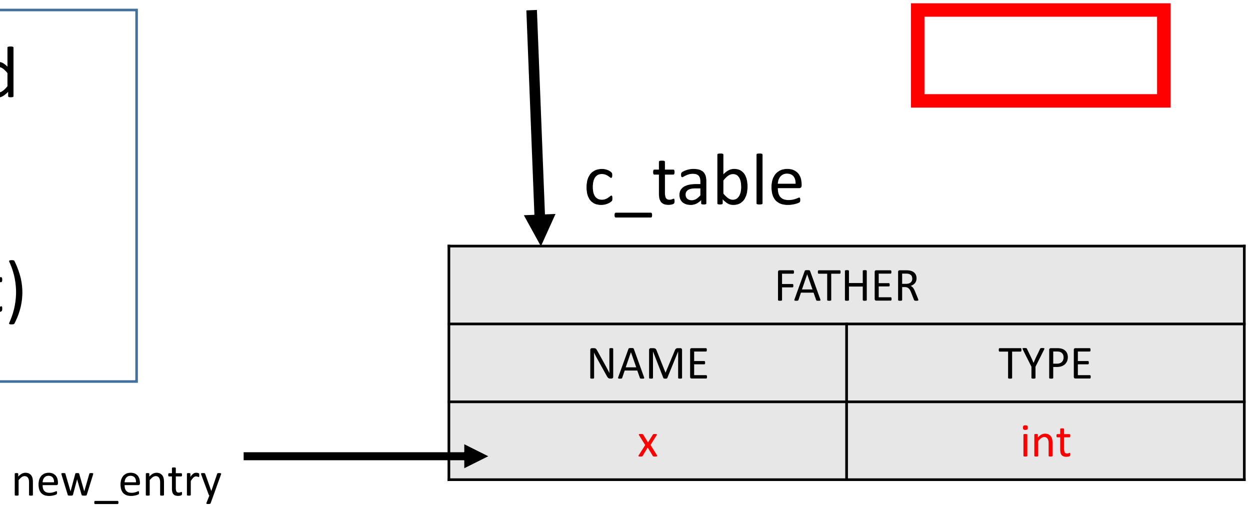

```
int x, y ;
x := y ;
{
  real x, w ;
  x := x + y ;
```

```
# typedef {integer, real, error_type, } elm_type ;

table_entry find (table_ptr current_table, char* id_name)
{
     /* hierarchical search is implemented as a series of local searches */
    table_ptr tab = current_table;
    while ( tab != NULL )
    {
          id_entry = lookup(tab, id_name);
          if (id_entry != NULL)
               return (id_entry);
          else
               tab = tab->father;
     }
     printf ("ERROR: undeclared identifier %s \n", id_name);
     return NULL;
}
```

void **set_id_type** (table_entry id_entry, elm_type id_type)

- called when id declaration is processed
- stores id’s type in the symbol table
- <u>example</u>:   set_id_type(new_entry , int)



elm_type **get_id_type** (table_entry id_entry)

- called when id’s use is processed
- returns variable's type (integer or real)
- <u>example</u>:   id_type = get_id_type(id_entry)

Scoping rules by example

$P \rightarrow L\ S$
$L \rightarrow L\ D$
$L \rightarrow \varepsilon$
$D \rightarrow T\ N\ ;$
$N \rightarrow N\ ,\ id$
$N \rightarrow id$
$T \rightarrow real$
$T \rightarrow int$
$S \rightarrow S\ C$
$S \rightarrow \varepsilon$
$C \rightarrow id := E\ ;$
$C \rightarrow \{L\ S\}$
$E \rightarrow E + id$
$E \rightarrow id$

Type checking – updated scheme with scoping

Attributes

- T.type and E.type (synthesized)
- N.type (inherited)

Modified grammar

- Replacement of rules that cause creation of a new scope:

P → L S	⇒	P → BB L S FB
C → {L S}		C → {BB L S FB}
		BB → epsilon
		FB → epsilon

BB – begin block

FB – finish block

Scoping rules by example

P → BB L S FB

BB → ε

FB → ε

L → L D

L → ε

D → T N ;

N → N , id

N → id

T → real

T → int

S → S C

S → ε

C → id := E ;

C → {BB L S FB}

E → E + id

E → id

Derivation rule	Semantic action
P → ① BB L S FB	① **cur_table** = NULL; // משתנה גלובלי cur_table
BB → ε ①	① **cur_table** = make_table (**cur_table**)
FB → ε ①	① **cur_table** = pop_table (**cur_table**)

Derivation rule	**Semantic action**
L → L D	
L → ε	
D → T ① N ;	① **N**.type = **T**.type
T → real ①	① **T**.type = real
T → int ①	① **T**.type = integer

<table>
<tr><th>Derivation rule</th><th colspan="2">Semantic action</th></tr>
<tr><td>N → id ①</td><td colspan="2">① id_table_entry = insert (cur_table, id.name);
if (id_table_entry != NULL)
then set_id_type (id_table_entry, N.type);</td></tr>
<tr><td rowspan="2">N → ① N1 , id ②</td><td colspan="2">① N1.type = N.type</td></tr>
<tr><td colspan="2">② id_table_entry = insert (cur_table, id.name);
if (id_table_entry != NULL)
then set_id_type (id_table_entry, N.type);</td></tr>
</table>

Derivation rule	**Semantic action**
E → id ①	① id_table_entry = find (**cur_table**, **id**.name); //חיפוש היררכי ***if*** (id_table_entry != NULL) ***then*** **E**.type = get_id_type (id_table_entry); ***else*** **{** **E**.type = NULL_type; // error-type printf (“ERROR: undeclared id %s … … ..”, **id**.name); **}**

<table>
<tr><th>Derivation rule</th><th>Semantic action</th></tr>
<tr><td>E → E1 + id ①</td><td><pre>① if ((E1.type == NULL_type))   then E.type = NULL_type
   else {
           id_table_entry = find (cur_table, id.name);   //חיפוש היררכי
           if (id_table_entry != NULL)   then {
                id_type = get_id_type (id_table_entry);
                if ((id_type == integer) && (E1.type == integer))
                        then E.type = integer ;
                else E.type = real;   // legal types are either int or real
           }
           else   {
                   E.type = NULL_type;
                printf (“ERROR: undeclared id %s”, id.name);
           }
   }</pre></td></tr>
</table>

<table>
<tr><th>Derivation rule</th><th>Semantic action</th></tr>
<tr><td>S → S C</td><td></td></tr>
<tr><td>S → ε</td><td></td></tr>
<tr><td>C → { BB L S FB}</td><td></td></tr>
<tr><td>C → id := E ① ;</td><td>① id_table_entry = find (cur_table, id.name); //חיפוש היררכי
if (id_table_entry != NULL)
then {
id_type = get_id_type (id_table_entry);
if (id_type != E.type)
then printf (“ERROR: type missmatch … … ..”, ….);
}
else printf (“ERROR: undeclared id %s”, id.name);</td></tr>
</table>