# Static Semantic Analysis

After syntax analysis we would like to do the following:

1. Declaration processing and symboltable creation: Process the declarative part of the program an record the information that it contains in a structure called a SymbolTable. Information is recorded regarding:
   - -Types and sizes of local variables and parameters, their offsets in the activation record layout.
   - - return types of functions and their nesting depth.

2. Use this information for type checking and overloading resolution:
   - - Are the LHS and the RHS of an assignment "type compatible"?
   - - Are the operands of an operation compatible with an operator? How do the operands have to be typecast? If the operator is overloaded, what does it resolve to?
   - - Are the arguments to a function call "type compatible" with the declaration of the function?

3. Perform extra grammatical syntactic checks:
   - Is the number of arguments in a function call same as that in the declaration?
   - Does the LHS of a assignment have a l-value?

4. Intermediate code generation: While not an analysis, the techniques used in static analysis can also be used for intermediate code generation.

# Static Semantic Analysis - An Example

```
#include <stdlib.h>

int x = 2;

int * f (int* p, float r)
{
  int two = x;
  float pi = 3.1415;
  p = (int*) malloc(sizeof(int));
  *p = two * 3.1415 * r;
  return p;
}
```

What is the type of x?

Is this assignment type correct?

Is this a valid operation

Global Symbol table

| Symbol Name | var/fun/type | type/return type | size | symtab |
|---|---|---|---|---|
| malloc | fun | VOID* | | |
| x | var | INT | 4 | |
| f | fun | INT* | | |

Symbol table for f

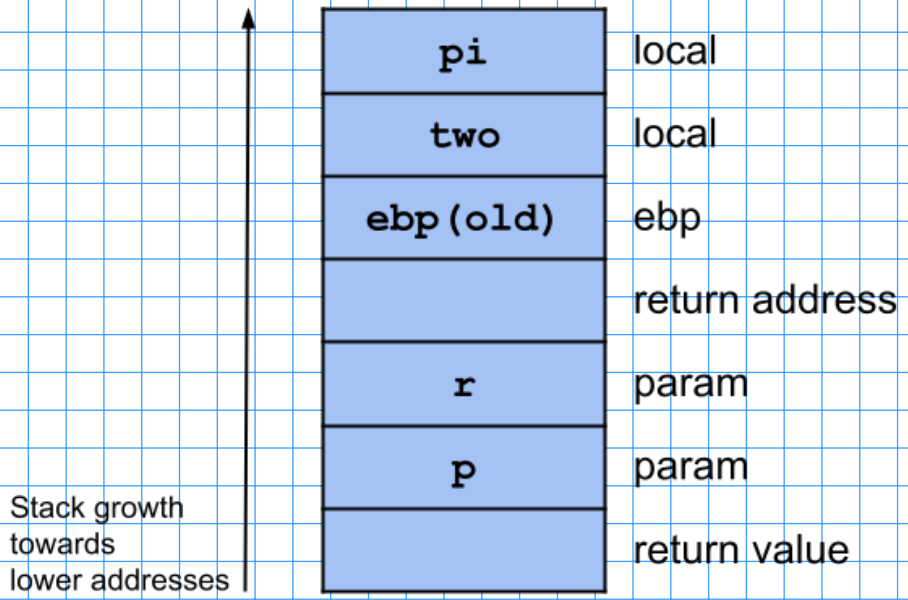
| Symbol Name | var/fun | param/local | type/return type | size | offset | symtab |
|---|---|---|---|---|---|---|
| p | var | param | INT * | 8 | 12 | |
| r | var | param | FLOAT | 4 | 8 | |
| two | var | local | INT | 4 | -4 | |
| pi | var | local | FLOAT | 4 | -8 | |

Symbol table for malloc

| Symbol Name | var/fun | param/local | type/return type | size | offset | symtab |
|---|---|---|---|---|---|---|
| size | var | param | size_t | 4 | 12 | |

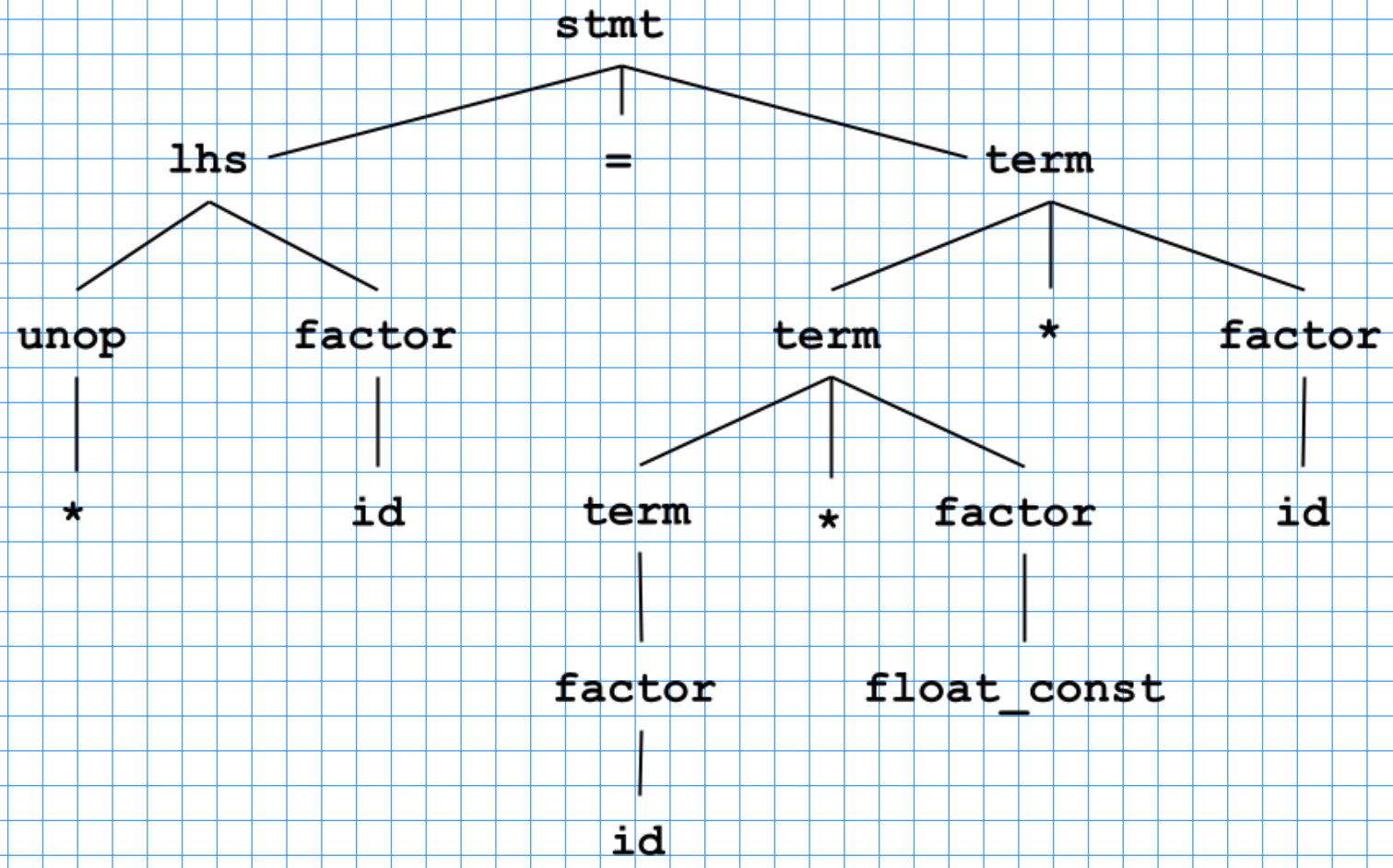
# Static Semantic Analysis - An Example

The offsets assume an activation record layout that looks like this:



# Static Semantic Analysis

```
*p = two * 3.1415 * r
```



# Static Semantic Analysis

1. We associate attributes with non-terminals

```
factor --  type
term   --  type
id     --  type
lhs    -- type
unop   -- op
```

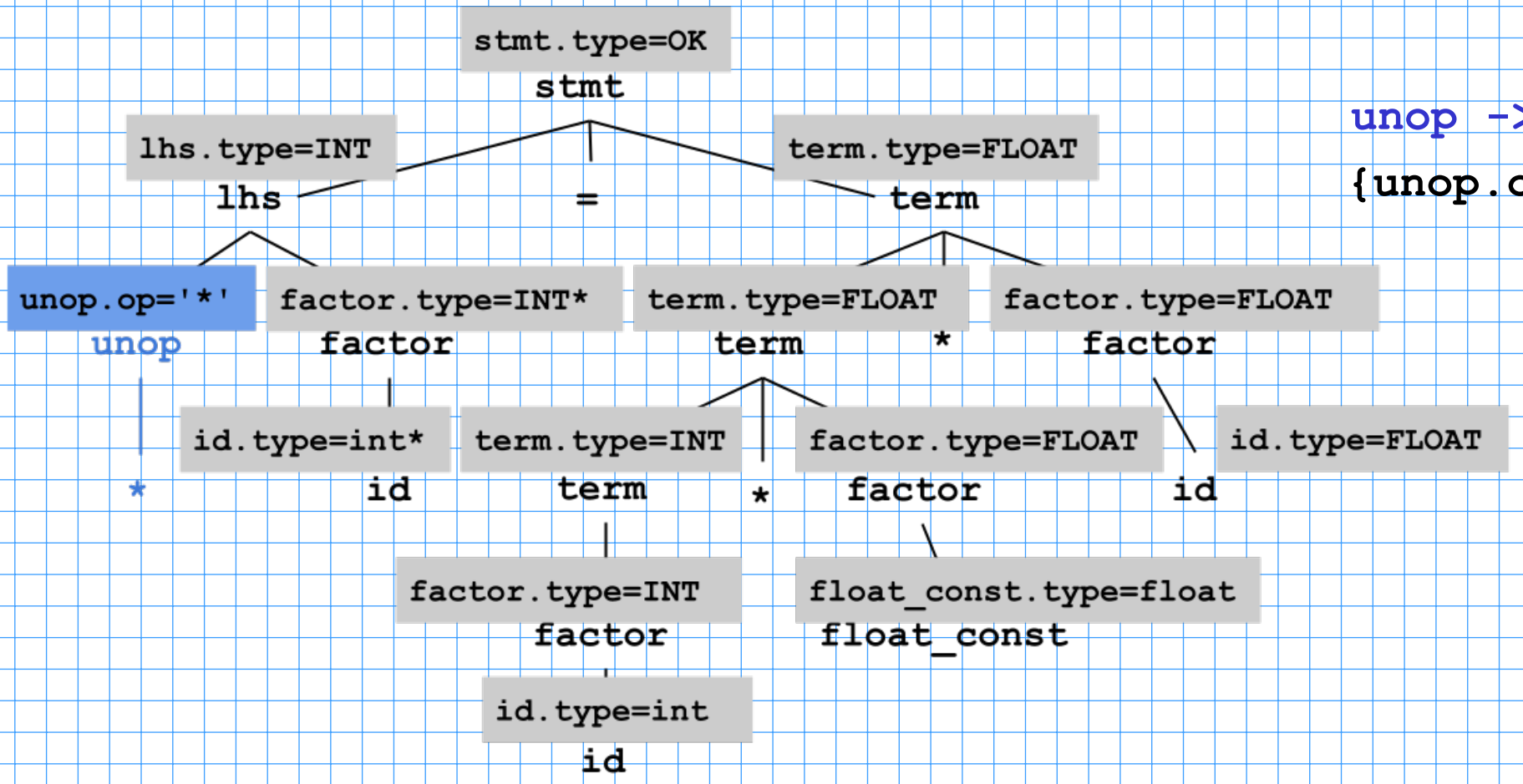
One can think of attributes as properties of non-terminals.

2. Attributes have values.

Some of the possible value of type are:
{INT, FLOAT, INT*, ERROR, OK}
Possible values of op are {'*', '-'}

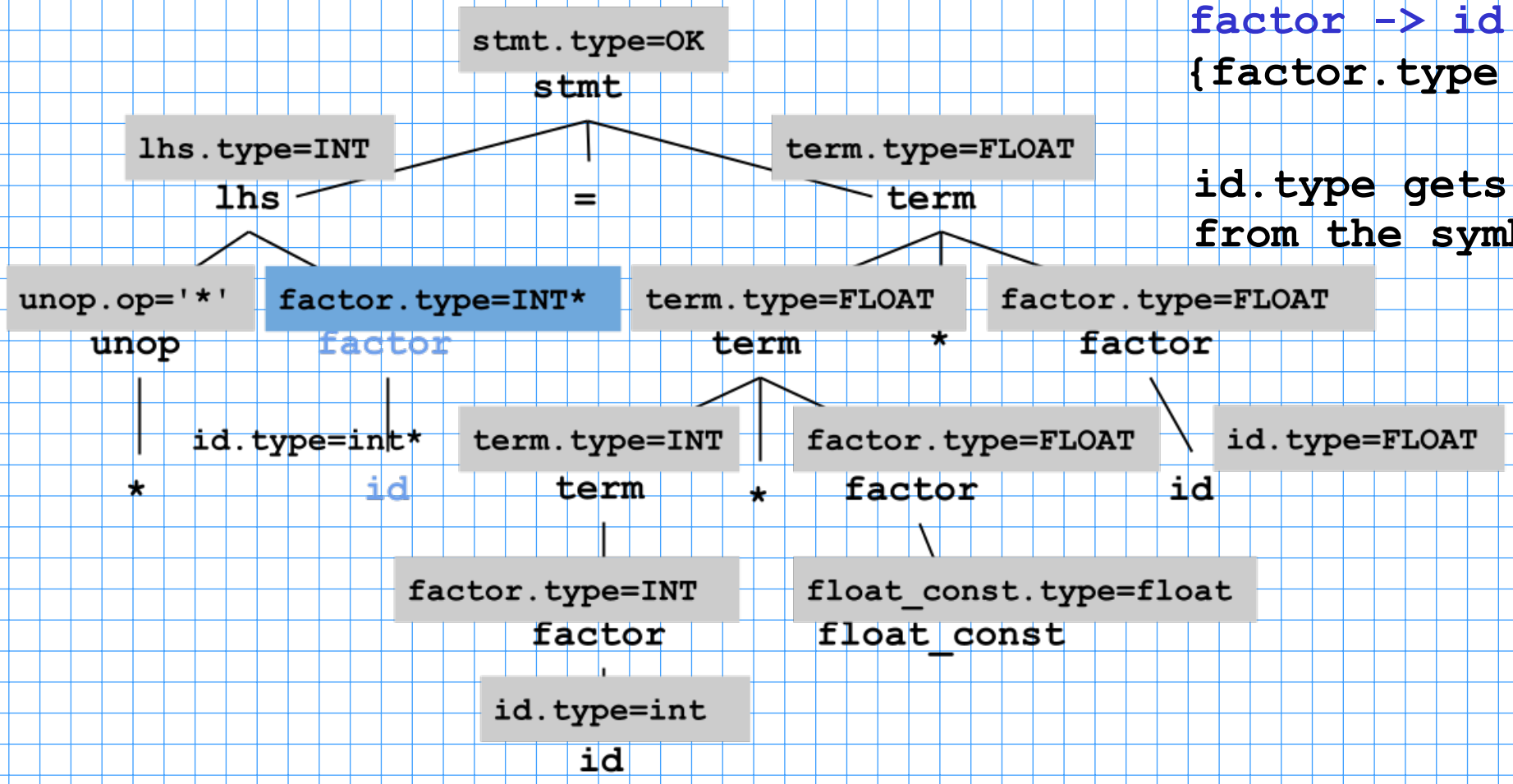
3. To compute the attribute of a non-terminal, we associate rules with productions. Attributes of a terminal are supplied by the lexer, or the symboltable.

# Rules for Attribute Evaluation



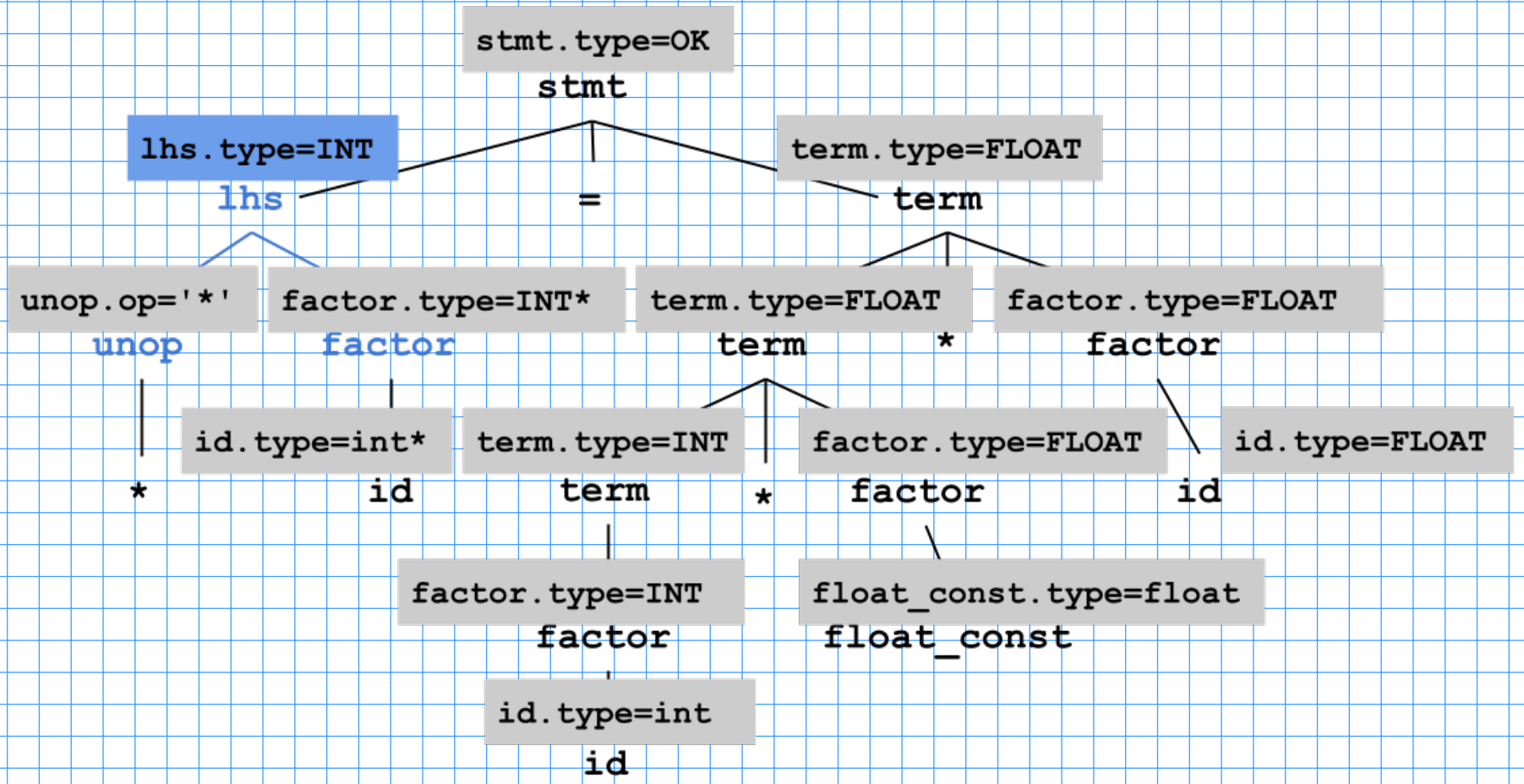

```
unop -> '*'
{unop.op = '*'}
```

# Rules for Attribute Evaluation



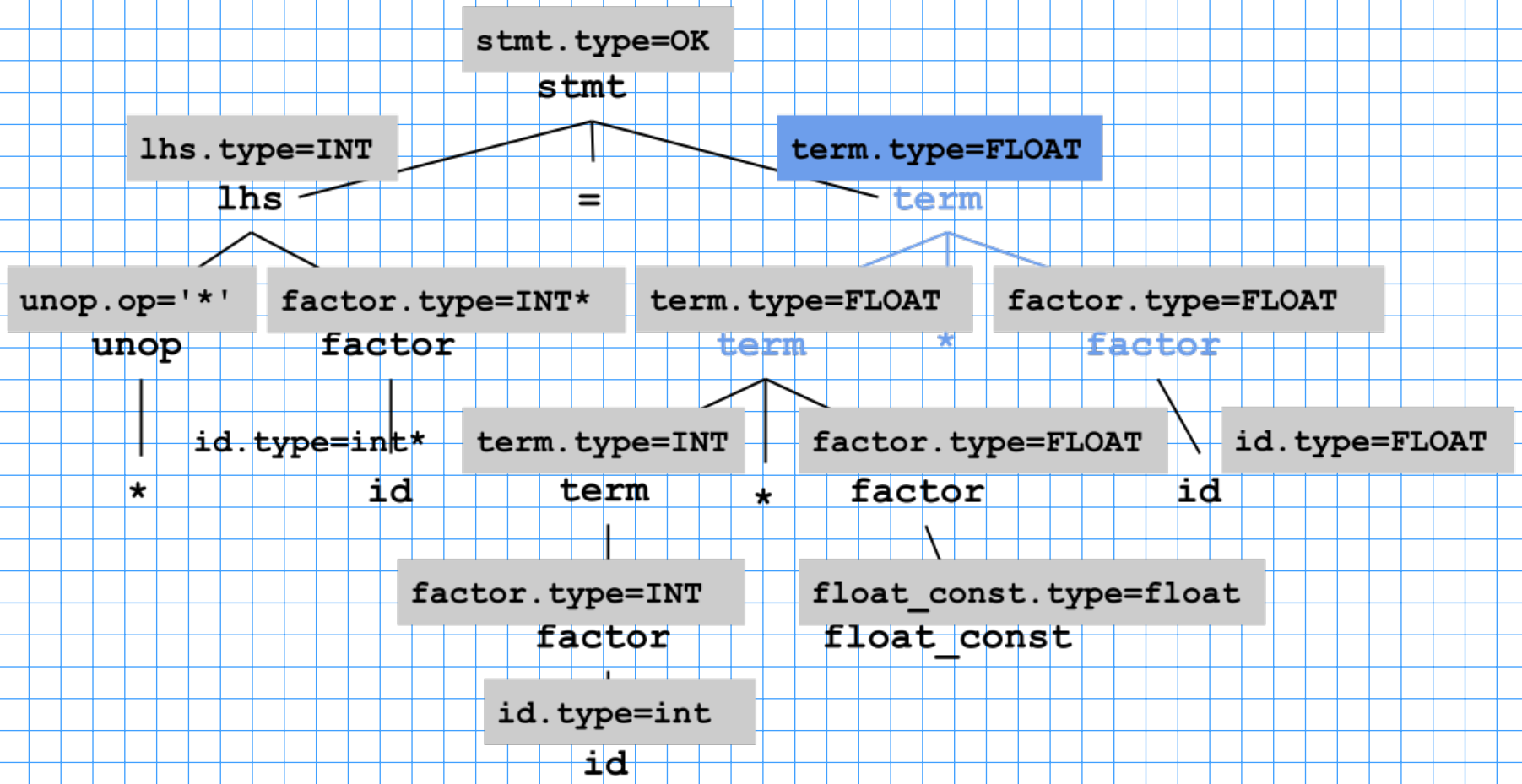

```
factor -> id
{factor.type = id.type}

id.type gets its value
from the symboltable
```

# Rules for Attribute Evaluation




```
lhs -> unop factor
{if (unop.op = '*')
   if (factor.type==INT*) lhs.type = INT;
   elseif (factor.type==FLOAT*) lhs.type = FLOAT;
   else error();
 if (unop.op = '-') lhs.type = factor.type;}
```

# Rules for Attribute Evaluation




```
term1-> term2 * factor

{case (term2.type,factor.type) of
    (INT,INT)-> term1.type = INT
    (FLOAT,INT)-> term1.type = FLOAT
    (INT,FLOAT)-> term1.type=FLOAT
    (FLOAT,FLOAT)->term1.type=FLOAT
    (_,_)-> error()}
```

# Syntax Directed Translation

## Tying everything together:

```
factor -> id                          {factor.type = id.type}

factor -> float_const                 {factor.type = FLOAT}

term -> factor                        {term.type = factor.type}

term1-> term2 * factor                {case (term2.type,factor.type) of
                                          (INT,INT)-> term1.type = INT
                                          (FLOAT,INT)-> term1.type = FLOAT
                                          (INT,FLOAT)-> term1.type=FLOAT
                                          (FLOAT,FLOAT)->term1.type=FLOAT
                                          (_,_)-> error()}

unop -> *                           {unop.op = '*'}

lhs -> unop factor                    {if (unop.op = '*')
                                        if (factor.type==INT*) lhs.type = INT;
                                        elseif (factor.type==FLOAT*) lhs.type = FLOAT;
                                        else eror();
                                      if (unop.op = '-') lhs.type = factor.type;}

stmt -> lhs = term                    {if (lhs.type == term.type) or
                                          (both lhs.type and term.type are
                                           non-pointer types) stmt.type = OK
                                           else error();
```

# Syntax directed definition (SDD)

Section 5.1 -Aho-Sethi-Ullman -Lam (ASUL)

- Is a grammar along with semantic rules.
- Associates a set of attributes with each grammar symbol.

A synthesized attribute for a grammar symbol A at a parse-tree node N is defined by a semantic rule associated with the production at N. Note that the production must have A as its head. A synthesized attribute at node N is defined only in terms of attribute values of the children of N, and the attribute values of N itself.

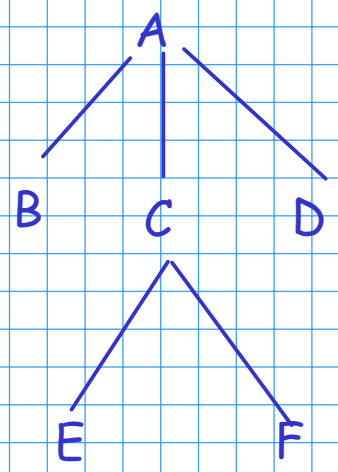
An inherited attribute for a grammar symbol at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N. Note that the production must have B as a symbol in its body. An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself, and N's siblings.

Note that terminals also have synthesized attributes that are derived from the lexical analyser or the symboltable.

# Syntax directed definition (SDD)

A -> B C D

C -> E F




Each grammar symbol has, in general
- one or more inherited attributes
- one or more synthesized attributes

semantic rules

$C.s = f(E.i, E.s, F.i, F.s, C.i, C.s')$

$C.i = g(A.i, A.s, B.i, B.s, C.i', C.s, D.i, D.s)$

For attributes to be computable, there should not be any circular dependences.

Important: While doing attribute evaluation, it is assumed that the parse tree already exists.

# Syntax directed translation scheme (SDTS)

Section 5.2.2 -ASUL

To evaluate the attributes along with parsing, we impose two restrictions:

-S-attributed definitions: Only synthesized attributes are used. Easy implementation, restrictive.

-L-attributed definitions: In this, each attribute must be either

1. Synthesized, or
2. Inherited, but with the rules limited as follows. Suppose that there is a production A -> X1X2 · . . Xn , and that there is an inherited attribute Xi.a computed by a rule associated with this production. Then the rule may use only:
   - ( a ) Inherited attributes associated with the head A.
   - ( b ) Either inherited or synthesized attributes associated with the occurrences of symbols Xl , X2, ..., X{i-l} located to the left of Xi .
   - (c) (Other) inherited attributes of Xi itself.

When actions are inserted within the productions to indicate the place where they are to be triggered, we get a Syntax Directed Translation Scheme (SDTS).

# Syntax directed translation scheme (SDTS)

A -> B C D
C -> E F

## S-attributed definitions:

There are no inherited attributes

C.s = f(E.s, F.s, C.s')
A.s = g(B.s, C.s, D.s, A.s')

## L-attributed definitions:

C.s = f(E.i, E.s, F.i, F.s, C.i, C.s')
C.i = g(A.i, B.i, B.s, C.i')

An inherited attribute of a grammar symbol depends on (i) the inherited attributes of its parent, (ii) the inherited and synthesized attributes of its left siblings and (iii) the inherited attribute of the grammar symbol itself

Embed actions before a grammar symbol in the RHS of a production to compute its inherited attributes.
Embed actions at the end of a production to compute the synthesized attributes of the non-terminal on the LHS of the production.

A -> B {C.i = g(A.i, B.i, B.s, C.i)} C D

C -> E F {C.s = f(E.i, E.s, F.i, F.s, C.i, C.s)}

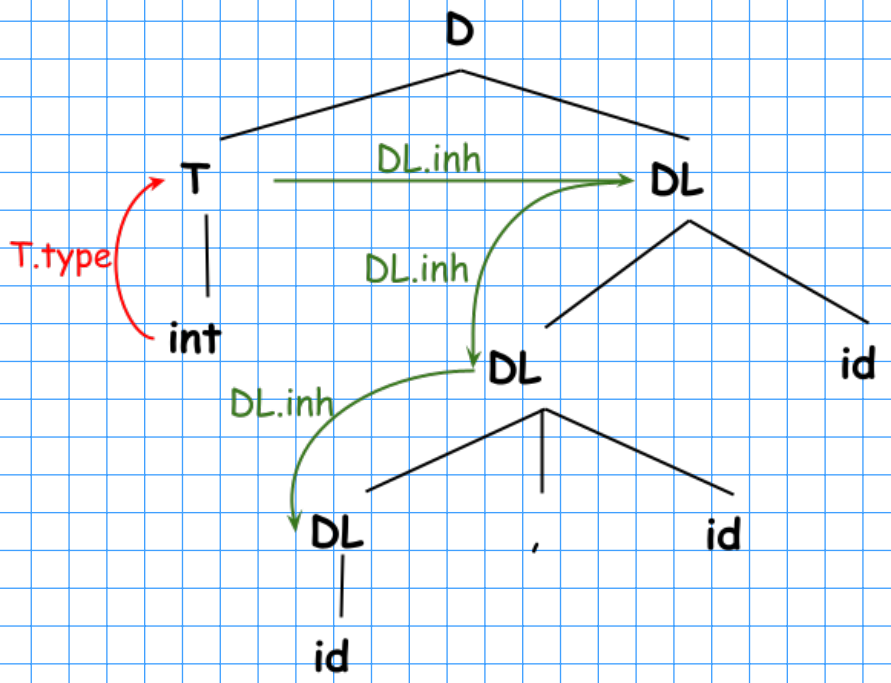
# Why Inherited Attributes?

Example from ASUL:

D -> T DL
T -> int
T -> float
DL -> DL1, id
DL -> id

We want to add the type of each id to the symboltable:

D -> T {DL.inh = T.type} DL
T -> int {T.type = INT;}
T -> float {T.type = FLOAT;}
DL ->{DL1.inh = D.inh} DL1, id{addtoST(id, DL.inh)}
DL -> id {addtoST(id, DL.inh)}



# Relation between attributes and $ symbols in bison

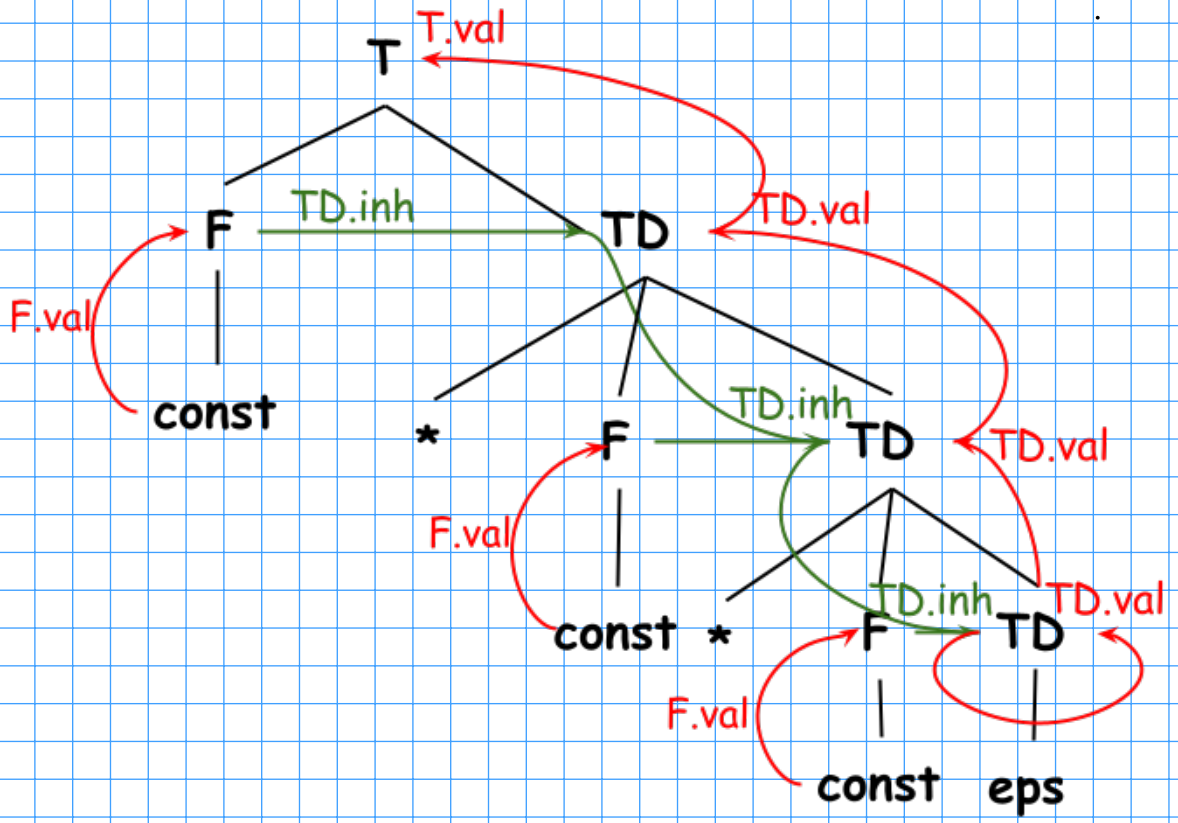
T -> F TD

TD -> * F TD

TD -> eps

F -> const

Compute a attribute T.val that gives the value of the expression T.



# Relation between attributes and $ symbols in bison

```
T -> F  TD
TD -> * F  TD
TD -> eps
F -> const
```

```
T ->  F  {TD.inh = F.val;}  TD   {T.val = TD.val;}

TD1 -> eps {TD1.val = TD1.inh;}

      |  '*' F  {TD2.inh = F.val * TD1.inh;} TD2    {TD1.val = TD2.val;}

F -> CONST  {F.val = CONST.val;}
```

# Relation between attributes and $ symbols in bison

Positional conventions:

A -> {action1} X {action2} Y {action3} Z {action4}

1 2 3 4 5 6

| Symbol | In position | Condition | Signifies |
|---|---|---|---|
| $0 | any | | Inh attr of LHS |
| $$ | rightmost | | Syn. attr of LHS |
| $i | any | position i has grammar symbol X | Syn attr of X |
| $i | any | position i has an action. | Inh attr of grammar symbol at i+1 |
| $$ | in position i*, not rightmost | | Inh attr of grammar symbol at i+1 |

* - provided, it is allowed by the rules of L-attributed definition.

# Relation between attributes and $ symbols in bisonc

T -> F T'

T' -> * F T'

T' -> eps

$0 - Inh attr of LHS
$$ (in rightmost position) - Syn. attr of LHS
$i (1 is a grammar symbol X) - Syn attr of X
$i (i is the position of a grammar symbol X)
- Syn attr of X
$i (i is an action) - inh attribute of grammar symbol position i+1
$$ (in position i, not rightmost) - inh attribute of grammar symbol at position i+1

```
                {TD.inh = F.val;}      {T.val = TD.val;}
T :  F   {$$ = $1;}   TD{  $$ = $3;};
```

```
                {TD1.val = TD1.inh;}          {TD2.inh = F.val * TD1.inh;}       {TD1.val = TD2.val;}
TD1 : { $$ = $0; }  |    '*'   F {$$ = $2 * $0;}    TD2     {$$ = $4;};
```

```
                                {F.val = CONST.val;}
F :     CONST     { $$ = stoi($1);};
```

# Conversion of attributes into positions in the value stack

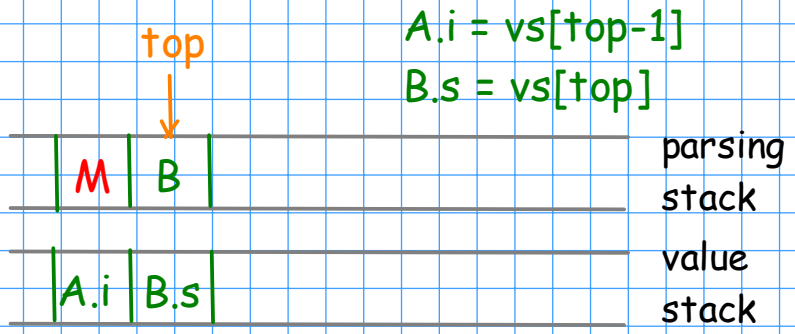
Section 5.5.4, ASUL

```
A -> B    {C.inh = A.inh + B.syn}
     C    {D.inh = C.inh * A.inh}
     D    {A.syn = D.syn}
```

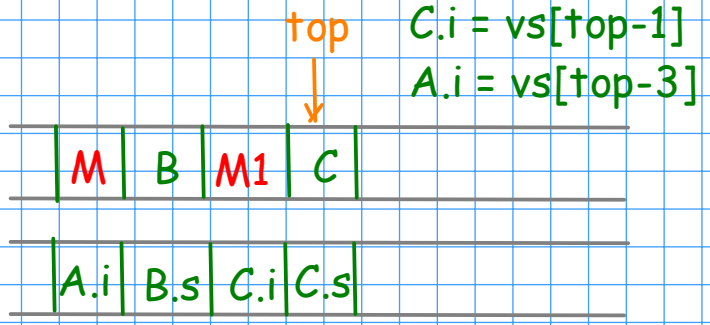
Converted into:

```
A -> B  M1
     C  M2
     D {A.syn = D.syn}

M1 -> eps  {C.inh = A.inh + B.syn}
M2 -> eps  {D.inh = C.inh * A.inh}
```

stack configurations just before reducing epsilon to M1.



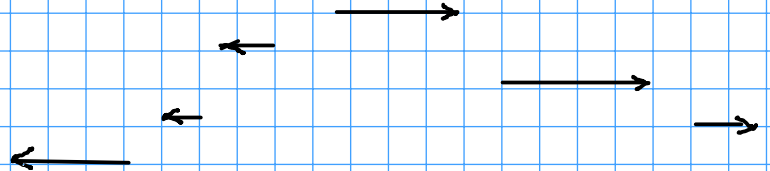

stack configurations just before reducing epsilon to M2.



# Type Analysis

## How do we read C type declarations?

Example: char * (* (* * x [2] [2]) (int, int) ) [3]



Starting fom the variable, alternately move
- first to the right until a right parenthesis or end of the declaration is encountered
- then to the left until a left parenthesis or end of the beginning of the declaration is encountered

x is a array [2] [2] of
pointers to pointers to
a function that takes two ints as arguments
and returns a pointer to
an array [3] of
pointers to char

## Representing C types as trees

The type of x can be represented as

Reference: http://www.unixwiz.net/techtips/reading-cdecl.html

# Basic facts about arrays:

1. An array a declared as int [i][j], is also of the type int (* const . ) [j].
It cannot be assigned to because of the const-ness of the pointer.

a = { {1,2,3}, {4,5,6}, {7,8,9}}  a has the type int [3][3]

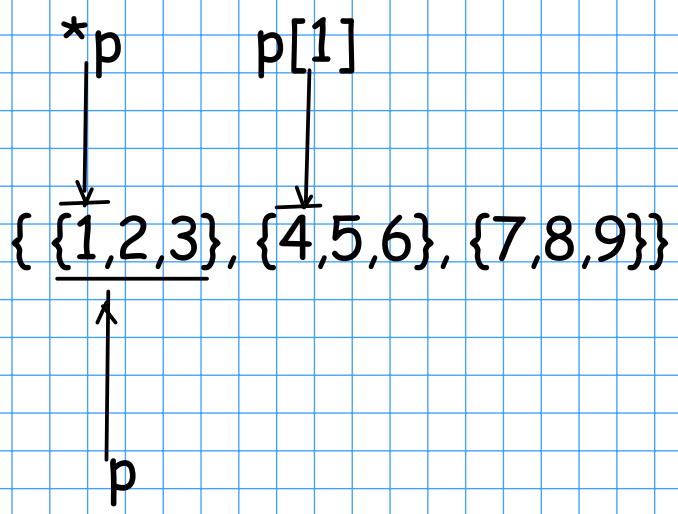
{ {1,2,3}, {4,5,6}, {7,8,9}}  a also has the type int (* .) [3]


2. If p is of the type int (* .) [j], one can assign to p. For example, p = a.
One can also index p like a normal array.

i = p[2][1]

# Basic facts about arrays

3. If p is of the type int (*) [j], then p[i] is *(p + i), where adding i advances the pointer p by i* sizeof(int) *j bytes. Thus p[0] = *p.




*p is of type int[3]
*p is of type int*
*p has the value {1,2,3}

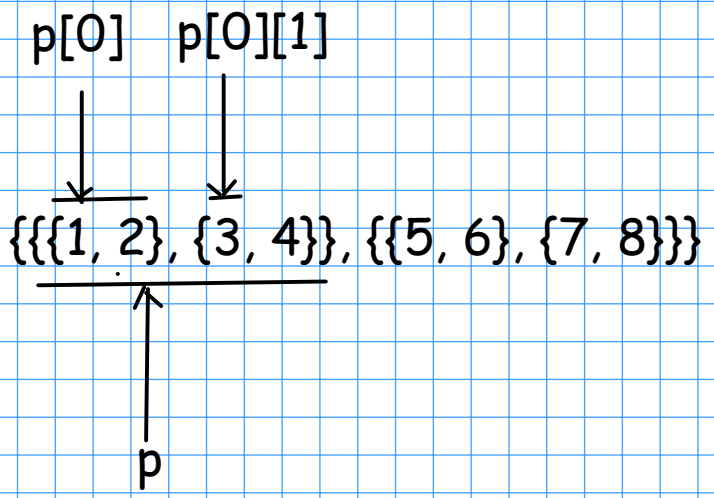
p[1] = *(p+1)

p[1] is of type int[3]
p[1] is of type int*

p[1][1] = *(p[1] +1) = 5

# Arrays: An Example

```
void fun(int p[2][2][2])
{
   int e[2][2][2] = {{{1, 2}, {3, 4}}, {{5, 6}, {7, 8}}};
   p = e;
   printf("%d\n", p[0][0][1]);
}
int main()
{
   int e[2][2][2] = {{{1, 2}, {3, 4}}, {{5, 6}, {7, 8}}};
   int(*p)[2][2] = e;
   *p[0][1] = 10; //which part of e is changed?
  (*p[0])[1] = 20; // which part of e is changed?
  fun(e);
}
```




Note: The array e int p[2][2][2] as a local varianle of main cannot be assigned to

```
e = p; // wrong
```

However int p[2][2][2] declared as a parameter of fun can be assigned to

```
p = e; // ok
```

# Type checking

Check whether the types of an expression matches its context.

```
main ()
{
   p = q;
}
```

```
main ()
{
   f (b)
}
```

```
void f (int * p)
{
   ...
}
```

The = context, and the parameter passing context demand that the types are "compatible".

Compatible does not mean the same.

# Type checking

1. Any numeric type or an enumerated type is compatible with any other numeric type or enumerated type. When a type is not the same as a its context, it is converted through an implicit cast.

```
main ()
{
  int i; float j;
  i = j;
}
```

```
main ()
{
  char i; float j;
  i = j;
}
```

```
main ()
{
  float i; char j;
  i = j;
}
```

```
typedef enum {sat, sun} weekend;
typedef enum {true, false} boolean;

main ()
{
  float i; weekend j;
  i = j;
}
```

```
typedef enum {sat, sun} weekend;
typedef enum {true, false} boolean;

main ()
{
  weekend i; boolean j;
  i = j;
}
```

All the above programs typecheck

# Type checking

2. None of the numeric type are compatible with the void type. Variables cannot have the void type. void* is compatible with any pointer type.

```
int main ()
{
  void* i;
  int* j;
  float* k;
  i =j; // ok
  k = i;// ok
}
```

# Type checking

In the case of pointers to numeric types, two types are compatible, if they point to the same types. Thus numeric types are freely assignable to each other, pointer to numeric types are not. We discussed in the class why.

```
typedef int *t;
typedef int *s;
typedef float *r;

int main()
{
   t i;   s j;   r k;   int *l;
   l = i; //ok
   j = i; //ok
   j = k; // not ok. gives warning
   k = j; // not ok. gives warning
}
```

# Type checking

struct types are compatible if they are declared in the same declaration

```
typedef struct {int i;} s, t;
typedef struct {int i;} r;

int main()
{
   t i;   s j;   r k;
   struct {int i;} l;
   j = i;  //ok, s, t are compatible types.
   j = k; //not ok, r and s are not compatible types.
   l = k;  //not ok.
}
```

# Type checking

Pointers to non-numeric types s and t are compatible, if s and t are by themselves compatible.

```
typedef struct {int i;} s, t;
typedef struct {int i;} r;

int main ()
{
  t*  i;  s*  j;  r*  k;
  struct {int i;} *l;
  j = i;   //ok, s, t are compatible types, see above.
  j = k;   //not ok, r and s are not compatible types.
  l = k;   // not ok.
}
```

# Type checking

```
typedef int arr[7][8];

void f(int (*i)[5])
{
}
int main ()
{
  arr a;      // essentially int (*.)[8]
  f(a);       // therefore ok.
  int ** p;
  int  (*i)[5], *k[5];
  i = a;                  // not ok: int (*)[5] with  int (*)[5]
  i = k;                  // not ok: int (*)[5] incompatible with int **
  p = i;                  // not ok: int **  incompatible with int (*) [5]
  int l = p[1][2];        // ok: An expression with type int** p can be converted into int p [][]
  l = **a; //ok:
}
```

int (*)[5] may not be compatible with int**. But an expression of the type int (*)[5] can be converted to an expression of the type int** and vice-versa

# Type checking

Type checking structs: Two struct values are compatible if they have the same name

```
struct a{int x;};
struct b{int x;};

int main ()
{
  struct a i;
  struct a j;
  struct b k;
  i = j;
  i = k;    // incompatible types when assigning to type
            // {struct a} from type {struct b}
}
```

# Type checking

There are two names for the first struct below. "struct a" and "cell". Similarly there are two names for the second struct "struct b" and "cell1".

```
typedef struct a
{
  int val;
  struct a* next;
} cell;

typedef struct b
{
  int val;
  struct a* next;
} cell1;
```

```
main ()
{
  cell p; cell1 q;
  struct a r;
  p = q;      // Not ok
  p = r;      // ok
}
```

# Type checking

## Type Equivalence:

Two types t1 and t2 are structurally equivalent if, any one of the following conditions hold:

1. t1 and t2 are eqivalent basic types or the same type name.
2. t1 and t2 are constructed by applying the same constructor on structurally equivalent types.
3. One of t1 or t2 is a name and the other is structurally equivalent to the type denoted by this name.

Two types t1 and t2 are name equivalent if, any one of the following conditions hold:

1. t1 and t2 are equivalent basic types or the same type name.
2. t1 and t2 are constructed by applying the same constructor on name equivalent types.

# Type checking

```
typedef int cell array[1][2];
typedef int cell1 array[1][2];
```

```
struct
{
   int val;
   cell *next;
} x;

struct
{
   int val;
   cell1 *next;
} y;
```

x and y are type equivalent under structural equivalence but not under name equivalence.