ARITHMETIC EXPRESSIONS

- **The “workhorse” of procedural computing**
- **Includes assignment (:=) as an operator and functions as multi-operand prefix operators**
- **Functions can also be treated as unary operators with an operand of a simple list**
- **Argument lists (and parameter lists) treat separators (comma, space) as “stacking” or “append”operators**
- **Can treat keyword language statements as functions in which the remainder of the statement is the operand**

REPRESENTATION OF EXPRESSIONS - I

- **Parenthesized infix (the most common notation)**

  → **relies on parentheses and a precedence order of operators to determine meaning or value**

  → **Ex: 5 + 6 * (-3) / 22 ^ 2**

- **Polish (Lukasiewicz) Notation**

  → **parenthesis free - operators bind with their nearest neighbours to form evaluative groups and order is determined by availability of evaluated operand**

  → **Operators can be either prefix (to the left of the operands) or postfix (to the right of the operands, sometimes called suffix)**

  → **operands can be in their normal L-R order or reversed (there are some advantages of reversing the order, such as the order in which they may be accessed)**

  → **Ex: + 5 * 6 / - 3 ^ 22 2 is prefix, forward while**
  + * / ^ 2 22 - 3 6 5 is prefix reverse

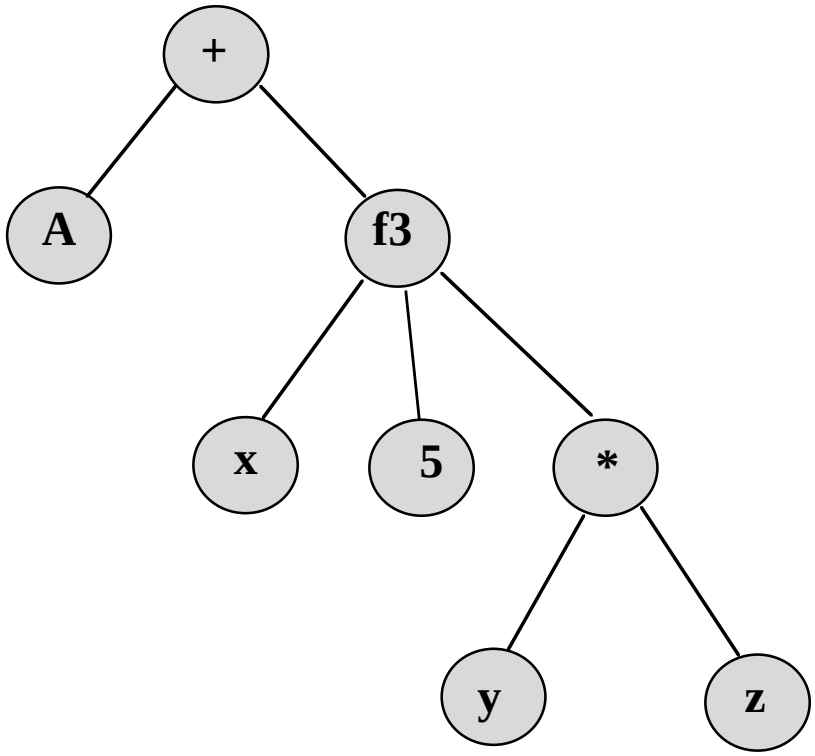
REPRESENTATION OF EXPRESSIONS - II

- **Operator Tree representation**

  → **A tree in which each subtree represents a subexpression with the operator in the root and the operands as leaves:**

Represents:

A + f3 (x , 5, y * z)



Ideally, we want syntactic trees for expressions to reflect the same evaluation order as operator trees -- i.e. bottom to top

REPRESENTATION OF EXPRESSIONS - III

- **Abstract Syntax**

  → **gives semantic components, independent of spatial placement of operands**

  → **common notation for formal descriptors that does not specify syntactic format**

  → **Ex: (opr opd_1 opd_2)**

Operator Evaluation Order - I

- **Precedence**

  → **Operator precedence rules define the order in which operators of different precedence levels are evaluated:**

Parenthetical groups (...)	
Exponeniation	**
Mult & Div	* , /
Add & Sub	+ , -
Assignment	:=

  A * B + C ** D / E - F = ((A * B) + ((C ** D) / E) - F)

  QUESTIONS:

  (1) Where do functions come in the hierarchy?

  (2) Where do unary operators lie?

  (3) Develop a complete listing of hierarchy including logical and boolean operators

Operator Evaluation Order - II

- **Associativity**

  → **Operator associativity rules define the order in which adjacent occurrences of operators with the same precedence levels are evaluated:**

  Left associative * , / , + , -

  Right associative **

  B ** C ** D - E + F * G / H =

  (((B ** (C ** D)) - E) + ((F * G) / H))

  EFFECTIVELY

  Most programming languages evaluate expressions from left to right

  LISP uses parentheses to enforce evaluation order

  APL is strictly RIGHT to LEFT, taking note only of parenthetical groups

  SmallTalk is LEFT to RIGHT but with no reference to precedence except for parenthetical groups

ANALYSIS OF EXPRESSIONS

- **Fully parenthesize and then rely on depth of parentheses to determine order of evaluation**
- **Evaluate subexpressions in order of precedence of operators:**
  - → **(...)**
  - → *** /**
  - → **+ -**
  - → **:=**
  - → **and thereafter evaluate left to right**

    (May require several passes over the expression)
- **Samelson/Bauer (stack/cellar method)**
  - → **Uses a Y-shaped railroad track to evaluate expression**
- **Tree evaluation (operator trees)**
- **Or the programming language may decide to completely ignore precedence and hierarchy**

THINGS TO KNOW ABOUT OPERATORS

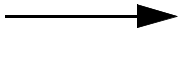
- **Their relative hierarchy or precedence**
- **Associativity**
- **Number of required operands**
- **Placement of operands**
- **Potential distinct meanings (depending on context)**

Side Effects and Order of Operand Evaluation

- **A functional side effect occurs when a fucntion either changes one of its parameters or a global variable**

Consider:

```
Procedure sub 1 (...);
    var a : integer;
        function fun (x : integer) : integer;
            x :=  a + 2;
            return (x)
        end;
    a := 7;
    b := a + fun ( a );
    print ( a , b );
end;
```



w/o SE:	7 , 16	
w/ SE:	9 , 16	a , fun(a)
w/ SE:	9 , 18	fun(a) , a

Overloading

An operator or function is overloaded if its meaning depends on the number or types of its arguments

- **+ (real, integer)**
- **- (unary, binary)**

Also called ad-hoc polymorphism

Some languages permit overloading of user-defined functions.

Good idea??

Three kinds of overloading:

- literal
- operator
- subprogram

(These are covered in three different sections in Sebesta)

Coercion (aka “Casting”)

- **Implicit, or automatic, type coercion of operators.**

```
var x: integer;
    y, z: real;
     ...
y := x + z;
```

→ **"mixed-mode" expression**

- **Coercion is different from explicit type conversion:**

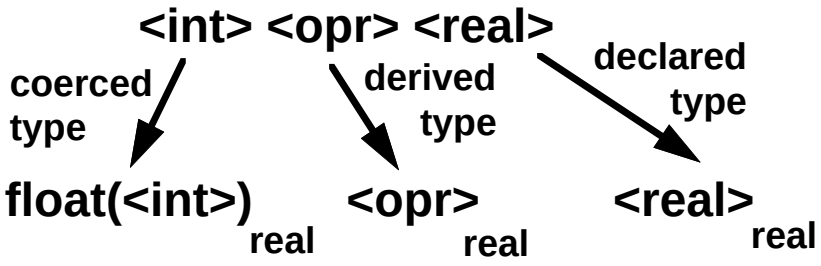
```
int x,y
...
(double) x / (double) y
```

However if we treat (double) as a function reference for explicit type conversion then there still remains the derivation of the type of the / operator to double.

COERCION (continued)

- **Provides type compatability across program elements**

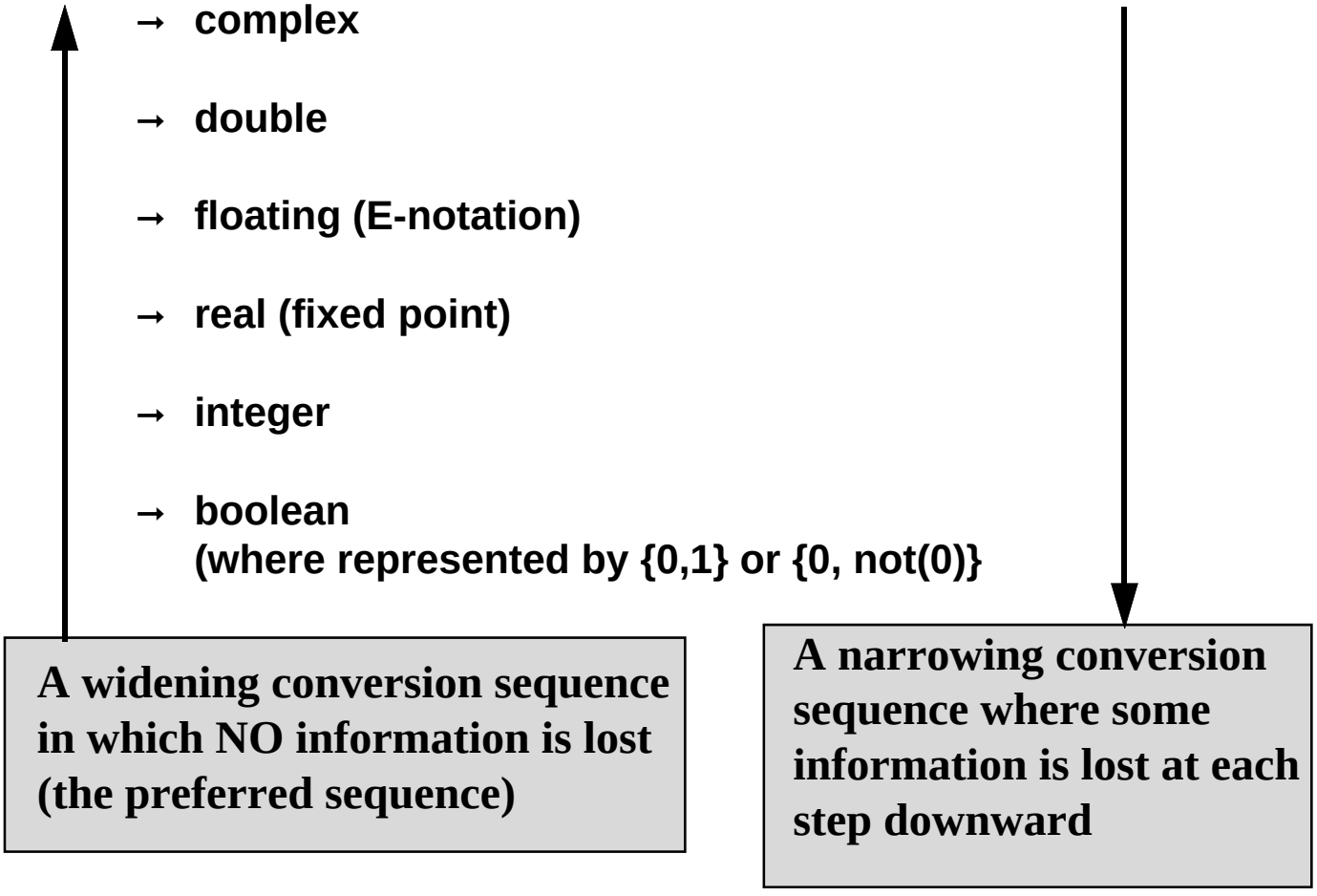
  mixed mode:



Coercion provides the “solution” to overloading

HIERARCHY OF NUMERIC TYPES

- **There is an implied ordering of numeric types which provides the step by step conversion, for example:**



Question: Should narrowing involve rounding or truncation?

CHOICE OF CONVERSION DIRECTION

- **Given a choice it is better to widen if possible.**

Example:

int: x, y
real: z

then what would be the most appropriate conversion sequence for the expression

x := y * z ?

(a) the order of evaluation is:
(y * z) then (x := result)

(b) So widen (y * z) to ((real y) * z) so as to coerce the * operation to real, and then

(c) Narrow the assignment to (x := (int result))

Example:

y := 3
z := 5.9

then x = 15 if narrowed first, and 17 if widened first

Coercion Extremes (Algol 68)

```
int    i;
real   r;
[1:10] int row i;
ref int ref i;
union (int,real)  ir;
proc int p;
   ...
r := i/r;              -- widening (of i)
ir := i;               -- uniting
i := refi;             -- dereferencing
i := p;                -- deproceduring
row i := 5             -- rowing
```

Relational and Boolean Expressions

- **Relational operators compare two operands and return a boolean**

  = < > <= >= <>

- **Lower precedence than arithmetic operators**

  a + b < c + d ≡ (a + b) < (c + d)

- **Boolean values: true, false**

- **Boolean operators: and, or, xor, not, =**

- **True boolean values are helpful.**

  → **In C, use integers:**

  0 = false

  other = true

  → **a > b > c is legal!**

Short-Circuit Evaluation

- **Get a result without evaluating entire expression.**
  - → **x and y ≡ if x then y else false**
  - → **x or y ≡ if x then true else y**
  - → **(x and y are arbitrary boolean expressions)**
- **Same as regular evaluation in the absence of side-effects.**
- **May have a choice, e.g. Ada:**
  - → **x or else y**
  - → **x and then y**
  - → **Otherwise, need to know if it's used.**

EXPRESSION SEMANTICS

- **In terms of ATTRIBUTE GRAMMARS**
- **On the basis of BNF grammar, add ‘attributes’to each element (Non-terminals and Terminals)**
- **Represented as a subscript list:**

  → $E_{r,s}$

- **Where attributes can be associated values, types, or other information to be carried through the syntactic domain**
- **Syntactic rules are then augmented with semantic rules that describe the properties of the attributes**

  → $E_{r,s} \rightarrow E_{p,q} + T_{m,n}$

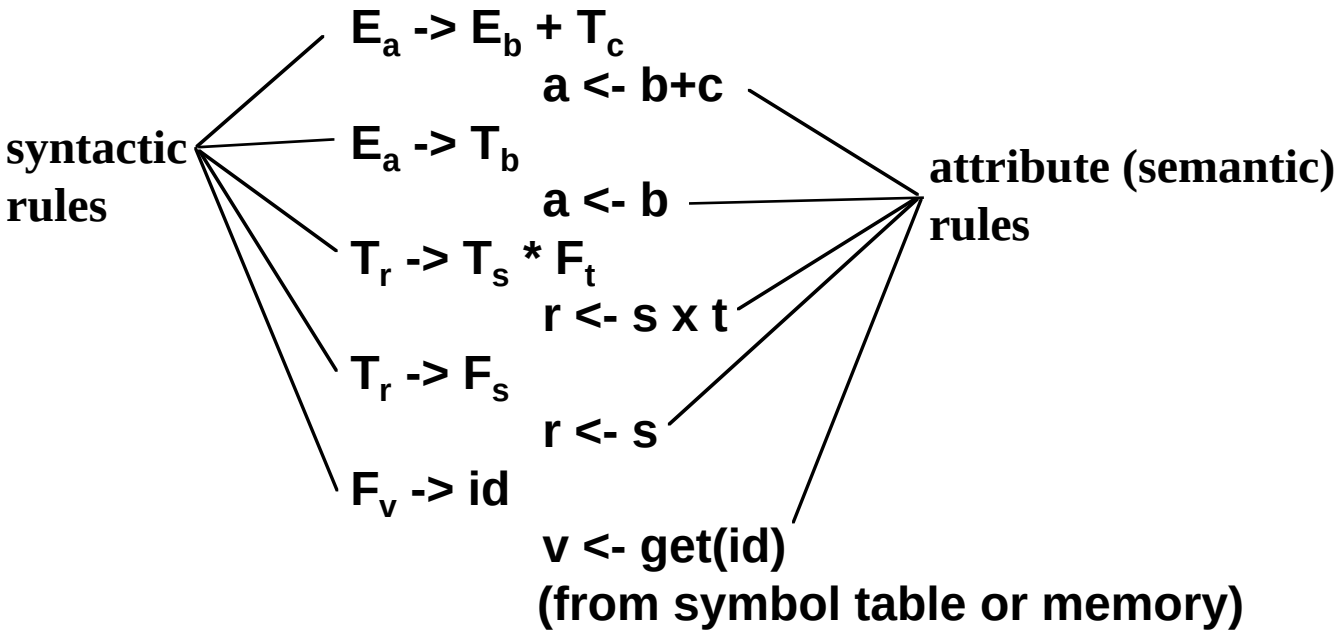
  $s = \max(q,n)$

  $r = p +_r m$

Where s, q, n are types and r, p, m are associated “values”

EXAMPLE

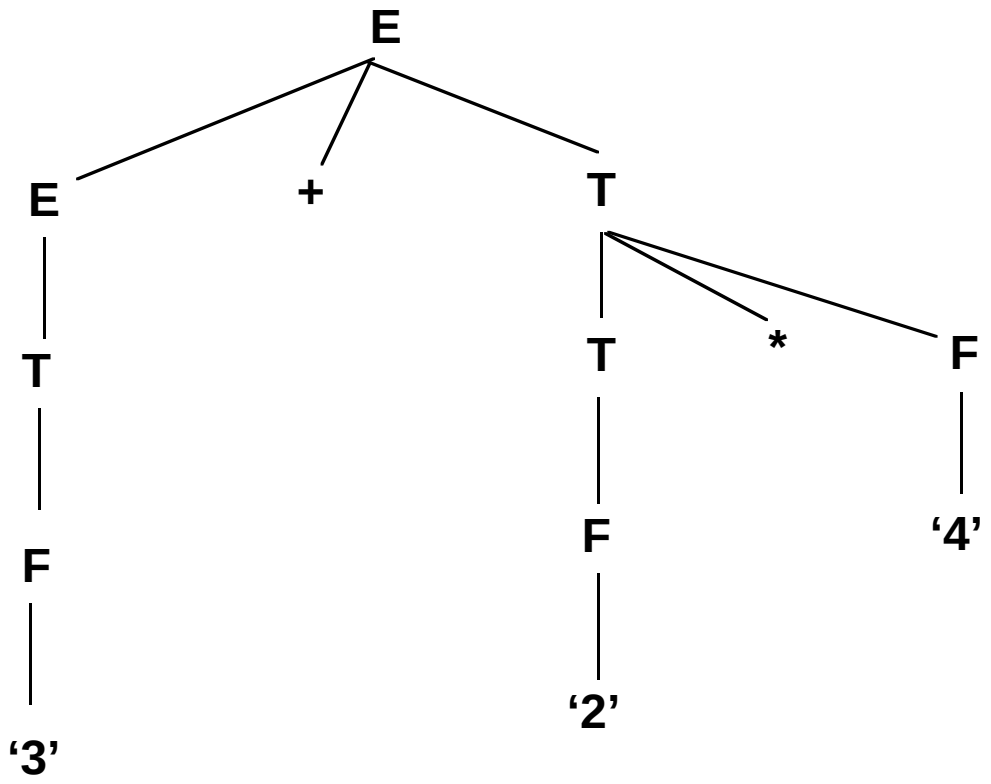
Attribute Grammar:



Note: The scope of an attribute is the rule in which it is attached to a language element

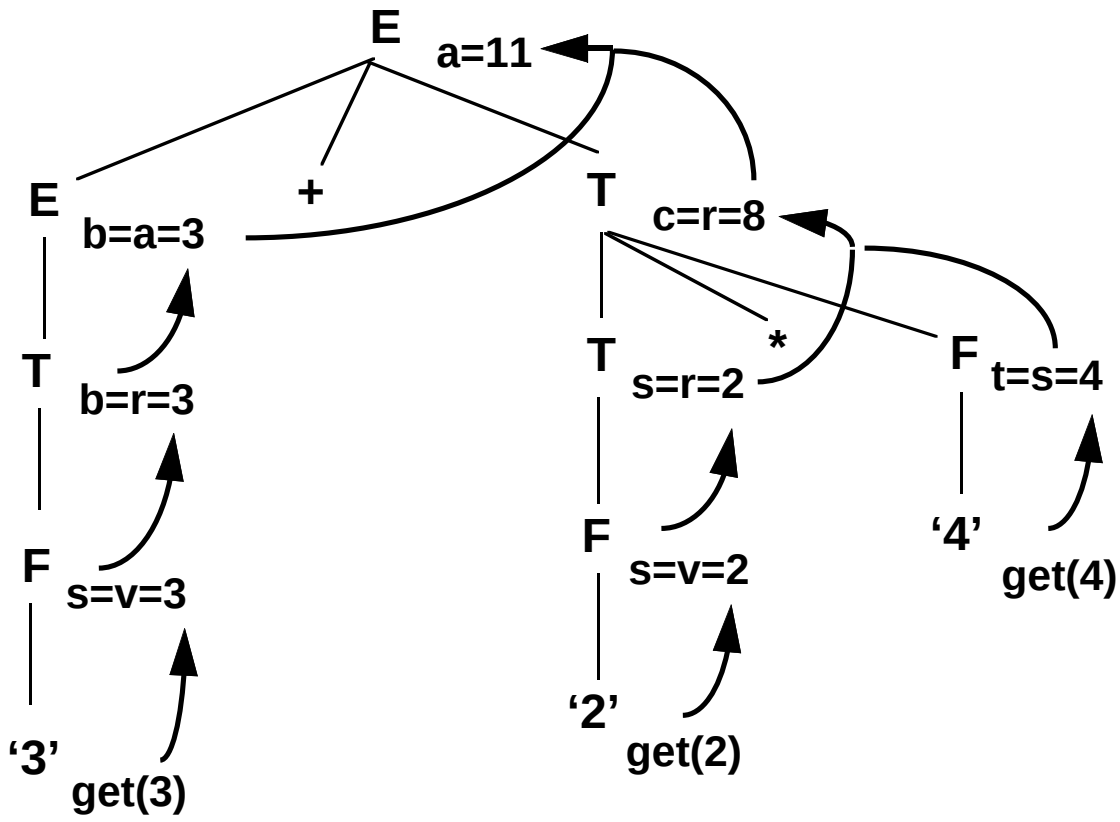
EXAMPLE (continued)

(a) Create the parse tree



(b) Now add the attribute information

EXAMPLE (continued)

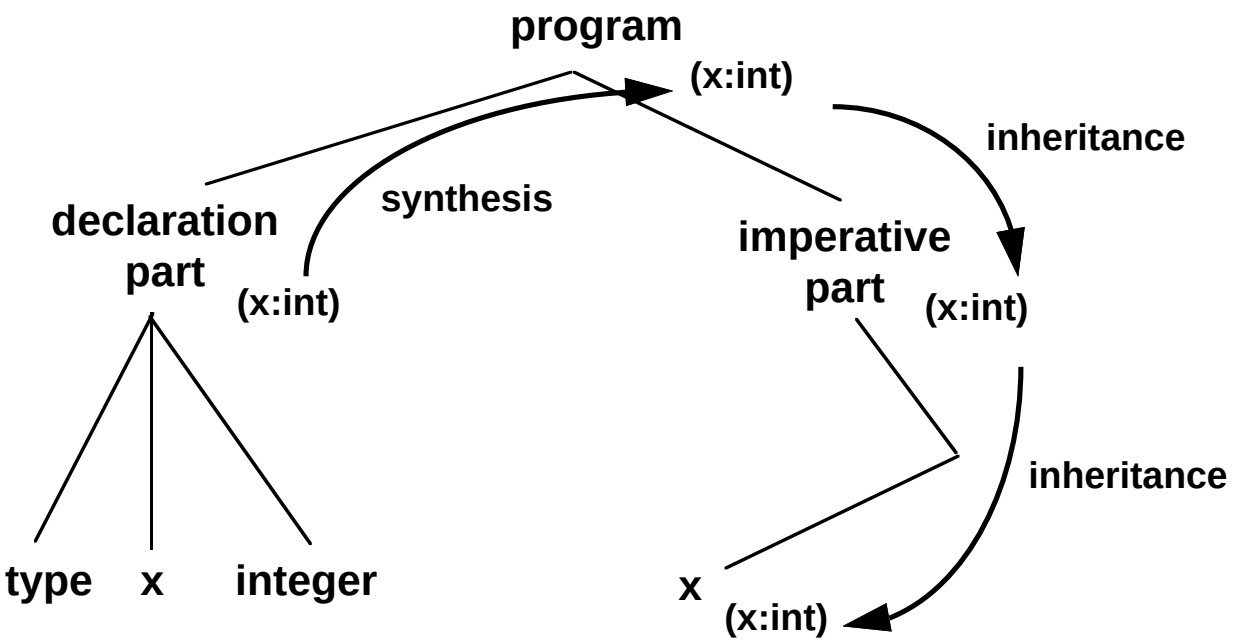


Note that in most cases of "interior nodes" each element has two names - one when it is the root symbol for the subtree and one when it is a leaf of a higher tree

ATTRIBUTE RULES

Attribute rules can act in either direction:

s <- p implies a movement of attributes
UP the tree (synthesis)
s -> p implies a movement of attributes
DOWN the tree (inheritance)



EXTENDED EXAMPLE GRAMMAR

- **get(v), where v is an identifier, returns a pair (g_1,g_2) where g_1 is the value associated with v and g_2 is the type possessed by v (R,I)**
- **float(v) returns a real representation of the value associated with v**
- **$+_R$ = real addition**
  $+_I$ = integer addition
- **Grammar:**

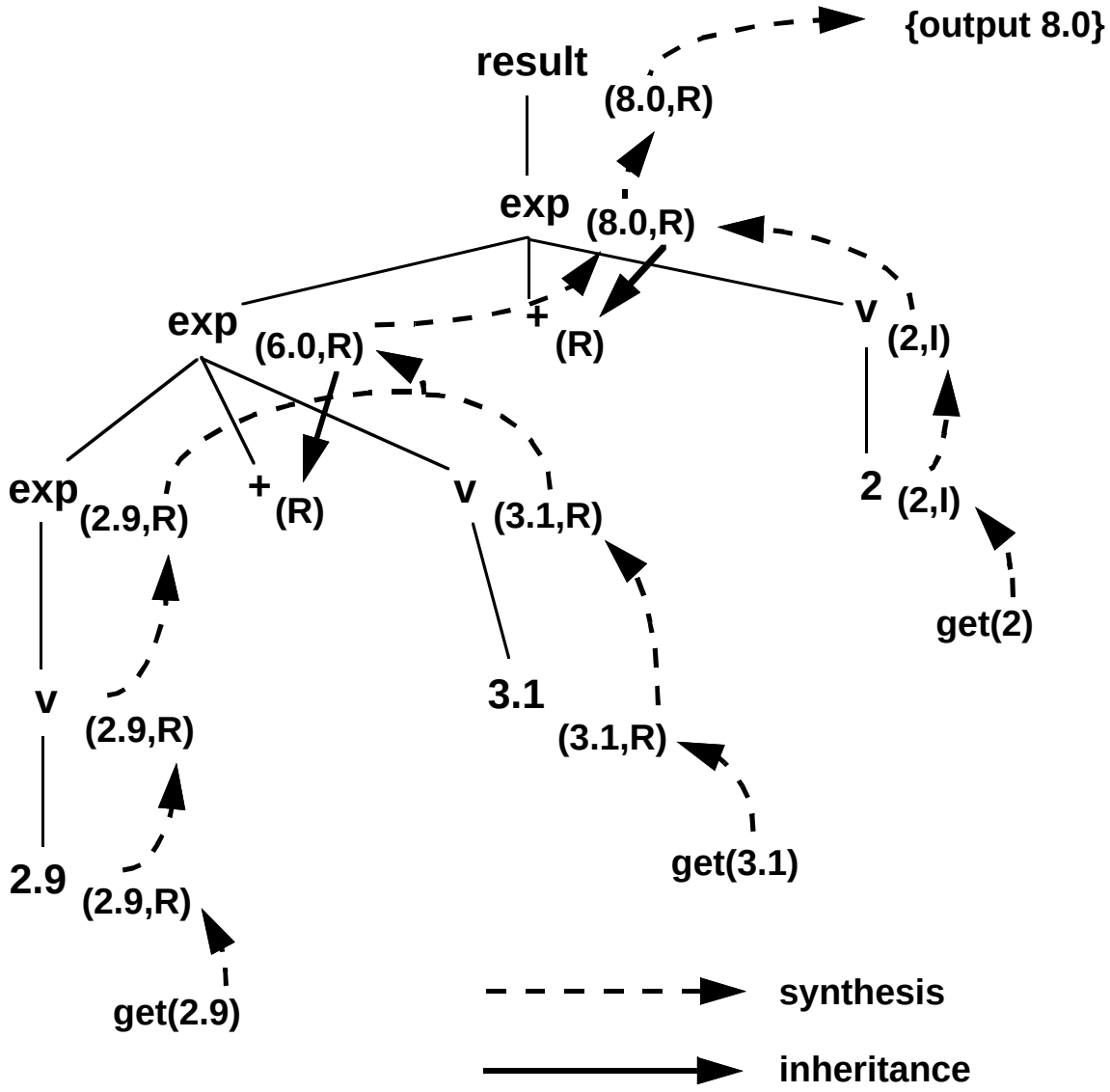
→ result $_w$ -> exp $_v$ {x} where {} indicates output
w <- v
w -> x

exp $_a$ -> exp $_c$ + $_d$ v $_f$
if $c_2 = f_2$ then a_2 <- c_2
a_2 -> d
a_1 <- $c_1 +_d$ f_1
elsif c_2 = R then a_2 <- R
a_2 -> d
a_1 <- c_1 + $_d$ float(f_1)
else a_2 <- R
a_2 -> d
a_1 <- float(c_1) + $_d$ f_1

exp$_a$ -> v$_h$
a <- h

v$_g$ -> id
g <- get(id)

EXTENDED EXAMPLE GRAMMAR (cont'd)



ANOTHER SIMPLE EXAMPLE

- **Semantics of integer binary strings:**

B_a -> B_b D_c
a <- bx2 + c

B_a -> D_c
a <- c

D_e -> 0
e <- get('0')

D_f -> 1
f <- get ('1')

- **Binary fraction:**

F_a -> .B_b
a <- b/2

B_c -> D_d B_e
c <- d + e/2

B_c -> D_d
c <- d

D_f -> 0
f <- get('0')

D_f -> 1
f <- get('1')