Semantic Analysis

- Phase computing additional information (necessary for compiling), known the syntactical structure of the program

- Computation of information that cannot be computed by syntax methods

- Syntax = { Structural rules governing the construction of phrases } → Call to `P()` preceded by def ?

- Double meaning of semantics
  - checking: which phrases are semantically correct?
  - computation: what does the program compute? *requires*

Semantic Analysis (ii)

- Staticity of semantic analysis: since information is computed before execution

  Typically: $\left\langle \begin{array}{l}\text{construction of symbol table} \rightarrow \text{trace of meaning of names} \\ \text{inference/check of types for expressions/statements to establish correctness}\end{array}\right.$

- Amount of analysis depending on the degree of language staticity
  (increasing staticity: *Lisp* → *C* → *Pascal* → *Ada*)

- Parallelism with lexical/syntax analysis: tools for $\left\langle \begin{array}{l}\text{specification} \\ \text{analysis}\end{array}\right.$

- Difference: $\nexists$ standard (Regexpr / BNF)

Specification

- **Attribute grammar** = syntax rules $\cup$ semantic rules

$\Downarrow$

Appropriate for PL guided by the principle of **syntax-directed semantics**:
semantic content of a program strictly related to its syntax

- **Semantic rule** $\equiv$ equation of semantic attributes
- **Attribute** $\equiv$ property of a language entity
- Problems when defining an attribute grammar (AG):

1. AG not provided by the designer of the language $\rightarrow$ written by the designer of the compiler
2. AG unnecessary complicated in order to adhere to the concrete syntax of the language
3. Abstract syntax not provided by the user manual of the language

Analysis

- $\nexists$ clear and standardized algorithms as for syntax analysis (top-down, bottom-up, ...)

- Simplification: when computation <u>after</u> (complete) construction of syntax tree

$$\Downarrow$$

Sufficient specifying $\left\langle \begin{array}{l} \text{traversing order of syntax tree} \\ \text{computation associated with each node} \end{array} \right.$

- $\nexists$ standard tools for automatic generation of semantic analyzers

Attributes

- Attribute = any property of a language construct

  Exmp: **variable**(*name*, *address*, *type*, *value*, *lifetime*, *scope*)

- Variance of attributes w.r.t.
  - type of information
  - complexity
  - binding time
    - language design
    - compiler design
    - compiling
    - run

- Examples of attributes:
  1. Type of variable (compile-time $\rightarrow$ type checker)
  2. Value of expression (run-time, unless static expr)
  3. Address of variable (either compile-time or run-time)

Attribute Grammars

- Principle: attributes associated with grammar symbols of PL

- Given **X** = grammar symbol, **a** = attribute associated with **X** $\Rightarrow$ **X.a** = value of **a** associated with **X**

- **Principle of syntax-directed semantics**:

  Given a collection of attributes { a_1, ..., a_k } associated with grammar symbols,
  $\forall$ grammar rule $X_0 \to X_1\ X_2\ \ldots\ X_n$, the values of attributes $X_i.a_j$ of each grammar symbol X_i depend on the values of the attributes of the symbols <u>within the grammar rule</u>"

$$\Downarrow$$

$$X_i.a_j = \boldsymbol{f}_{ij}(X_0.a_1,\ \ldots,\ X_0.a_k,\ X_1.a_1,\ \ldots,\ X_1.a_k,\ \ldots,\ X_n.a_1,\ \ldots,\ X_n.a_k)$$

- In theory: AG very complex; in practice $\boldsymbol{f}_{ij}$ simple, $X_i.a_j = \boldsymbol{f}_{ij}$(few attributes)

Attribute Grammars (ii)

- AG expressed in tabular form:

Production	*Semantic rules*
P_1	$\{ E_{11}, E_{12}, \ldots, E_{1m_1} \}$
P_2	$\{ E_{21}, E_{22}, \ldots, E_{2m_2} \}$
...	...
P_n	$\{ E_{n1}, E_{n2}, \ldots, E_{nm_n} \}$

Attribute Grammars (iii)

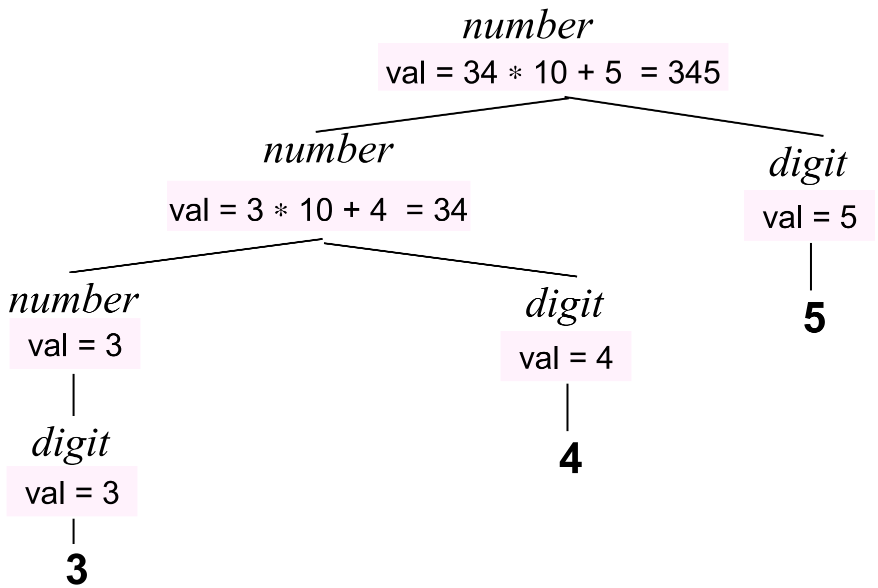
1. $number \to number\ \ digit \mid digit$

   $digit \to$ **0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**

⇨ val = meaningful attribute of a number

Production	*Semantic rules*
$number_1 \to number_2\ \ digit$	$number_1$.val = $number_2$.val $*$ 10 + $digit$.val
$number \to digit$	$number$.val = $digit$.val
$digit \to$ **0**	$digit$.val = 0
$digit \to$ **1**	$digit$.val = 1
...	...
$digit \to$ **9**	$digit$.val = 9

345



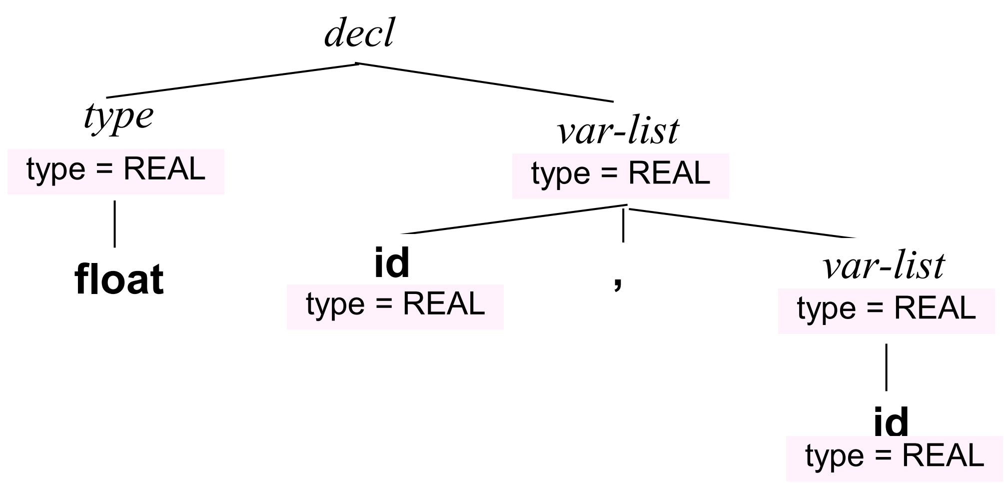
Attribute Grammars (iv)

2. *decl* → *type var-list*
type → **int** | **float**
var-list → **id,** *var-list* | **id**

⇨ type (INT, REAL) = meaningful attribute of a variable

Production	*Semantic rules*
decl → *type var-list*	*var-list*.type = *type*.type
type → **int**	*type*.type = INT
type → **float**	*type*.type = REAL
$var\text{-}list_1$ → **id,** $var\text{-}list_2$	**id**.type = $var\text{-}list_1$.type $var\text{-}list_2$.type = $var\text{-}list_1$.type
var-list → **id**	**id**.type = *var-list*.type

float x, y



Attribute Grammars (v)

3.
$qnum \rightarrow num\ qualifier$
$qualifier \rightarrow$ **o** | **d**
$num \rightarrow num\ digit \mid digit$
$digit \rightarrow$ **0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**

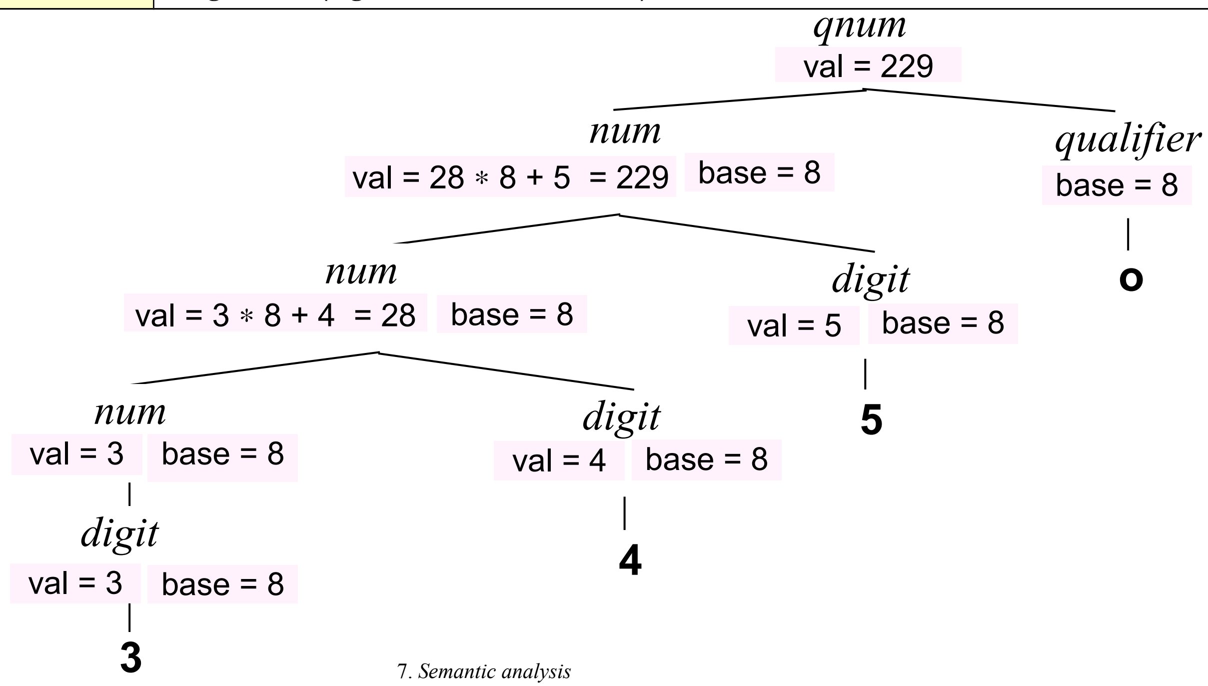
⇨ **345o**
229d → **229o** : error! (detectable with ≠ syntax)

⇨ necessary base to compute value : A = { base, val }

Production	*Semantic rules*
$qnum \rightarrow num\ \ qualifier$	$qnum$.val = num.val num.base = $qualifier$.base
$qualifier \rightarrow$ **o**	$qualifier$.base = 8
$qualifier \rightarrow$ **d**	$qualifier$.base = 10
$num_1 \rightarrow num_2\ \ digit$	num_1.val = ($digit$.val = error **or** num_2.val = error ? error : num_2.val $*$ num_1.base + $digit$.val) num_2.base = num_1.base $digit$.base = num_1.base
$num \rightarrow\ digit$	num.val = $digit$.val $digit$.base = num.base
$digit\ \rightarrow$ **0**	$digit$.val = 0
$digit\ \rightarrow$ **1**	$digit$.val = 1
…	**…**
$digit\ \rightarrow$ **7**	$digit$.val = 7
$digit\ \rightarrow$ **8**	$digit$.val = ($digit$.base = 8 ? error : 8)
$digit\ \rightarrow$ **9**	$digit$.val = ($digit$.base = 8 ? error : 9)

Attribute Grammars (vi)

Production	Semantic rules
$qnum \rightarrow num\ \ qualifier$	$qnum$.val = num.val num.base = $qualifier$.base
$qualifier \rightarrow$ **o**	$qualifier$.base = 8
$qualifier \rightarrow$ **d**	$qualifier$.base = 10
$num_1 \rightarrow num_2\ \ digit$	num_1.val = ($digit$.val = error **or** num_2.val = error ? error : num_2.val $*$ num_1.base + $digit$.val) num_2.base = num_1.base $digit$.base = num_1.base
$num \rightarrow digit$	num.val = $digit$.val $digit$.base = num.base
$digit \rightarrow$ **0**	$digit$.val = 0
$digit \rightarrow$ **1**	$digit$.val = 1
...	...
$digit \rightarrow$ **7**	$digit$.val = 7
$digit \rightarrow$ **8**	$digit$.val = ($digit$.base = 8 ? error : 8)
$digit \rightarrow$ **9**	$digit$.val = ($digit$.base = 8 ? error : 9)



345o

Attribute Grammars (vii)

- Problem: semantics for PL $\rightarrow$ semantics for language of attribute equations

- **Metalanguage** $\equiv$ { expressions to specify semantic rules }

  Properties: 1. Clear semantics (meta-semantics)
  2. Compatible with implementation L of compiler, because need for mapping:

  Attribute equations $\rightarrow$ Compiler code

- Our context: Metalanguage $\supseteq$ { arithmetic expr, logical expr, conditional statements }

- Further possibility: use of functions whose body is specified somewhere else

digit $\rightarrow$ **c**	*digit*.val = `numval`(**c**)

```
int numval(char c)
{
  return((int)c – (int)'0');
}
```

Computation of Attributes

- Basic problem: transformation of attribute equations into computational steps

- Semantic rules = equality relations on elements in $\{ X_0, X_1, \ldots, X_n \} \times \{ a_1, \ldots, a_k \}$ = pairs (X_i, a_j)

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

- Key step: equation viewed as <u>assignment</u> of $X_i.a_j$ with the RHS

$$\Downarrow$$

Values in RHS: shall be already computed!

- Pb specification of algorithm for resolution of system of semantic equations relevant to an AG: corresponds to the determination of an <u>assignment order</u> of attributes assuring the availability of involved values

- Precedence constraints for attribute computation expressed by dependency graphs

Dependency Graphs

- <u>Def</u>: Given an AG, each <u>grammar rule</u> (alternative) is associated with a dependency graph so defined:

  1. $\exists$ a node $\forall$ attribute $X_i.a_j$ of each symbol X_i within the grammar rule;
  2. $\forall$ equation $X_i.a_j = f_{ij}(\ldots, X_h.a_k, \ldots)$ associated with the grammar rule, $\exists$ an arc $X_i.a_j \leftarrow X_h.a_k$ from each node in RHS of equation to the node in LHS of equation.

- <u>Def</u>: A dependency graph associated with a <u>phrase</u> of L(G) is the composition (union) of the dependency graphs associated with the grammar rules (alternatives) relevant to (internal) nodes of the syntax tree of the phrase.

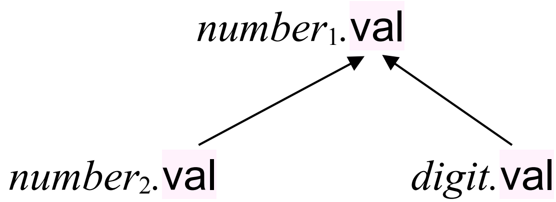
$\Downarrow$

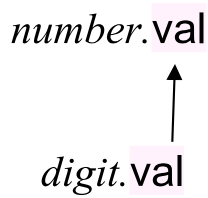
Identifies a set of "trajectories" to compute the semantics of the phrase

Dependency Graphs (ii)

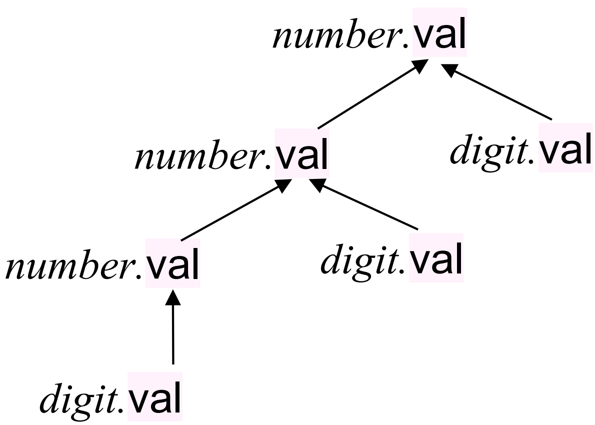
1.

Production	*Semantic rules*
$number_1 \rightarrow number_2\ \ digit$	$number_1$.val = $number_2$.val $*$ 10 + $digit$.val
$number \rightarrow$ *digit*	$number$.val = $digit$.val
$digit \rightarrow$ **0**	$digit$.val = 0
$digit \rightarrow$ **1**	$digit$.val = 1
...	**...**
$digit \rightarrow$ **9**	$digit$.val = 9





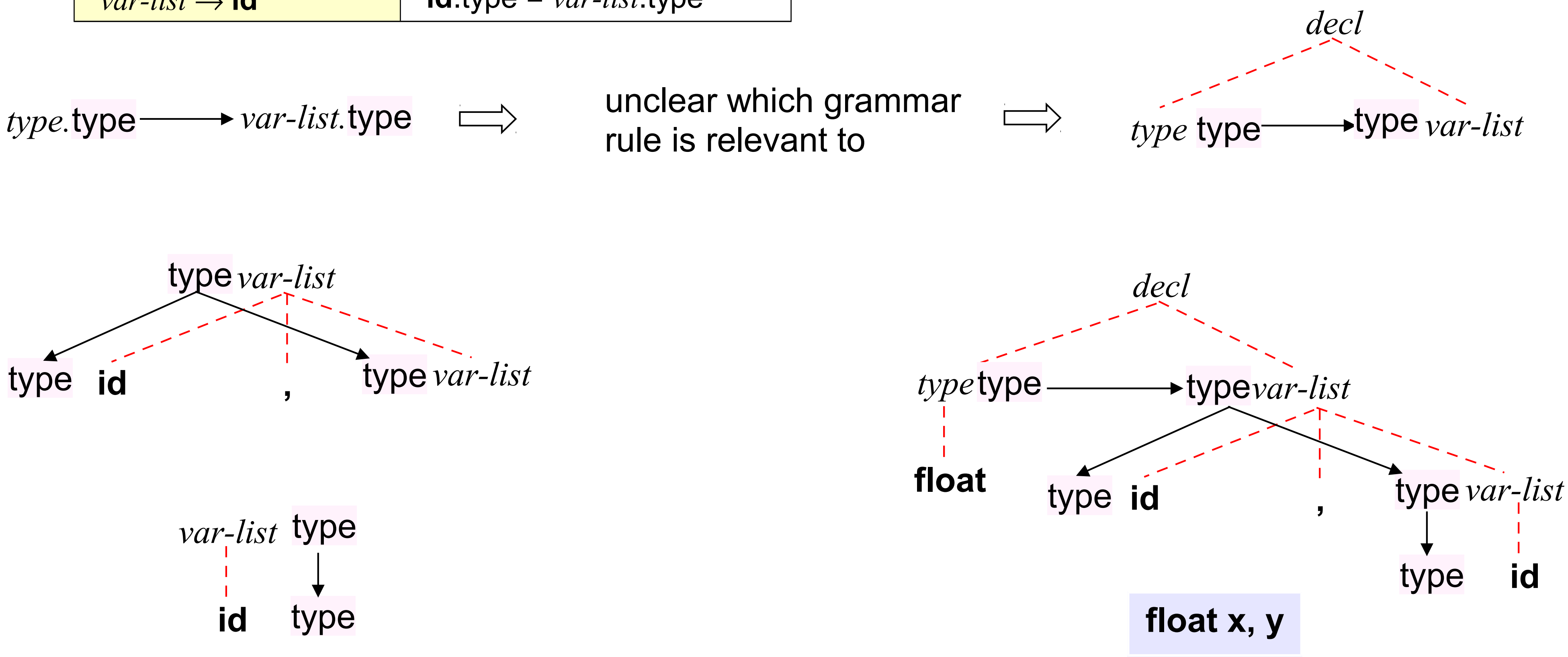
Dependency graph associated with phrase **345**



Dependency Graphs (iii)

2.

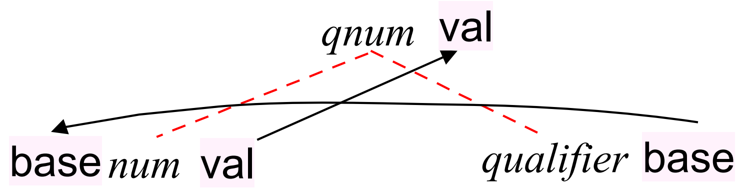
Production	*Semantic rules*
decl → *type var-list*	*var-list*.type = *type*.type
type → **int**	*type*.type= INTEGER
type → **float**	*type*.type = REAL
$var\text{-}list_1$ → **id**, $var\text{-}list_2$	**id**.type = $var\text{-}list_1$.type $var\text{-}list_2$.type = $var\text{-}list_1$.type
var-list → **id**	**id**.type = *var-list*.type

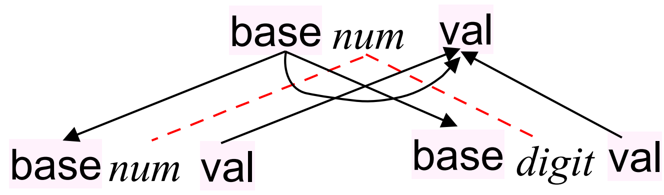


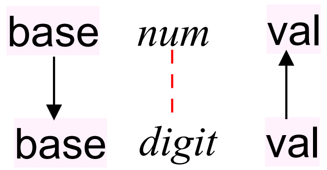
Dependency Graphs (iv)

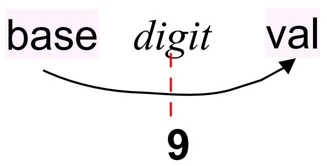
3.

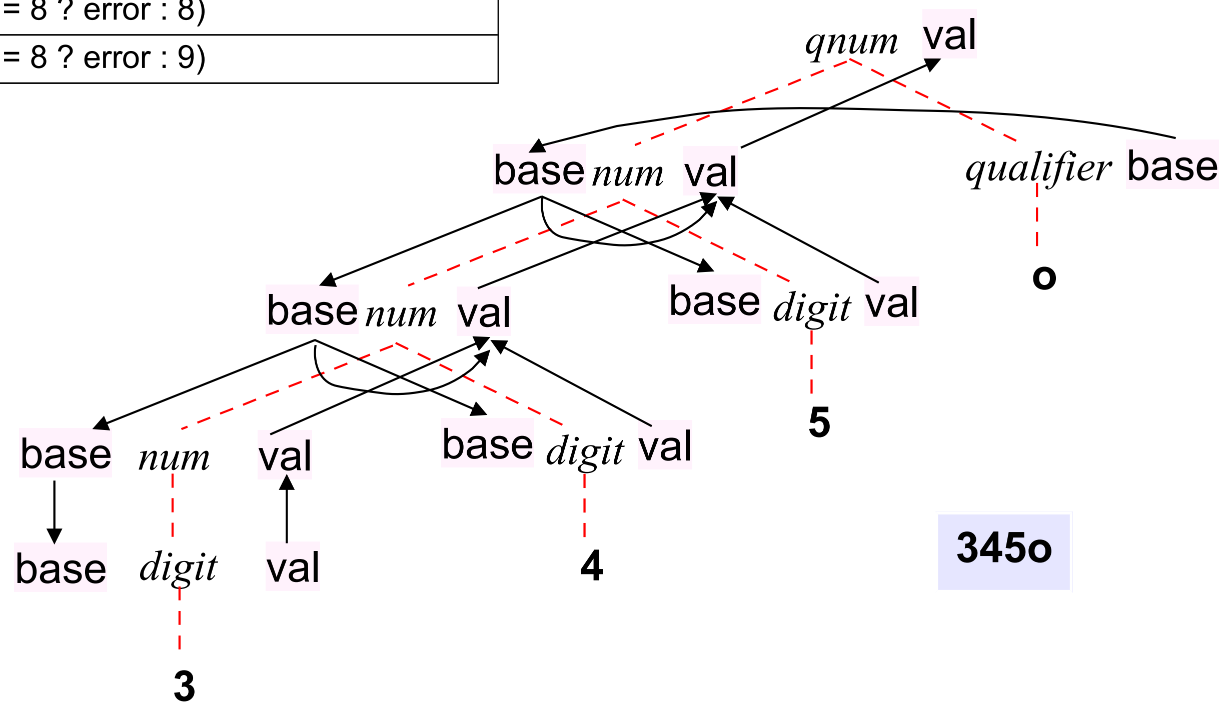
Production	Semantic rules
$qnum \rightarrow num\ \ qualifier$	$qnum$.val = num.val num.base = $qualifier$.base
$qualifier \rightarrow$ **o**	$qualifier$.base = 8
$qualifier \rightarrow$ **d**	$qualifier$.base = 10
$num_1 \rightarrow num_2\ \ digit$	num_1.val = ($digit$.val = error **or** num_2.val = error ? error : num_2.val $*$ num_1.base + $digit$.val) num_2.base = num_1.base $digit$.base = num_1.base
$num \rightarrow digit$	num.val = $digit$.val $digit$.base = num.base
$digit \rightarrow$ **0**	$digit$.val = 0
…	…
$digit \rightarrow$ **8**	$digit$.val = ($digit$.base = 8 ? error : 8)
$digit \rightarrow$ **9**	$digit$.val = ($digit$.base = 8 ? error : 9)











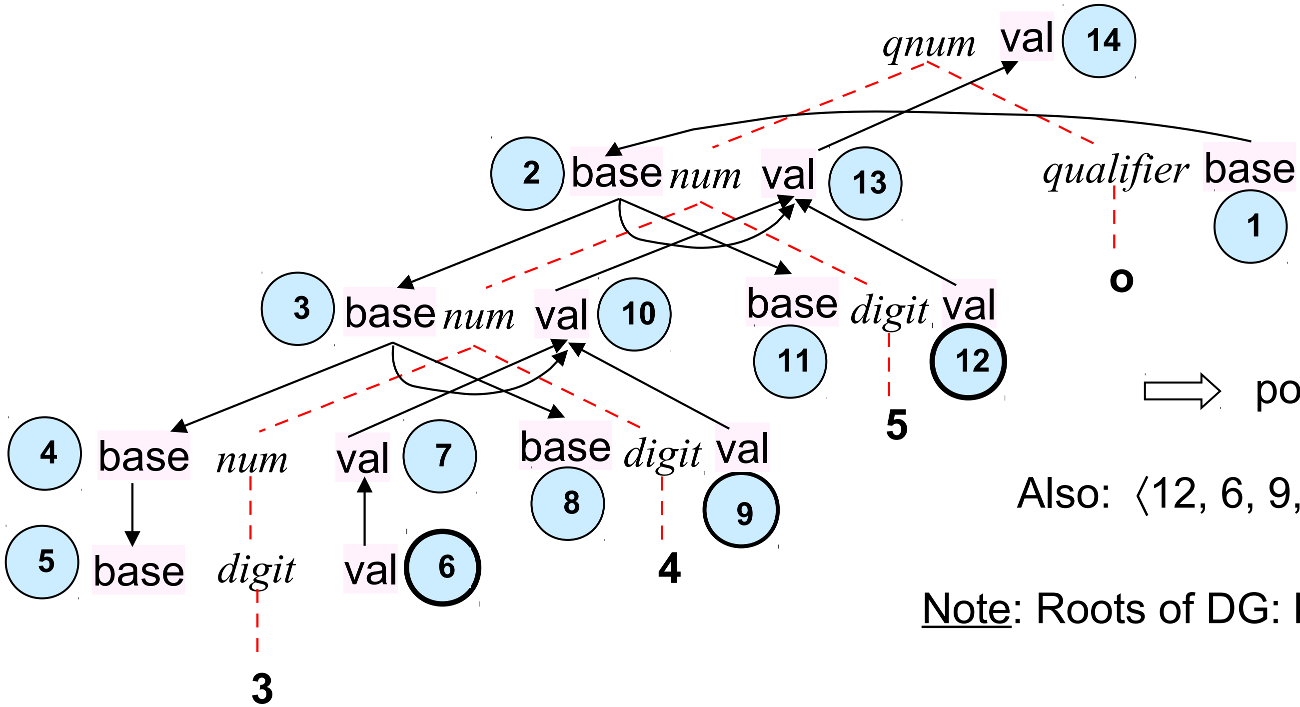
Dependency Graphs (v)

- Dependency graph associated with phrase: establishes precedence constraints for computation of attributes

- **Topological sort** ≡ An order of DG that fulfills the precedence constraints

$$\|\, DG \,\| = \{ TS_1, TS_2, \ldots, TS_n \} \text{ (finite!)}$$

- N&S condition for existence of topological sort in DG = <u>acyclicity</u> of DG (DAG)
  (otherwise: $\|\, DG \,\| = \varnothing$)



⟹ possible topological sort

Also: $\langle 12, 6, 9, 1, 2, 11, 3, 8, 4, 5, 7, 10, 13, 14 \rangle$

<u>Note</u>: Roots of DG: known attribute values

Dependency Graphs (vi)

- Def: AG is **noncircular** when DG of every phrase is acyclic.

- Two types of algoritms for attribute computation:

1. Parse-tree method:

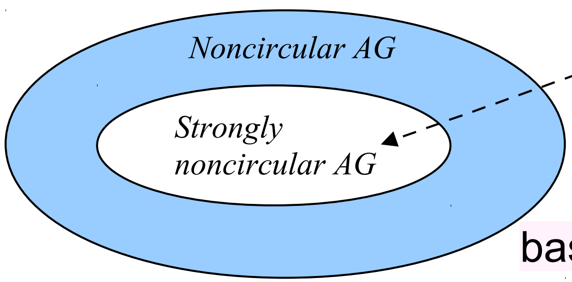
- Construction of (decorated) syntax tree
- Determination of a topological sort (*compile-time*)

Problems: a) Complexity of construction of DG + TS at compile-time
b) Discovery of a circularity → wrong AG ⇨ AG to be tested before! (∃ algorithms)

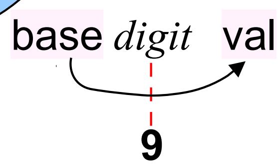
2. Rule-based method: Determination of evaluation order of attributes a priori (*compiler-construction-time*)

⇓

independent of specific syntax tree → independent of specific phrase of L(G)!



in practice: all "reasonable" AG



Note: Previous example: nodes 6, 9, 12: roots → in theory, might be at the beginning of a topological sort.
Instead: rule-based method: cannot be roots since values 8 and 9 depend on base too.

Synthesized Attributes

- Attribute evaluation: by a traversal of syntax tree $\rightarrow$ possible in different ways: depends on the type of semantic rules $\rightarrow$ <u>attribute classification</u>

- <u>Def</u>: An attribute is **synthesized** if relevant to a nonterminal in LHS of the syntax rule associated with the semantic equation:

$$A \rightarrow X_1\ X_2\ \ldots\ X_n \quad \Rightarrow \quad A.a = f\,(X_1.a_1,\ \ldots,\ X_1.a_k,\ \ldots,\ X_n.a_1,\ \ldots,\ X_n.a_k)$$

- <u>Def</u>: AG where all attributes are synthesized is called an **S-attributed grammar**

$\Downarrow$

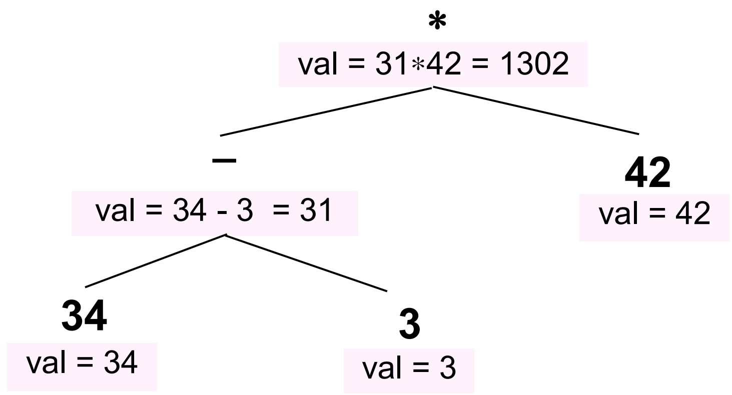
<u>Property</u>: Attribute values computable by a single bottom-up step (post-order)

```
procedure PostEval(N: node)
begin
   for each child C of N do
      PostEval(C);
   Compute all synthesized attributes of N
end.
```

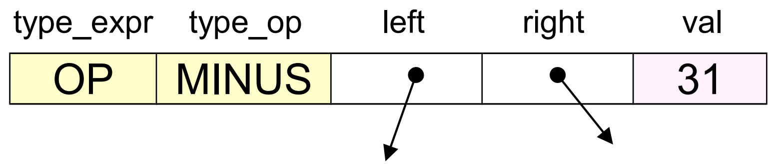
Synthesized Attributes (ii)

(34 – 3) ∗ 42

Production	*Semantic rules*
$expr_1 \rightarrow expr_2 + expr_3$	$expr_1$.val = $expr_2$.val + $expr_3$.val
$expr_1 \rightarrow expr_2 - expr_3$	$expr_1$.val = $expr_2$.val - $expr_3$.val
$expr_1 \rightarrow expr_2 * expr_3$	$expr_1$.val = $expr_2$.val ∗ $expr_3$.val
$expr_1 \rightarrow$ **(** $expr_2$ **)**	$expr_1$.val = $expr_2$.val
$expr \rightarrow$ **num**	$expr$.val = **num**.val

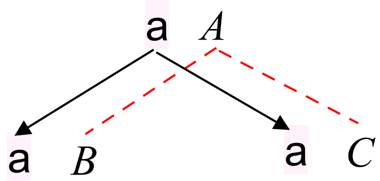


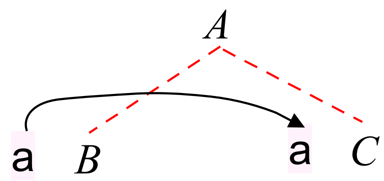

```
typedef enum {OP, CONST} TypeExpr;
typedef enum {PLUS, MINUS, MUL} TypeOp;
typedef struct tnode {
    TypeExpr type_expr;
    TypeOp type_op;
    struct tnode *left, *right;
    int val;
} Node;

postEval(Node *p)
{
  if(p->type_expr == OP)
  {
    postEval(p->left); postEval(p->right);
    switch(p->type_op)
    {
      case PLUS: p->val = p->left->val + p->right->val; break;
      case MINUS: p->val = p->left->val - p->right->val; break;
      case MUL: p->val = p->left->val * p->right->val; break;
    }
  }
  else return;    /* type_expr = CONST: values available */
}
```



Inherited Attributes

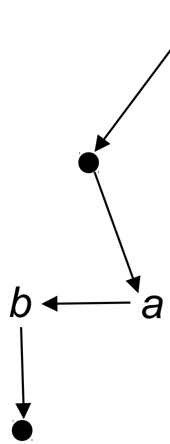
- Def: A non-synthesized attribute is called an **inherited** attribute
- Def: An AG where all attributes are inherited is called **I-attributed grammar**
- Note: Dependencies
  - parent $\rightarrow$ child
  - sibling $\rightarrow$ sibling





Property: Attribute values computable by a single top-down traversal (pre-order)

```
procedure PreEval(N: node)
begin
   for each child C of N do
   begin
      Compute all inherited attributes of C;
      PreEval(C)
   end
end.
```



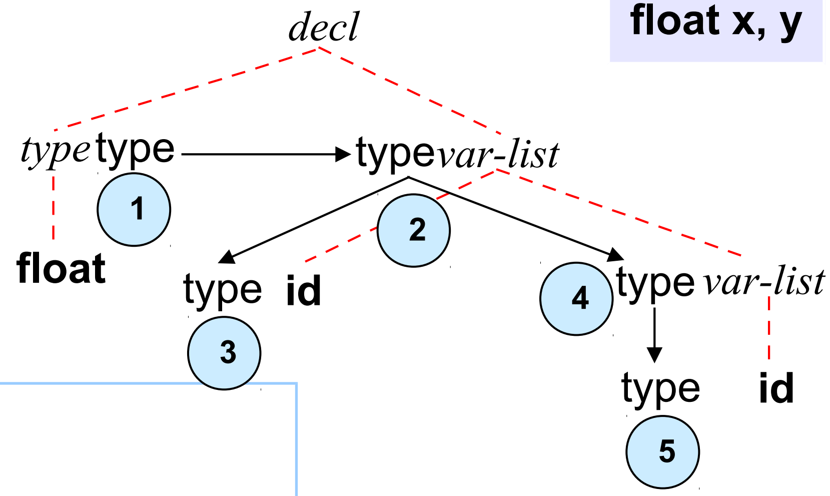
$\Rightarrow$ *a* before *b*

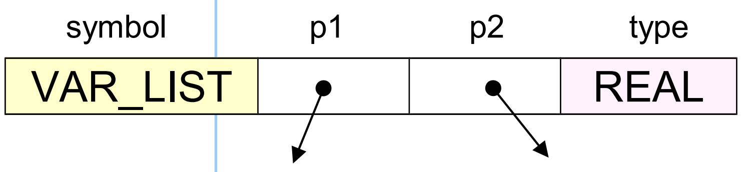
Note: Evaluation order of attributes: important! (owing to dependencies among siblings)

Inherited Attributes (ii)

Production	*Semantic rules*
decl → *type var-list*	*var-list*.type = *type*.type
type → **int**	*type*.type = INTEGER
type → **float**	*type*.type = REAL
var-list$_1$ → **id,** *var-list*$_2$	**id**. type = *var-list*$_1$. type *var-list*$_2$.type = *var-list*$_1$.type
var-list → **id**	**id**.type = *var-list*.type

float x, y






```
typedef enum {DECL, TYPE, VAR_LIST, INT, FLOAT, ID} Symbol;
typedef enum {INTEGER, REAL} Type;
typedef struct tnode
{ Symbol symbol;
  struct tnode *p1, *p2;
  Type type;
} Node;

evalType(Node *p)
{  switch(p->symbol){
   case DECL:     evalType(p->p1);
                  p->p2->type = p->p1->type;
                  evalType(p->p2); break;
   case TYPE:     p->type = (p->p1->symbol == INT ? INTEGER : REAL); break;
   case VAR_LIST: p->p1->type = p->type;
                  if(p->p2 != NULL){
                     p->p2->type = p->type;
                     evalType(p->p2);
                  }
                  break;
  }
}
```

Mixed Attributes

- AG with attributes
  - synthesized
  - inherited, such that i <u>not</u> $f(s)$ (else: several traversals of decorated tree)

```
procedure CombinedEval(N: node)
begin
   for each child C of N do
   begin
      Compute all inherited attributes of C;
      CombinedEval(C)
   end
   Compute all synthesized attributes of N
end.
```

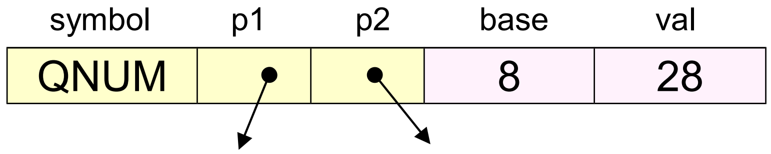
unique difference wrt *PostEval* (synthesized): computation of inherited when decending

unique difference wrt *PreEval* (inherited): computation of synthesized when ascending

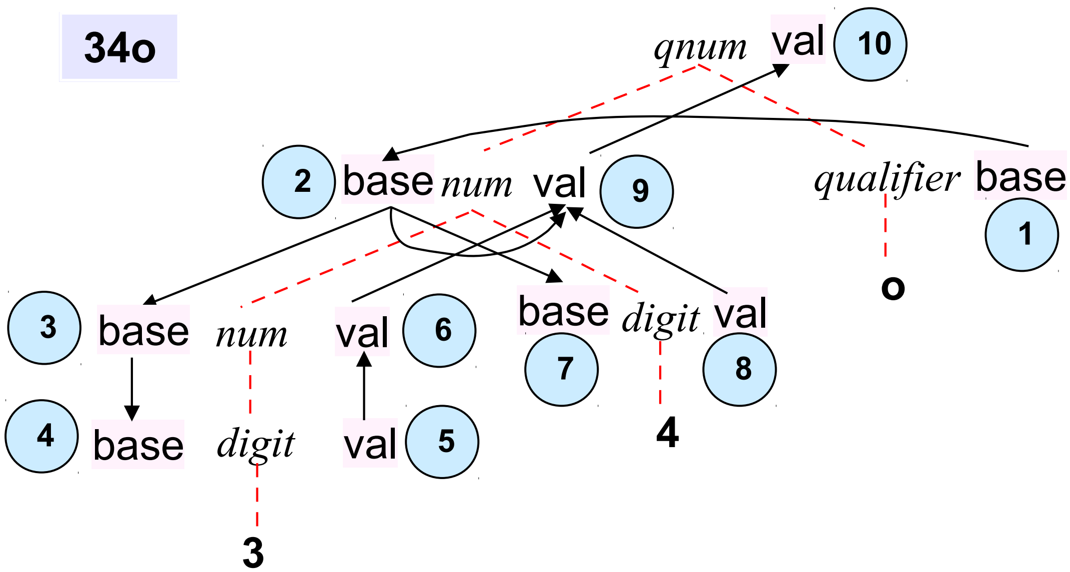
inherited ↓ ↑ synthesized

Mixed Attributes (ii)

Production	*Semantic rules*
$qnum \rightarrow num\ \ qualifier$	$qnum$.val = num.val num.base = $qualifier$.base
$qualifier \rightarrow$ **o**	$qualifier$.base = 8
$qualifier \rightarrow$ **d**	$qualifier$.base = 10
$num_1 \rightarrow num_2\ \ digit$	num_1.val = ($digit$.val = error **or** num_2.val = error ? error : num_2.val $*$ num_1.base + $digit$.val) num_2.base = num_1.base $digit$.base = num_1.base
$num \rightarrow\ digit$	num.val = $digit$.val $digit$.base = num.base
$digit\ \rightarrow$ **0**	$digit$.val = 0
…	…
$digit\ \rightarrow$ **8**	$digit$.val = ($digit$.base = 8 ? error : 8)
$digit\ \rightarrow$ **9**	$digit$.val = ($digit$.base = 8 ? error : 9)

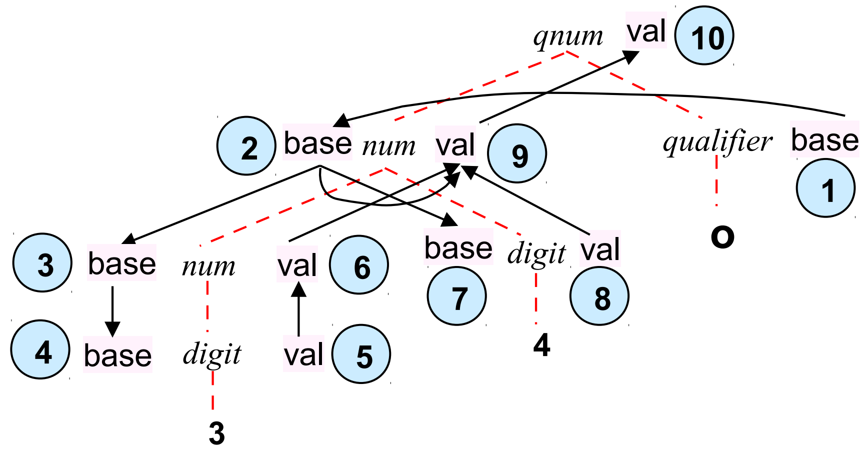



```
typedef struct tnode
{
    Symbol symbol;
    struct tnode *p1, *p2;
    int base, val;
} Node;
```



Mixed Attributes (iii)

```
evalQnum(Node *p)
{
  switch(p->symbol)
  {
   case QNUM: evalQnum(p->p2);
              p->p1->base = p->p2->base;
              evalQnum(p->p1);
              p->val = p->p1->val;
              break;

   case NUM:  p->p1->base = p->base;
              evalQnum(p->p1);
              if(p->p2 != NULL)
              {
                p->p2->base = p->base;
                evalQnum(p->p2);
                p->val = (p->p1->val != ERROR && p->p2->val != ERROR ?
                                p->p1->val * p->base + p->p2->val : ERROR);
              }
              else p->val = p->p1->val;
              break;

   case QUALIFIER: p->base = (p->p1->symbol == OCTAL ? 8 : 10) break;

   case DIGIT: p->val = (p->base == 8 && (p->p1->val == 8 || p->p1->val == 9) ? ERROR : p->p1->val);
               break;
  }
}
```



Attributes Stored within Activation Records of Functions

- Convenient when
  - many repeated attributes (type)
  - many attributes used as temporaries to compute other attributes (val)

$\Downarrow$

waste of space in syntax tree!

- In these cases: better using a recursive function mapping attributes
  - inherited $\rightarrow$ input parameters
  - synthesized $\rightarrow$ output parameters

$qnum \rightarrow num$ **qualifier**
$num \rightarrow num$ **digit** | **digit**

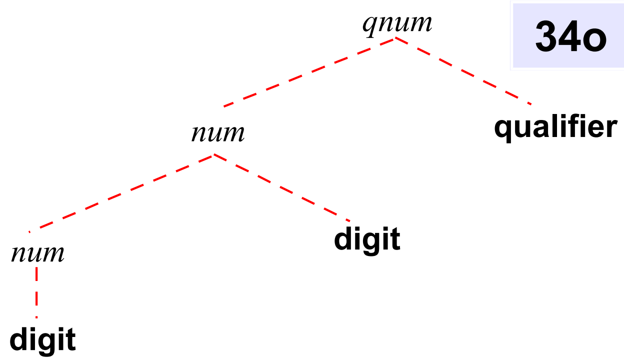
$\Rightarrow$ simplification of G: **digit**, **qualifier** = terminals

symbol	p1	p2	lexval
QUALIFIER	•	•	‘o’

lexval ← instantiated by lexer $\rightarrow \nexists$ semantic part!

Attributes Stored within Activation Records of Functions (ii)

```
typedef struct tnode
{   Symbol symbol;
    struct tnode *p1, *p2;
    char lexval;  /* for terminals */
} Node;

int valQnum(Node *p)  /* called on qnum */
{
  return(value(p->p1, qualifier(p->p2)));
}

int qualifier(Node *p) /* called on qualifier */
{
  return(p->lexval == 'o' ? 8 : 10);
}

int value(Node *p, int base)   /* called on num and digit */
{ int val1, val2;

  switch(p->symbol)
  {
  case NUM: val1 = value(p->p1, base);
            if(p->p2 != NULL)
            {
              val2 = value(p->p2, base);
              if(val1 != ERROR && val2 != ERROR) return(val1 * base + val2);
              else return(ERROR);
            }
            else return(val1);

  case DIGIT: val1 = (int)(p->lexval - '0');
              return(base == 8 && (val1 == 8 || val1 == 9) ? ERROR : val1);
  }
}
```



34o

Attributes Stored within External (Global) Structures

- Useful when same attributes necessary in different points during translation

- Use of data structures (tables, graphs, ...) to access attribute values

- Modified AG: attributes replaced by
  - global variables
  - calls to procedures manipulating the data structures representing attributes (surrogates)

⇓

"Extended" semantic rules: <u>no longer</u> representing an AG! (algorithmic evolution)

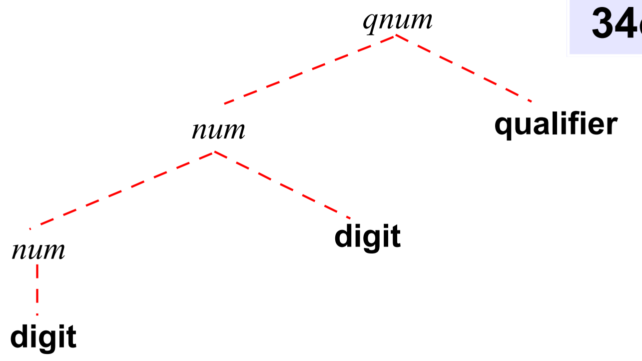
Attributes Stored within External (Global) Structures (ii)

1.

Production	*Semantic rules*
$qnum \to num\ qualifier$	$qnum$.val = num.val
$qualifier \to$ **o**	`base` := 8
$qualifier \to$ **d**	`base` := 10
$num_1 \to num_2$ **digit**	num_1.val = (**digit**.val = error **or** num_2.val = error ? error : num_2.val * `base` + **digit**.val)
$num \to$ **digit**	num.val = **digit**.val

```
int base;

void assignBase(Node *p)
{ base = (p->lexval == 'o' ? 8 : 10); }

int eval(Node *p)
{ int val1, val2;

  switch(p->symbol)
  {
  case QNUM: assignBase(p->p2);
             return(eval(p->p1));
  case NUM: val1 = eval(p->p1);
            if(p->p2 != NULL)
            {
              val2 = eval(p->p2);
              if(val1 != ERROR && val2 != ERROR) return(val1 * base + val2);
              else return(ERROR);
            }
            else return(val1);
  case DIGIT: val1 = (int)(p->lexval - '0');
              return(base == 8 && (val1 == 8 || val1 == 9) ? ERROR : val1);
  }
}
```

34o

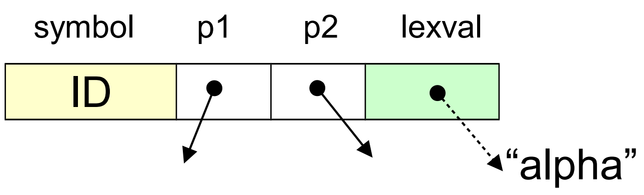


Attributes Stored within External (Global) Structures (iii)

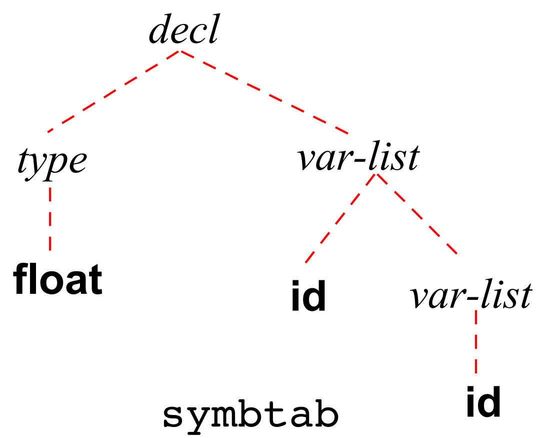
2. Symbol table
- `insert(name, type)`
- `lookup(name)`
- `delete(name)`

Production	*Semantic rules*
decl → *type var-list*	
type → **int**	`type` = INTEGER
type → **float**	`type` = REAL
var-list$_1$ → **id,** *var-list*$_2$	`insert(id.name, type)`
var-list → **id**	`insert(id.name, type)`

```
typedef struct tnode
{ Symbol symbol;
  struct tnode *p1, *p2;
  char *lexval;
} Node;

int type;

evalType(Node *p)
{ switch(p->symbol)
  {
   case DECL: evalType(p->p1);
              evalType(p->p2);
              break;

   case TYPE: type = (p->p1->symbol == INT ? INTEGER : REAL);
              break;

   case VAR_LIST: insert(p->p1->lexval, type);
                  if(p->p2 != NULL)
                     evalType(p->p2);
                  break;
  }
}
```



float alpha, beta



`symbtab`

alpha	REAL
beta	REAL
…	…

Type Checking

- Important task of compiler ⟨ **type inference** / **type checking** ⟩ ≈ type checking ⟨ **static** / dynamic ⟩
- Data type defined by type expression ⟨ simple `integer` / structured `array [1..100] of real` ⟩
- Information on types distributed in ≠ points of program → declaration of:
  - Variables `var x: array [1..100] of real;`
  - Types `type Vector = array [1..100] of real;`

    } explicit information on types
  - Constants `const ERROR = "Syntax error";` Implicitly: `array [1..12] of char`
- Information on types in ST: exploited by type checker when name referenced

`a[i]` = expr involving ⟨ `a` → `array [1..100] of real` / `i` → `integer` ⟩ ⟹ `a[i]`→ **`real`**

Type Equivalence

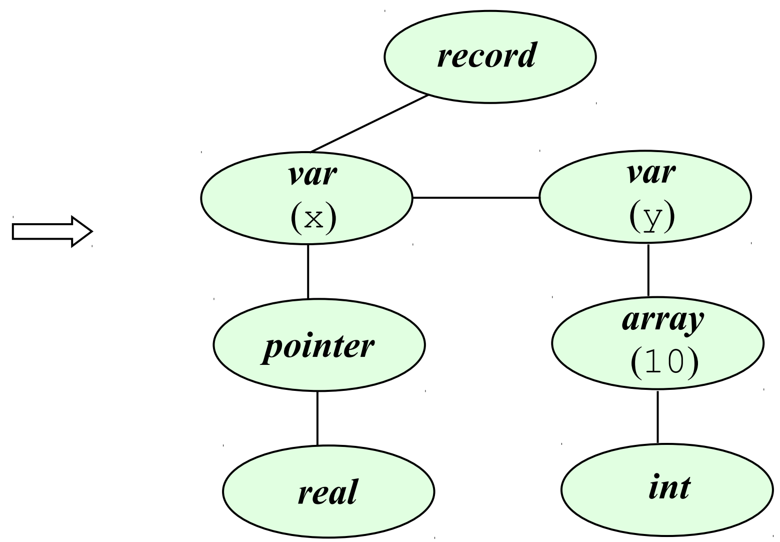
- Typical pb of type checker: check whether two type expr represent same type (equivalent)

- Equivalence criteria
  - **structural**
  - **based on names**

- Equivalence check: `function typeEqual(t1, t2: TypeExp): boolean;`

- Representation of types within compiler → abstract tree

```
record
    x: pointer to real;
    y: array [10] of int
end;
```

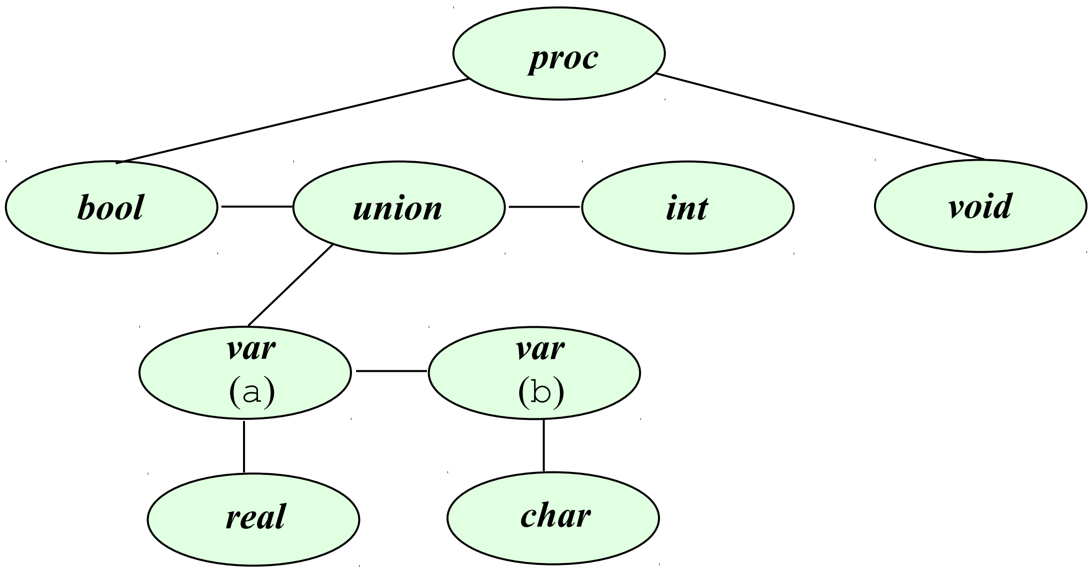


Attribute Grammar for Type Expressions

var-decls → *var-decls* **;** *var-decl* | *var-decl*
var-decl → **id :** *type-exp*
type-exp → *simple-type* | *structured-type*
simple-type → **int** | **bool** | **real** | **char** | **void**
structured-type → **array [num] of** *type-exp* |
record *var-decls* **end** |
union *var-decls* **end** |
pointer to *type-exp* |
proc(*type-exps***):** *type-exp*
type-exps → *type-exps* **,** *type-exp* | *type-exp*

`proc(bool, union a: real; b: char end, int): void`

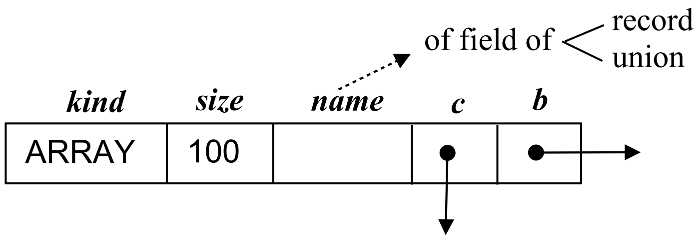
⇓



Check of Structural Equivalence

- Pragmatically: syntax trees of types → same structures (isomorphic)

```
function  typeEqual(t1, t2: TypeExp): boolean;
var ok: boolean;
    p1, p2: TypeExp;
begin
  if simpleType(t1) and simpleType(t2) then return (t1.kind = t2.kind);
  else if t1.kind = ARRAY and t2.kind = ARRAY then return (t1.size = t2.size and typeEqual(t1.c, t2.c));
  else if (t1.kind = RECORD and t2.kind = RECORD) or (t1.kind = UNION and t2.kind = UNION) then
    begin
      p1 := t1.c; p2 := t2.c; ok := true;
      while ok and p1 != nil and p2 != nil do
        begin
          if p1.name != p2.name then ok := false;
          else if not typeEqual(p1.c, p2.c) then ok := false;
          else begin p1 := p1.b; p2 := p2.b end
         end;
      return (ok and p1 = nil and p2 = nil)
    end;
  else if t1.kind = POINTER and t2.kind = POINTER then return (typeEqual(t1.c, t2.c));
  else if t1.kind = PROC and t2.kind = PROC then
    begin
      p1 := t1.c; p2 := t2.c; ok := true;
      while ok and p1 != nil and p2 != nil do
        begin
          if not typeEqual(p1.c, p2.c) then ok := false;
          else begin p1 := p1.b; p2 := p2.b end
      end;
      return (ok and p1 = nil and p2 = nil and typeEqual(t1.b, t2.b))
    end;
  else return (false)
end.
```



Attribute Grammar for Type Checking

program → *var-decls* **;** *stmts*
var-decls → *var-decls* **;** *var-decl* | *var-decl*
var-decl → **id :** *type-exp*
type-exp → **int** | **bool** | **array [num] of** *type-exp*
stmts → *stmts* **;** *stmt* | *stmt*
stmt → **if** *exp* **then** *stmt* | **id :=** *exp*
exp → *exp* **+** *exp* | *exp* **or** *exp* | *exp* **[***exp***]** | **num** | **true** | **false** | **id**

Access to ST:
`lookup(name)` → type
`insert(name, type)`

Production	*Semantic rules*
var-decl → **id :** *type-exp*	insert(**id**.name, *type-exp*.type)
type-exp → **int**	*type-exp*.type := INTEGER
type-exp → **bool**	*type-exp*.type := BOOLEAN
$type\text{-}exp_1$ → **array [num] of** $type\text{-}exp_2$	$type\text{-}exp_1$.type := typeNode(ARRAY, **num**.size, $type\text{-}exp_2$.type)
$stmt_1$ → **if** *exp* **then** $stmt_2$	**if not** typeEqual(*exp*.type, BOOLEAN) **then** typeError($stmt_1$)
stmt → **id :=** *exp*	**if not** typeEqual(lookup(**id**.name), *exp*.type) **then** typeError(*stmt*);
exp_1 → exp_2 **+** exp_3	**if not** (typeEqual(exp_2.type, INTEGER) **and** typeEqual(exp_3.type, INTEGER)) **then** typeError(exp_1); exp_1.type := INTEGER
exp_1 → exp_2 **or** exp_3	**if not** (typeEqual(exp_2.type, BOOLEAN) **and** typeEqual(exp_3.type, BOOLEAN)) **then** typeError(exp_1); exp_1.type := BOOLEAN
exp_1 → exp_2 **[** exp_3 **]**	**if** arrayType(exp_2.type) **and** typeEqual(exp_3.type, INTEGER) **then** exp_1.type := childType(exp_2.type) **else** typeError(exp_1)
exp → **num**	*exp*.type := INTEGER
exp → **true**	*exp*.type := BOOLEAN
exp → **false**	*exp*.type := BOOLEAN
exp → **id**	*exp*.type := lookup(**id**.name)