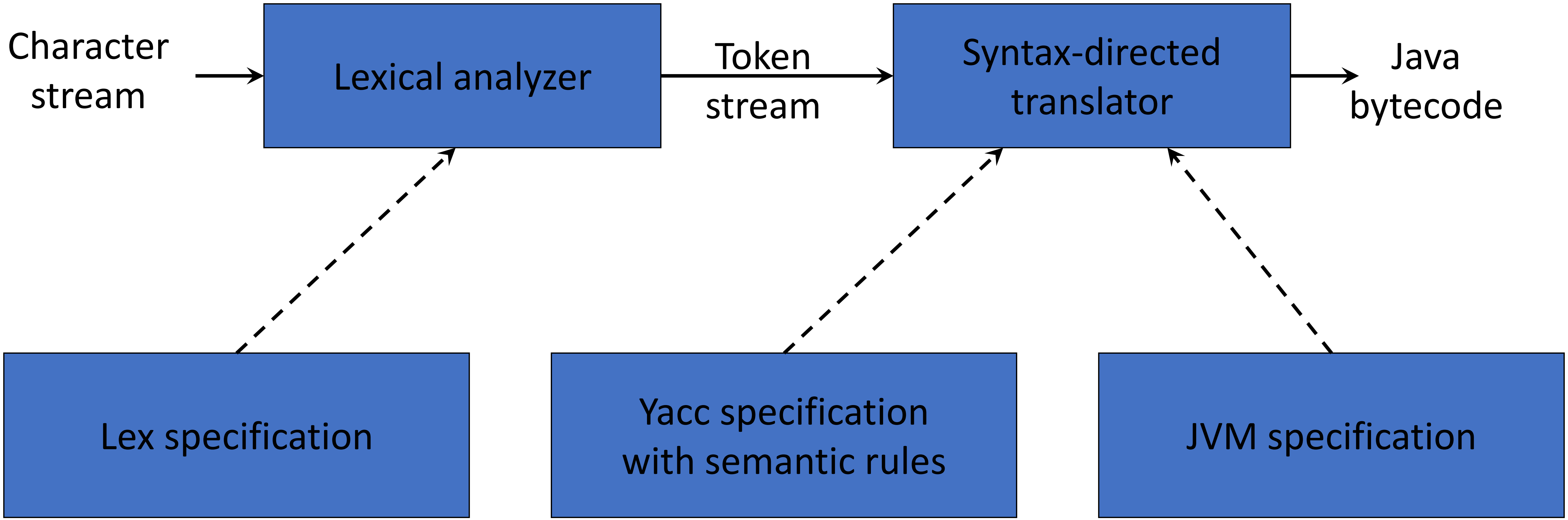
Semantic Phase of the Compiler

Functions of the semantic phase

$E^{val} \rightarrow E^{val} + T^{val}$

- Converting the parse tree to an annotated parse tree
    - Syntax directed Definition (SDD)
        - Sequence of semantic rules
    - Syntax directed Translation (SDT)
        - Uses SDD to convert one representation to another
- Writing semantic rules to check for semantic correctness
- Establishing evaluation order

The Structure of our Compiler



Syntax-Directed Definitions

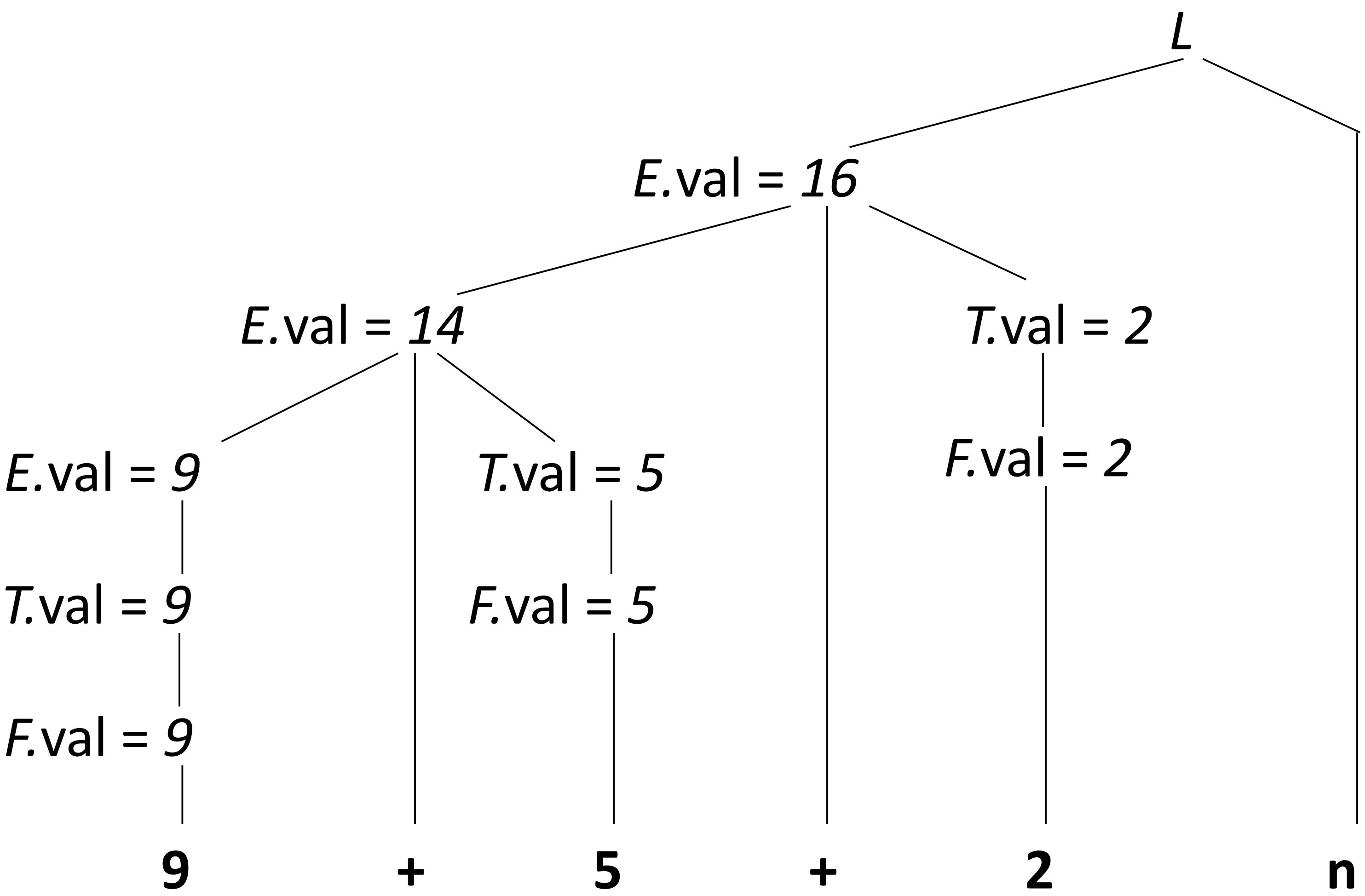
- SDD uses a CFG to specify the syntactic structure of the input
- SDD connects a set of semantic rules to productions
- Terminals and non-terminals have attributes
- A depth-first traversal algorithm is used to compute the values of the attributes in the parse tree using the semantic rules
- After the traversal is completed, the attributes contain the translated form of the input

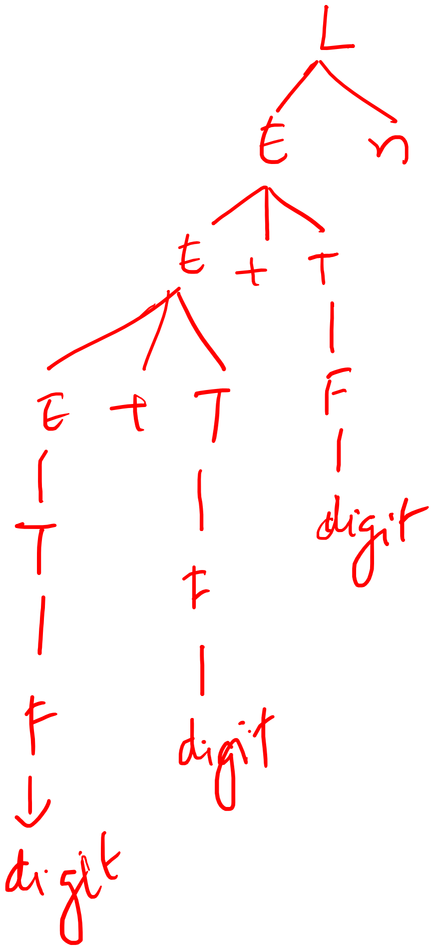
Example Attribute Grammar

Production	Semantic Rule
$L \rightarrow E\ \mathbf{n}$	$print(E.\mathrm{val})$
$E \rightarrow E_1\ \mathbf{+}\ T$	$E.\mathrm{val} := E_1.\mathrm{val} + T.\mathrm{val}$
$E \rightarrow T$	$E.\mathrm{val} := T.\mathrm{val}$
$T \rightarrow T_1\ \mathbf{*}\ F$	$T.\mathrm{val} := T_1.\mathrm{val} * F.\mathrm{val}$
$T \rightarrow F$	$T.\mathrm{val} := F.\mathrm{val}$
$F \rightarrow \mathbf{(}\ E\ \mathbf{)}$	$F.\mathrm{val} := E.\mathrm{val}$
$F \rightarrow \mathbf{digit}$	$F.\mathrm{val} := \mathbf{digit}.\mathrm{lexval}$

Annotated Parse Tree

- Parse tree showing the attribute values at each node





Annotating a Parse Tree

procedure *visit*(*n* : *node*);
begin
    for each child *m* of *n*, from left to right **do**
        visit(*m*);
    evaluate semantic rules at node *n*
end

Attributes

- Attribute values can represent
  - Numbers (literal constants)
  - Strings (literal constants)
  - Memory locations, such as a frame index of a local variable or function argument
  - A data type for type checking of expressions
  - Scoping information for local declarations
  - Intermediate program representations

Synthesized Vs Inherited Attributes

- Given a production

  $A \rightarrow \alpha$

  then each semantic rule is of the form

  $b := f(c_1, c_2, \ldots, c_k)$

  where f is a function and c_i are attributes of A and α, and either
  - b is a *synthesized* attribute of A
  - b is an *inherited* attribute of one of the grammar symbols in α

Synthesized vs Inherited Attribute

- Synthesized
  - For a node N, for a non-terminal A, attributes are defined by value of the children and the node itself – terminals have only synthesized attributes
- Inherited
  - For a node N, defined by the parent, itself and siblings

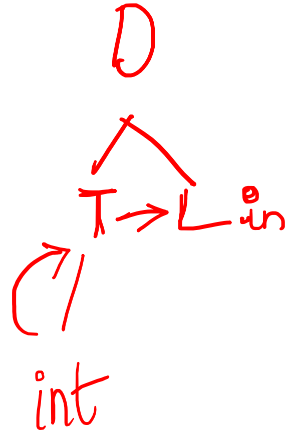
Synthesized Vs Inherited Attributes

Production	Semantic Rule
$D \rightarrow T\ L$	L.in := T.type
$T \rightarrow$ **int**	T.type := 'integer'
…	…
$L \rightarrow$ **id**	… := L.in

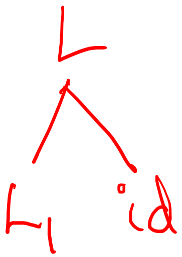
inherited

synthesized

Synthesized+Inherited Attributes



Production	Semantic Rule
$D \rightarrow T\ L$	L.in := T.type
$T \rightarrow$ **int**	T.type := 'integer'
$T \rightarrow$ **real**	T.type := 'real'
$L \rightarrow L_1$ **, id**	L_1.in := L.in; *addtype*(**id**.entry, L.in)
$L \rightarrow$ **id**	*addtype*(**id**.entry, L.in)



S-Attributed Definitions

- A syntax-directed definition that uses synthesized attributes exclusively is called an *S-attributed definition* (or *S-attributed grammar*)
- A parse tree of an S-attributed definition can be annotated with a simple bottom-up traversal
- Yacc only supports S-attributed definitions

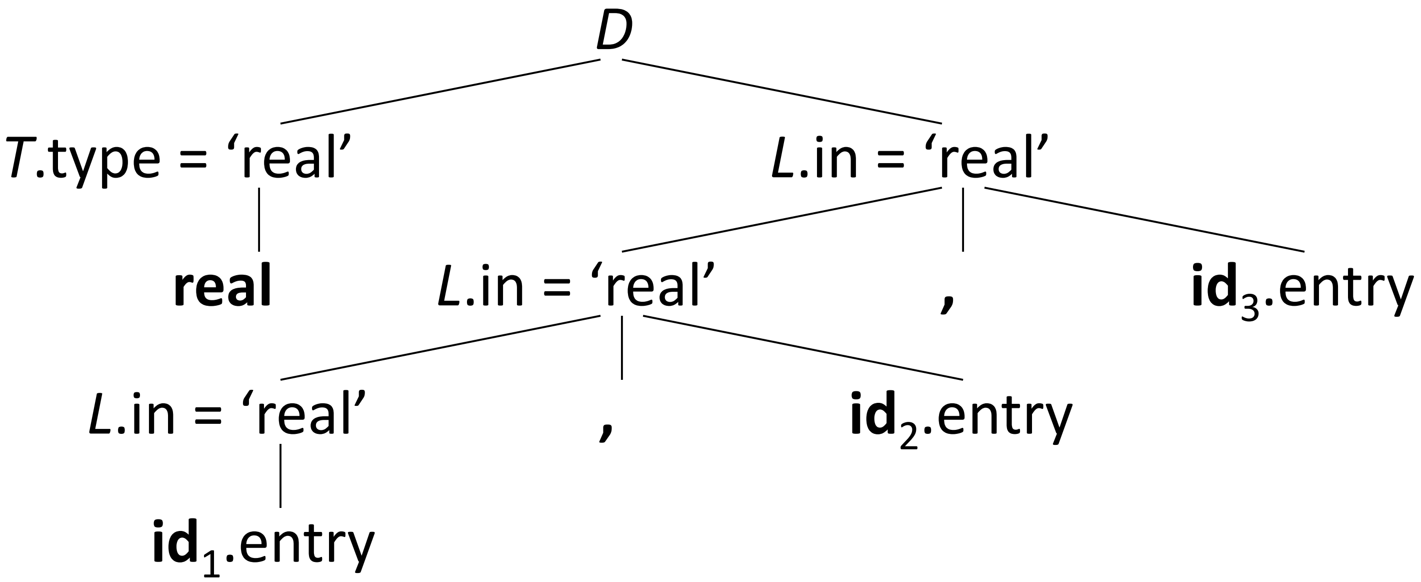
L-Attributed Definitions

- A syntax-directed definition is *L-attributed* if each inherited attribute of X_j on the right side of $A \rightarrow X_1\ X_2\ \dots\ X_n$ depends only on
  1. the attributes of the symbols $X_1, X_2, \dots, X_{j-1}$
  2. the inherited attributes of A

Annotated Parse Tree

real a, b, c

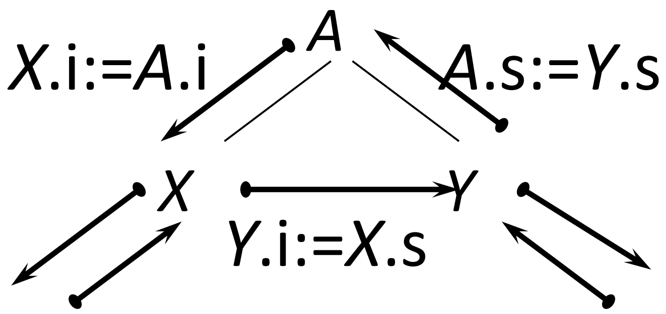
$D \rightarrow T\ L$
$T \rightarrow \textbf{int}$
$T \rightarrow \textbf{real}$
$L \rightarrow L_1\ \textbf{,}\ \textbf{id}$
$L \rightarrow \textbf{id}$



L-Attributed Definitions

- L-attributed definitions does a normal order of evaluating attributes which is depth-first and left to right

$A \rightarrow X\ Y$

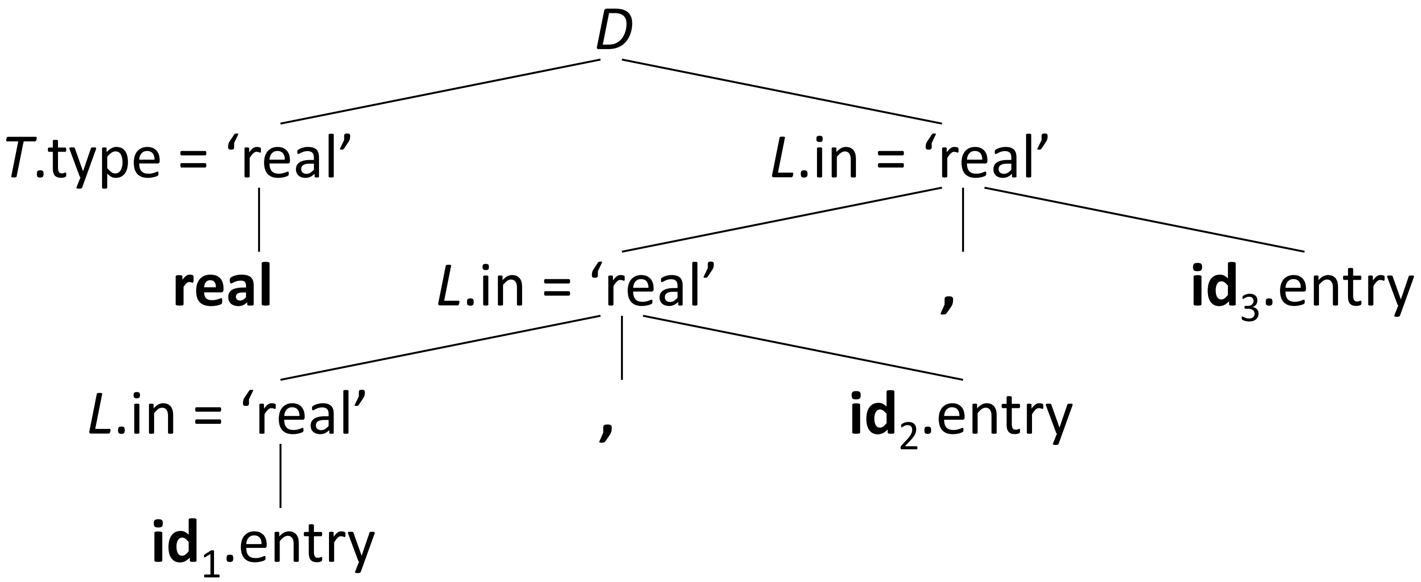


X.i := A.i
Y.i := X.s
A.s := Y.s

- Every S-attributed syntax-directed definition is also L-attributed

Annotated Parse Tree

$D \rightarrow T\ L$
$T \rightarrow$ **int**
$T \rightarrow$ **real**
$L \rightarrow L_1$ **, id**
$L \rightarrow$ **id**



Grammar with Synthesized + Inherited Attributes

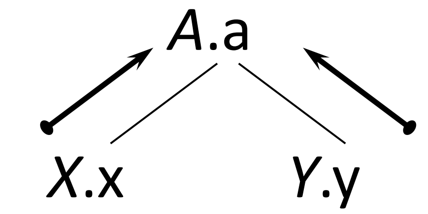
Production	Semantic Rule
$D \rightarrow T\ L$	L.in := T.type
$T \rightarrow$ **int**	T.type := 'integer'
$T \rightarrow$ **real**	T.type := 'real'
$L \rightarrow L_1$ **, id**	L_1.in := L.in; *addtype*(**id**.entry, L.in)
$L \rightarrow$ **id**	*addtype*(**id**.entry, L.in)

Synthesized: T.type, **id**.entry

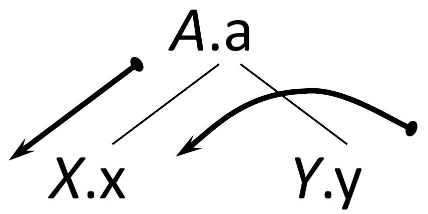
Inherited: L.in

Acyclic dependence Graph

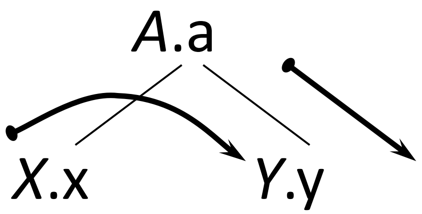
$A \rightarrow X\ Y$



A.a := f(X.x, Y.y)



X.x := f(A.a, Y.y)



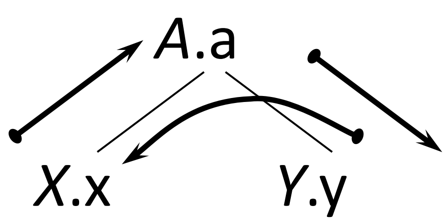
Y.y := f(A.a, X.x)

Direction of

value dependence

Dependency Graphs with Cycles?

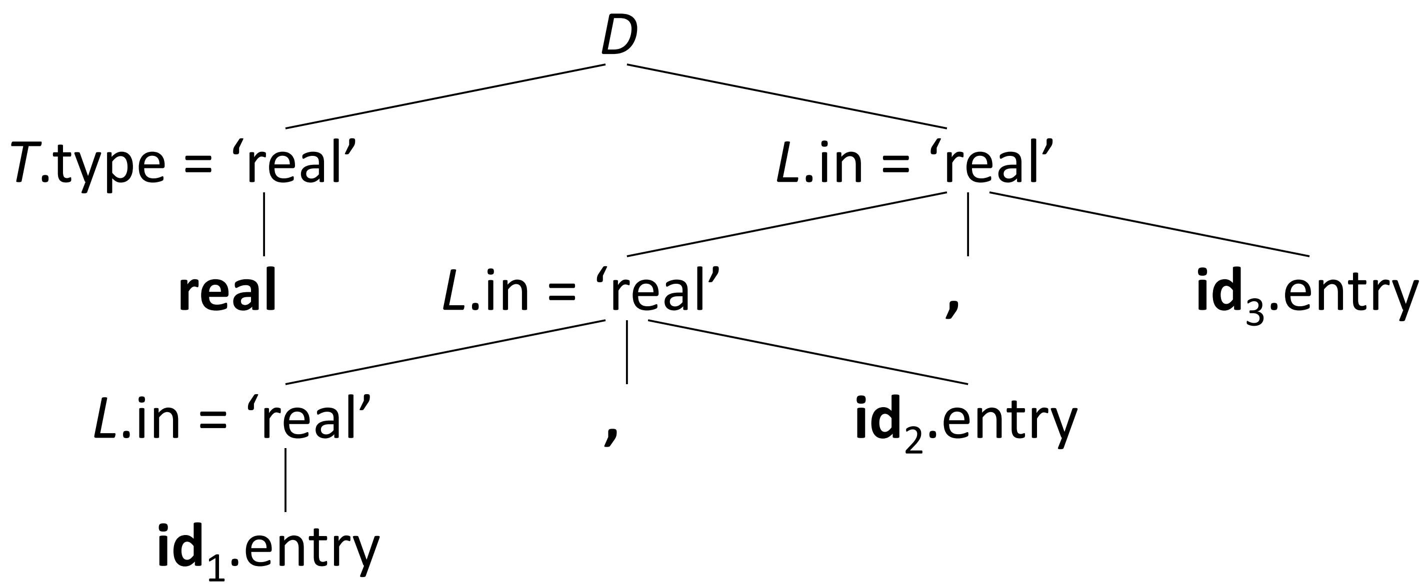
- Edges in the dependence graph show the evaluation order for attribute values
- Dependency graphs cannot be cyclic



A.a := f(X.x)
X.x := f(Y.y)
Y.y := f(A.a)

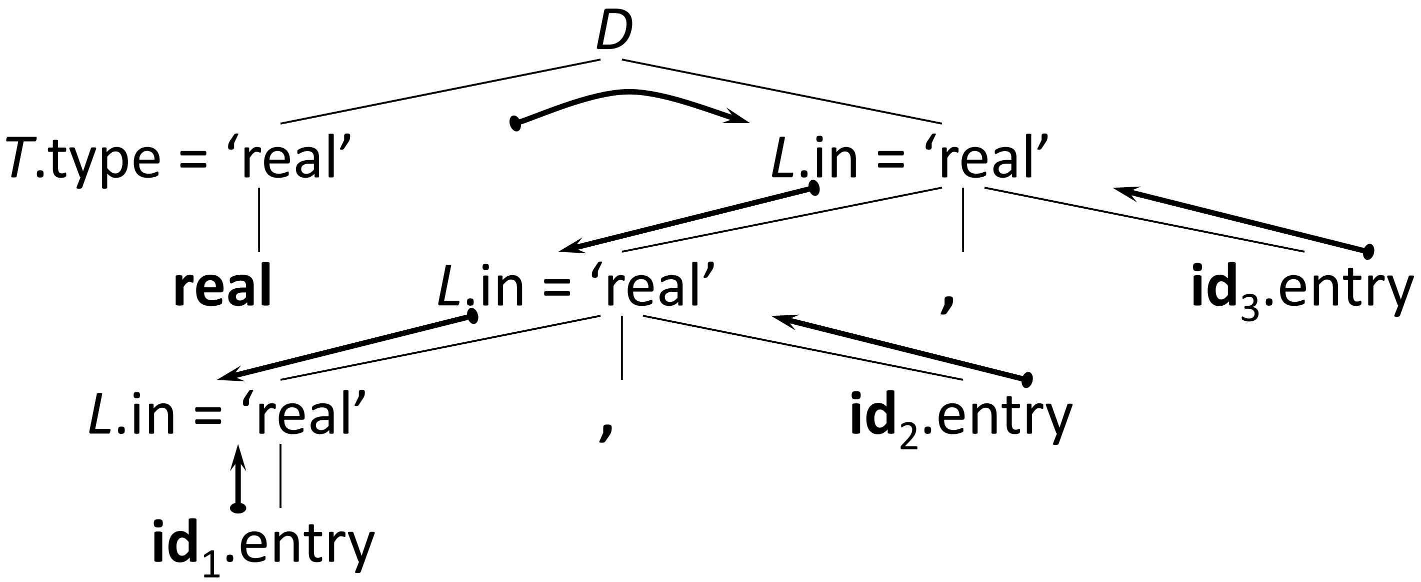
Error: cyclic dependence

Parse tree



D
T.type = 'real'
L.in = 'real'
real
L.in = 'real'
,
id3.entry
L.in = 'real'
,
id2.entry
id1.entry

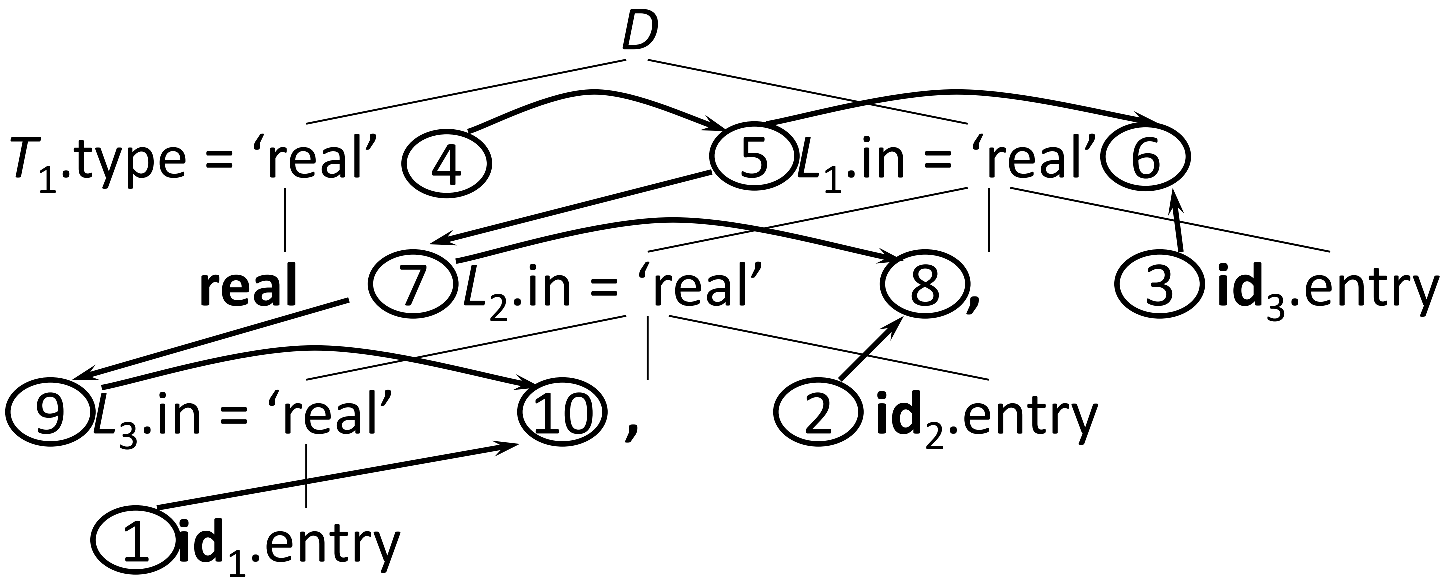
Parse tree and dependency graph



Evaluation Order

- A *topological sort* of a directed acyclic graph (DAG) is any ordering m_1, m_2, …, m_n of the nodes of the graph, such that if $m_i \to m_j$ is an edge then m_i appears before m_j
- Any topological sort of a dependency graph gives a valid evaluation order for the semantic rules

Example Topological sorting



1. Get **id**$_1$.entry
2. Get **id**$_2$.entry
3. Get **id**$_3$.entry
4. T_1.type='real'
5. L_1.in=T_1.type
6. *addtype*(**id**$_3$.entry, L_1.in)
7. L_2.in=L_1.in
8. *addtype*(**id**$_2$.entry, L_2.in)
9. L_3.in=L_2.in
10. *addtype*(**id**$_1$.entry, L_3.in)

Semantic Rules

Production	Semantic Rule
$D \rightarrow T\ L$	L.in := T.type
$T \rightarrow$ **int**	T.type := 'integer'
$T \rightarrow$ **real**	T.type := 'real'
$L \rightarrow L_1$ **, id**	L_1.in := L.in; *addtype*(**id**.entry, L.in)
$L \rightarrow$ **id**	*addtype*(**id**.entry, L.in)

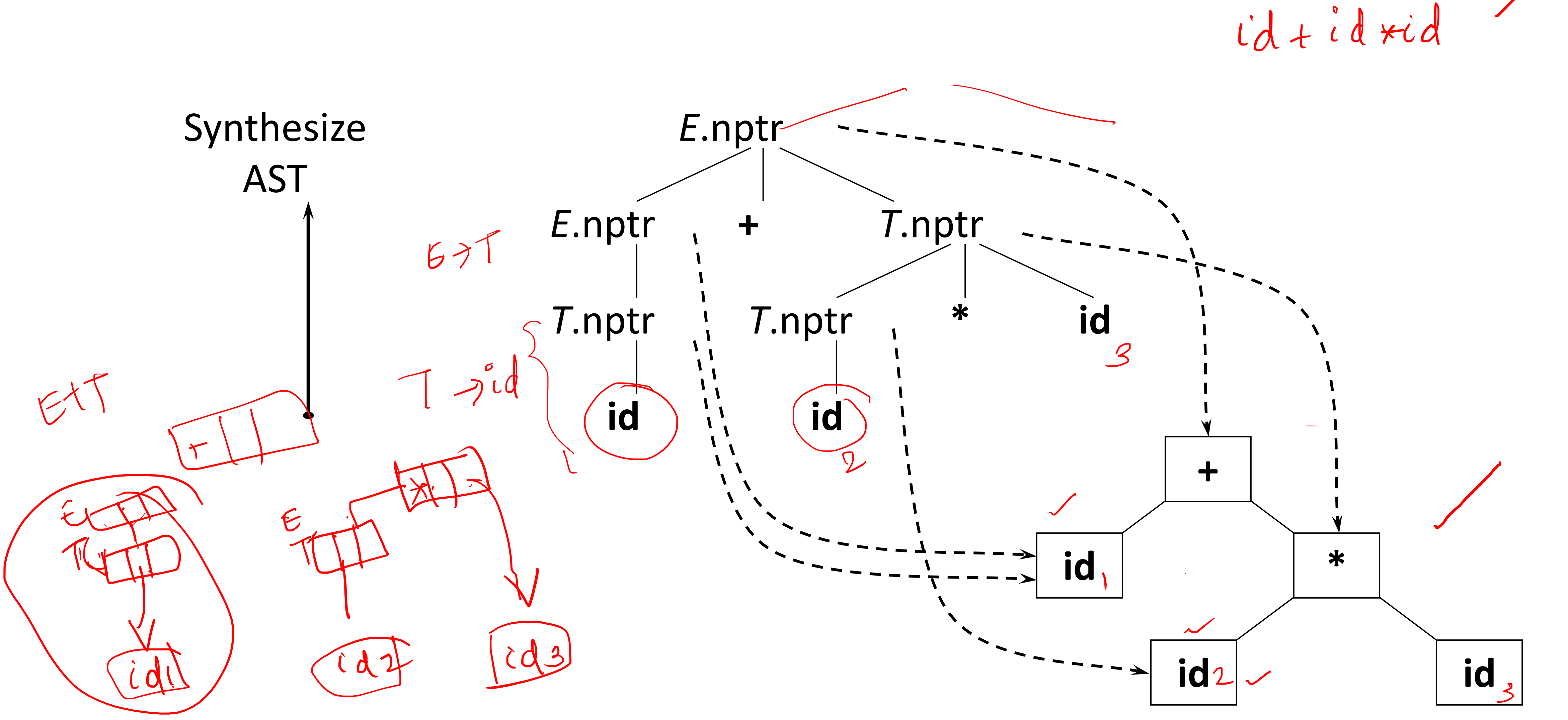
Translation Scheme

$D \to T$ { L.in := T.type } L
$T \to$ **int** { T.type := ‘integer’ }
$T \to$ **real** { T.type := ‘real’ }
$L \to$ { L_1.in := L.in } L_1 **, id** { *addtype*(**id**.entry, L.in) }
$L \to$ **id** { *addtype*(**id**.entry, L.in) }

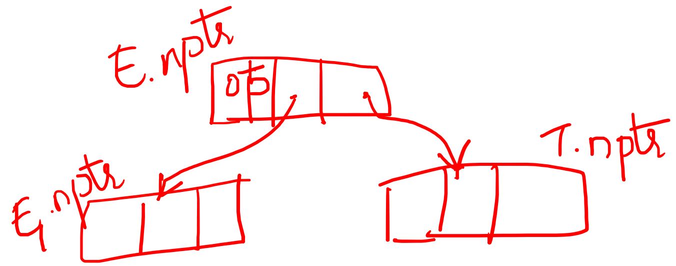
Syntax Directed Translation

- A parse tree is called a *concrete syntax tree*
- An *abstract syntax tree* (AST) is defined by the compiler writer as a more convenient intermediate representation

Generating Abstract Syntax Trees



Generating syntax tree



Production	Semantic Rule
$E \rightarrow E_1$ **+** T	E.nptr := mknode('+', E_1.nptr, T.nptr)
$E \rightarrow E_1$ **-** T	E.nptr := mknode('-', E_1.nptr, T.nptr)
$E \rightarrow T$	E.nptr := T.nptr
$T \rightarrow T_1$ *** id**	T.nptr := mknode('*', T_1.nptr, mkleaf(**id**, **id**.entry))
$T \rightarrow T_1$ **/ id**	T.nptr := mknode('/', T_1.nptr, mkleaf(**id**, **id**.entry))
$T \rightarrow$ **id**	T.nptr := mkleaf(**id**, **id**.entry)

$T \rightarrow T_1 * F$ $\quad$ T.nptr := mknode('*', T_1.nptr, F.nptr)

$F \rightarrow id$ $\quad$ F.nptr = mkleaf(id, id.entry)

$F \rightarrow (E)$ $\quad$ F.nptr = E.nptr

Type Checking

- Need to verify that the source program follows the syntactic and semantic conventions – Static checking
- Helps in reporting programming errors

Static Checking

- Type Checking – operator applied to an incompatible operand
  - Example: error if array variable is added with function variable
- Flow of control check – statements that results in a branch need to be terminated correctly
  - Example: Break statements

Static Checking

- Uniqueness check – object must be defined exactly once for some scenarios
    - Example: labels in case statements need to be unique in pascal, identifiers need to be unique
- Name-related checks – Same name appears more than once
    - Example: Ada where a name appears more than once and compiler to verify this

Type Checking

- `int op(int), op(float);`
- `int f(float);`
- `int a, c[10], d;`
- `d = c+d;      // FAIL`
- `*d = a;       // FAIL`
- `a = op(d);          // OK: overloading (C++)`
- `a = f(d);           // OK: coersion`
- `vector<int> v;//OK: template instantiation`

Flow of control check

```
myfunc()
{ …
  while (n)
  { …
    if (i>10)
      break; // OK
  }
}
```

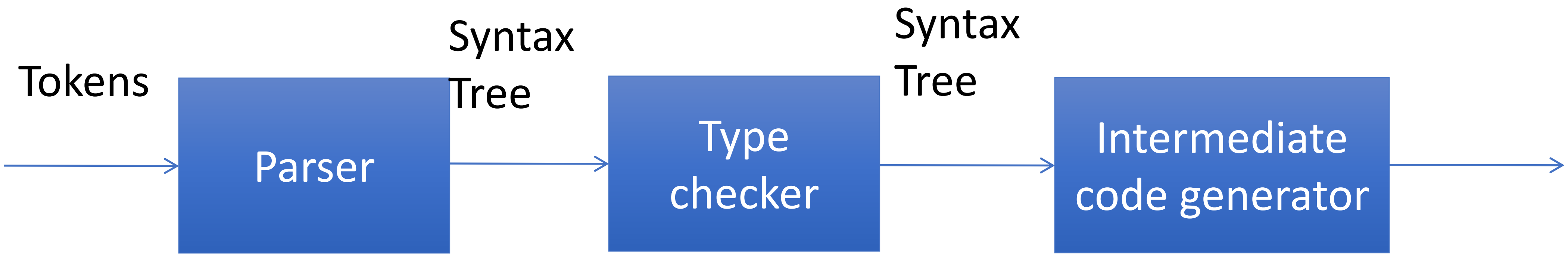
```
myfunc()
{ …
  break; // ERROR
}
```

Uniqueness check

```
myfunc()
{ int i, j, i; // ERROR
  …
}
```

```
cnufym(int a, int a) // ERROR
{    …
}
```

Position of Type checker



Type Checking

- This is carried out in semantic phase
- Type checking information added with the semantic rules
- Basic type checking is performed
- This is extended to type checking of complex attributes

Type Checking of Expressions

- E → literal {E.type = char}
- E → num {E.type = integer}
- E → id {E.type = lookup(id.entry}
- E → E1 mod E2 {E.type =

  {if E1.type == integer and E2.type == integer then
  integer

  else

  type_error}

Type Checking of Expressions

- E → E1[E2] {E.type =

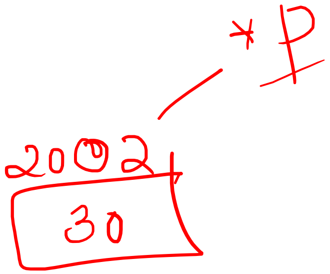
  if E2.type == integer and E1.type = array(s,t) then t

  else type_error}

- E → E1↑ {E.type =

  if E1.type = ptr(t) then t

  else type_error}



Type Checking of statements

- P → D; S
- S → id := E {S.type =

  if id.type = E.type then void

  else type_error}
- S → if E then S1 {S.type =

  if E.type = boolean then S1.type

  else type_error}

Type Checking of statements

- S → while E do S1 { S.type =

  if E.type = boolean then S1.type

  else type_error}

- S → S1 ; S2   {S.type =

  if S1.type = void and S2.type = void, then void
  else type_error }

Type checking of functions

- $E \rightarrow E1\ (E2)$ {E.type =

  if E2.type = s and E1.type = $s \rightarrow t$ then t

  else type_error}

- $T \rightarrow T1 \rightarrow T2$ { T.type = T1.type $\rightarrow$ T2.type}

Type checking rules for coercions

- Coercion – Implicit type conversions done by the compiler
- Explicit type conversion – done by the programmer

int c;

(float)c ✓

Type checking for coercions

• E → num	E.type = integer
• E → num.num	E.type = real
• E → id	E.type = lookup (id.entry)
• E → E1 op E2	E.type = if E1.type = integer and E2.type = integer
	then integer
	else if E1.type = integer and E2.type = real
	then real
	else if E1.type = real and E2.type = integer
	then real
	else if E1.type = real and E2.type = real
	then real
	else type_error

Summary

- Attributes type has been discussed
- Construction of Dependency graph
- Topological sorting for evaluation order
- Type checking