Syntax Directed Translation

Lecturer: Nguyen Van Vinh – UET, VNU Ha noi

Slides based on: Nguyen Phuong Thai, UET, VNU Hanoi

What is syntax-directed translation?

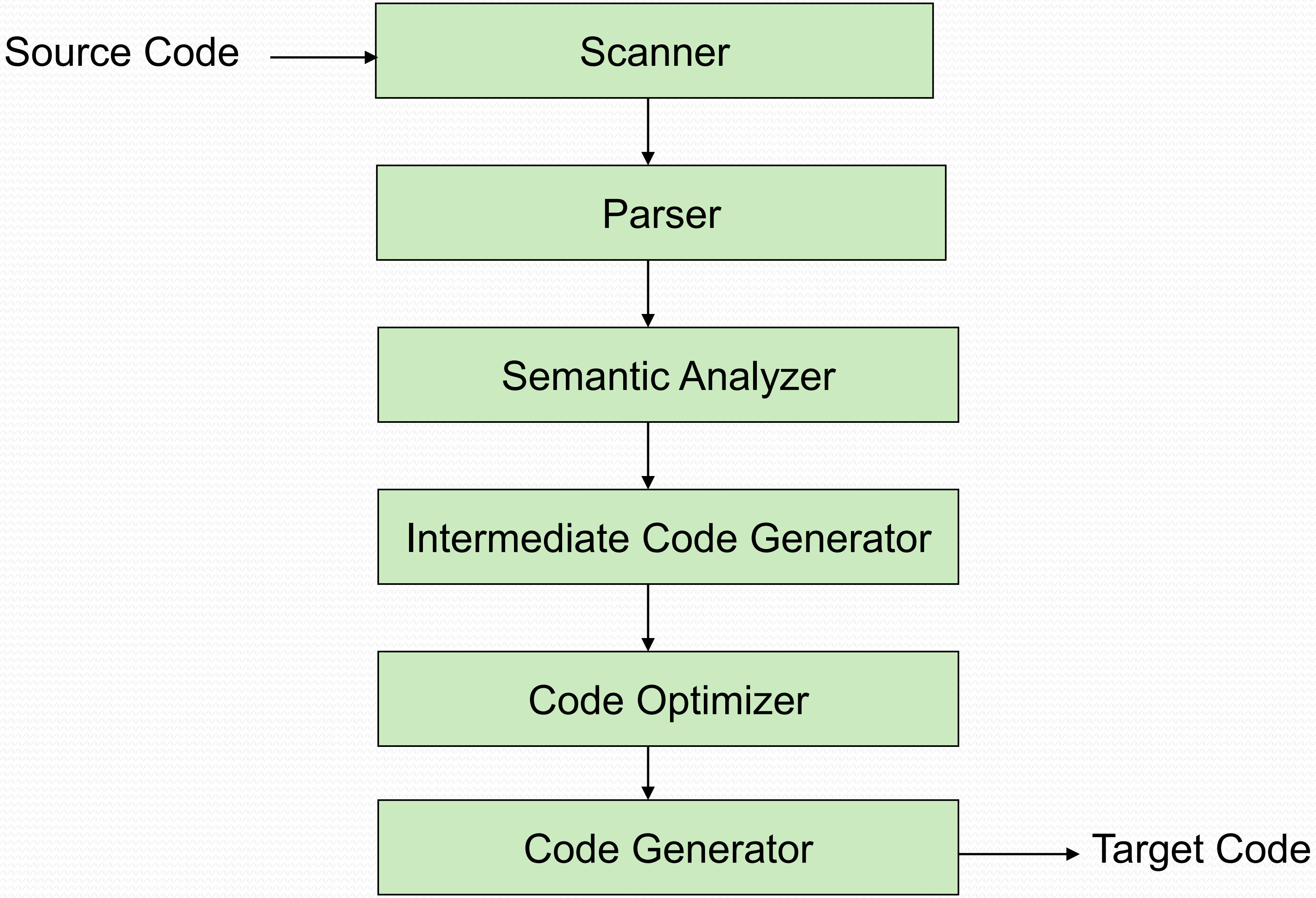
- The compilation process is driven by the syntax.
- The semantic routines perform interpretation based on the syntax structure
- Attaching **attributes** to the grammar symbols
- Values for attributes are computed by **semantic actions** associated with the grammar productions

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	$E.code = E_1.code \parallel T.code \parallel '+'$

Outline

- Attribute grammars
- Attribute types
- Dependency graphs
- Translation schemas
- Syntax directed translation in LL parsing

The Typical Structure of a Compiler



Attribute grammars: an informal definition

- Attribute grammars:
  - Generalisation of CFGs
  - Each attribute associated with a grammar symbol
  - Each semantic rule associated with a production defining attributes
  - High-level spec, independent of any evaluation order
- Dependences between attributes
  - Attributes computed from other attributes
  - Synthesised attributes: computed from children
  - Inherited attributes: computed from parent and siblings

Format for writing syntax-directed definitions (SDD)

	PRODUCTION	SEMANTIC RULES
1)	$L \rightarrow E$ **n**	$L.val = E.val$
2)	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \rightarrow T$	$E.val = T.val$
4)	$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \rightarrow F$	$T.val = F.val$
6)	$F \rightarrow (E)$	$F.val = E.val$
7)	$F \rightarrow$ **digit**	$F.val =$ **digit**.lexval

SDD for a desk calculator

- E.val is one of the attributes of E.
- **digit**.lexval is the attribute (integer value) returned by the lexical analyzer
- To avoid confusion, recursively defined nonterminals are numbered on the RHS.
- Semantic actions are performed when this production is "used".

Formal definitions of **synthesised and inherited attributes**

- Let $X_0 \rightarrow X_1\ X_2\ \ldots\ X_n$ be a production, and
- $A(X)$ be the set of attributes associated with a grammar symbol X
- Then **a synthesised attribute**, ***syn***, of X_0 is computed by:

  $X_0.syn = f(A(X_1), A(X_2), \ldots, A(X_n))$

  syn on a tree node depends on those on its children
- An **inherited attribute**, ***inh***, of X_i, where $1 <= i <= n$, is computed by:

  $X_i.inh = g(A(X_0), A(X_1), \ldots, A(X_i))$

  - inh on a tree node depends on those on its parent and/or siblings
  - Xi.inh can depend on the other attributes in A(Xi)

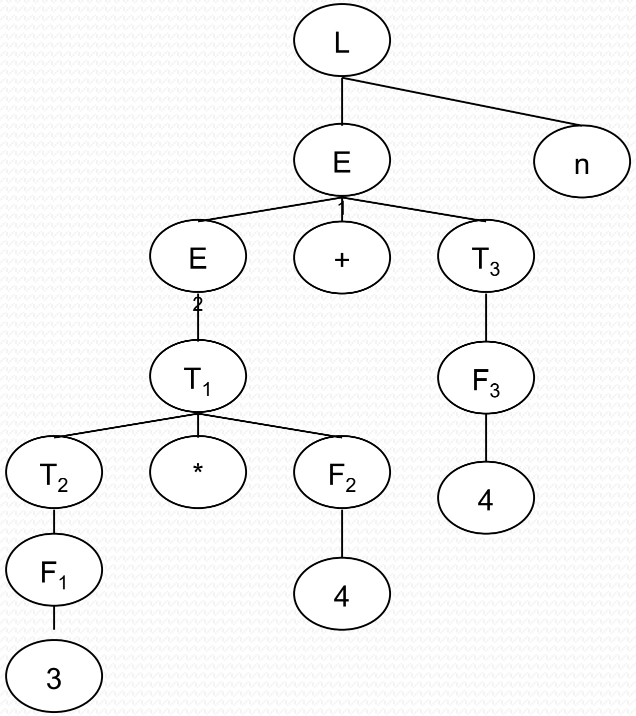
Attributes associated with a grammar symbol

- A attribute can represent anything we choose:
  - a string
  - a number
  - a type
  - a memory location
  - a piece of code
  - etc.
- Each attribute has a name and a type

An attribute grammar with synthesized attributes

Production	**Semantic rules**
L -> E **n**	*Print(E.val)*
E -> E_1 + T	$E.val = E_1.val + T.val$
E -> T	$E.val = T.val$
T -> T_1 * F	$T.val = T_1.val * F.val$
T -> F	$T.val = F.val$
F -> (E)	$F.val = E.val$
F -> **digit**	$F.val = digit.lexval$

Synthesized attributes (cont)



Input “3*4+4”

F_1.val=3 (syntax: F_1->3 semantic: F_1.val=3.lexical)

F_2.val=4 (syntax: F_2->3 semantic: F_2.val=4.lexical)

T_2.val=3 (syntax: T_2->F_1 semantic: T_2.val=F_1.val)

T_1.val=3*4=12 (syntax: T_1->T_2*F_2 semantic: T_1.val=T_2.val*F_2.val)

F_3.val=4 (syntax: F_3->4 semantic: F_3.val=4.lexical)

T_3.val=4 (syntax: T_3->F_3 semantic: T_3.val=F_3.val)

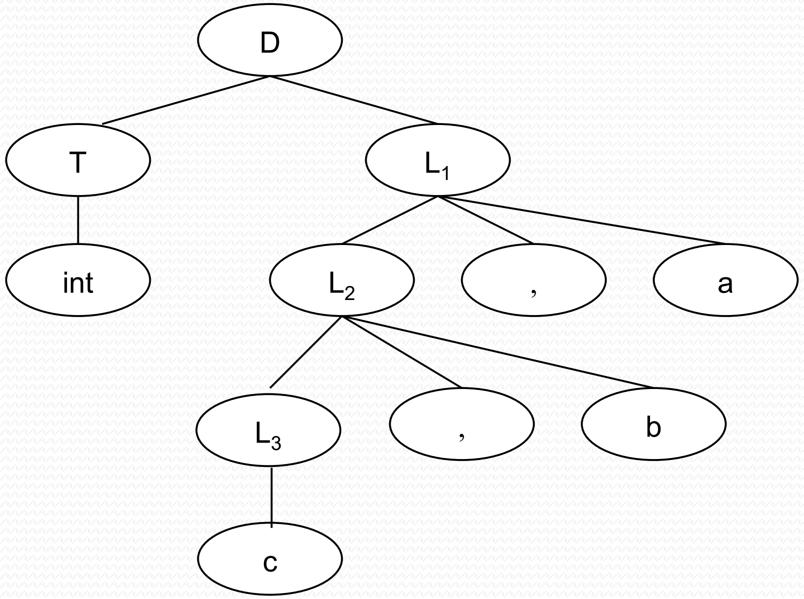
E_1.val=12+4=16 (syntax: E_1->E_2+T_3 semantic: E_1.val=E_2.val+T_3.val)

“16” (syntax: L->E_1 n semantic: print(E_1.val))

An attribute grammar with inherited attributes

Production	**Semantic rules**
D -> T L	L.in := T.type
T -> int	T.type := interger
T -> real	T.type := real
L -> L_1, id	L_1.in := L.in ; addtype(id.entry, L.in)
L -> id	addtype(id.entry,L.in)

Inherited attributes (cont)



Input “**int c, b, a**”

- Traverse the parse tree and evaluate semantic rules (depth first)
- Results:

T.type = interger (syntax:T->int semantic: T.type=interger)

L_1.in = interger (syntax: D -> T L_1 semantic: L_1.in=T.type)

L_2.in = interger (syntax: L_1 -> L_2 , a semantic: L_2.in = L_1.in)

L_3.in = interger (syntax: L_2 -> L_3 , b semantic: L_3.in = L_2.in)

c.entry = interger (syntax: L_3 -> c semantic: addtype(c.entry,L_3.in))

b.entry = interger (syntax: L_2 -> L_3 , b semantic: addtype(b.entry,L_2.in))

a.entry = interger (syntax: L_1 -> L_2 , a semantic: addtype(a.entry,L_1.in))

Semantic-action evaluation order

- **Semantic-action evaluation order is not arbitrary**
- **This order is constrained by**
  - Semantic rules
  - Program's syntactic structure
- **Order constraints:**
  - If attribute b depends on attribute c, then
    - Semantic action for b must be evaluated after semantic action for c.
  - In other words, for every semantic actions $b{:=}f(c_1,c_2, \ldots ,c_k)$
    - Dependent attributes' values $c_1,c_2, \ldots ,c_k$ must be evaluated before that of f()

Dependency graphs

- Evaluation order can be represented by a directed graph called a dependency graph
- If circles exist, semantic-action order can not be determined
  - Only DAGs (directed acyclic graph) are considered
- Topo arrangement
  - A semantic-action evaluation order determined by dependency graph
  - Satisfying order constraints

Dependency graphs (cont)

- Vertex construction

  for *each node n of the parse tree* **do**

      for *each attribute a of grammar symbol at n* **do**

          construct a node of dependency graph for a;

- Edge construction

  for *each node n of the parse tree* **do**

      for *each semantic action* $b{:=}f(c_1,c_2,\ldots,c_k)$

          associated with production applied at n **do**

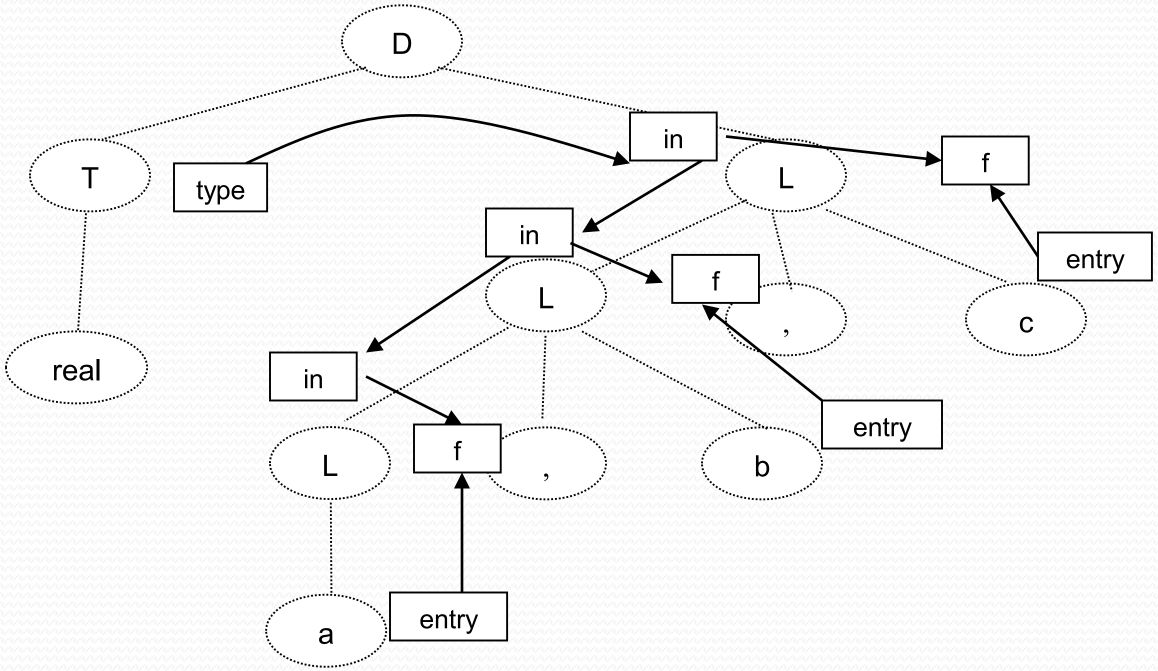
          for $i{:=}1$ *to k* **do**

                  construct an edge from node c_i *to node b*

Dependency graphs (cont)

- Some important notes
  - Dependency graph must be constructed based on syntax tree
  - In a dependency graph, each node represents an attribute of a grammar symbol
  - Not only attributes of the same type depend on each other

An example of dependency graph construction



Semantic-action evaluation order

- Suppose that nodes are ordered m_1, m_2, . . .,m_k
- If m_i ->m_j exists then
  - m_i preceeds m_j in that order
  - or $i<j$
- This is a correct evaluation order

An example of evaluation order

Node 4: a_4 := real

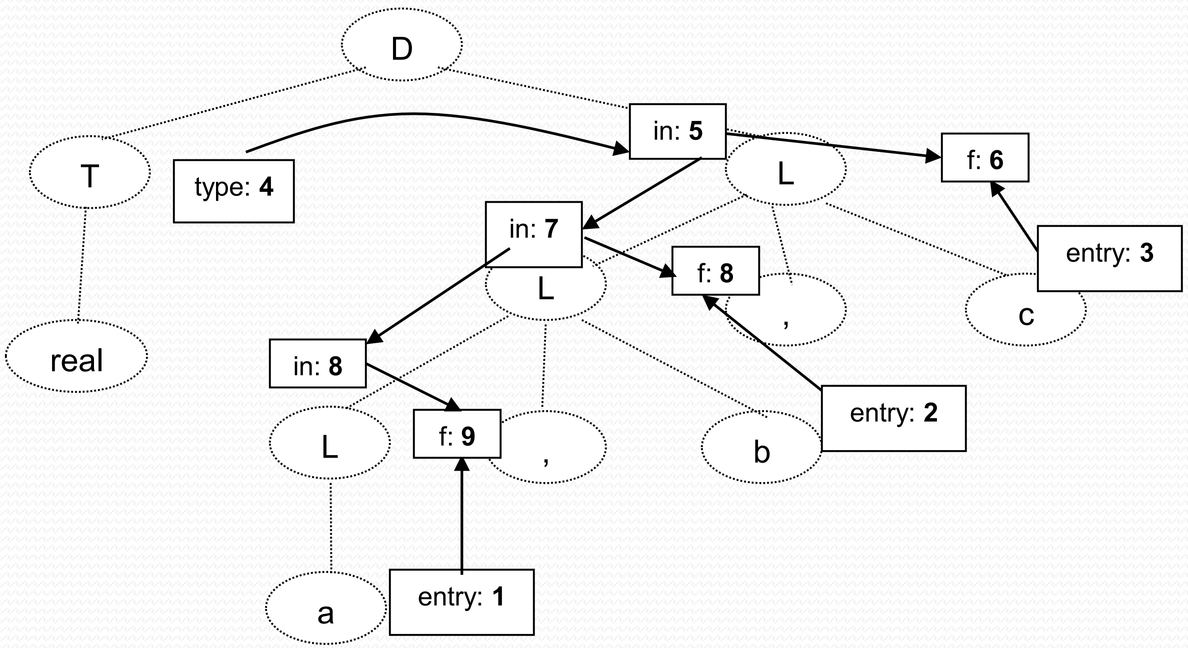
Node 5: a_5 := a_4 := real

Node 6: addtype(c.entry,a_5) = addtype(c.entry,real)

Node 7: a_7 := a_5 := real

Node 8: addtype(b.entry,a_7) = addtype(b.entry,real)

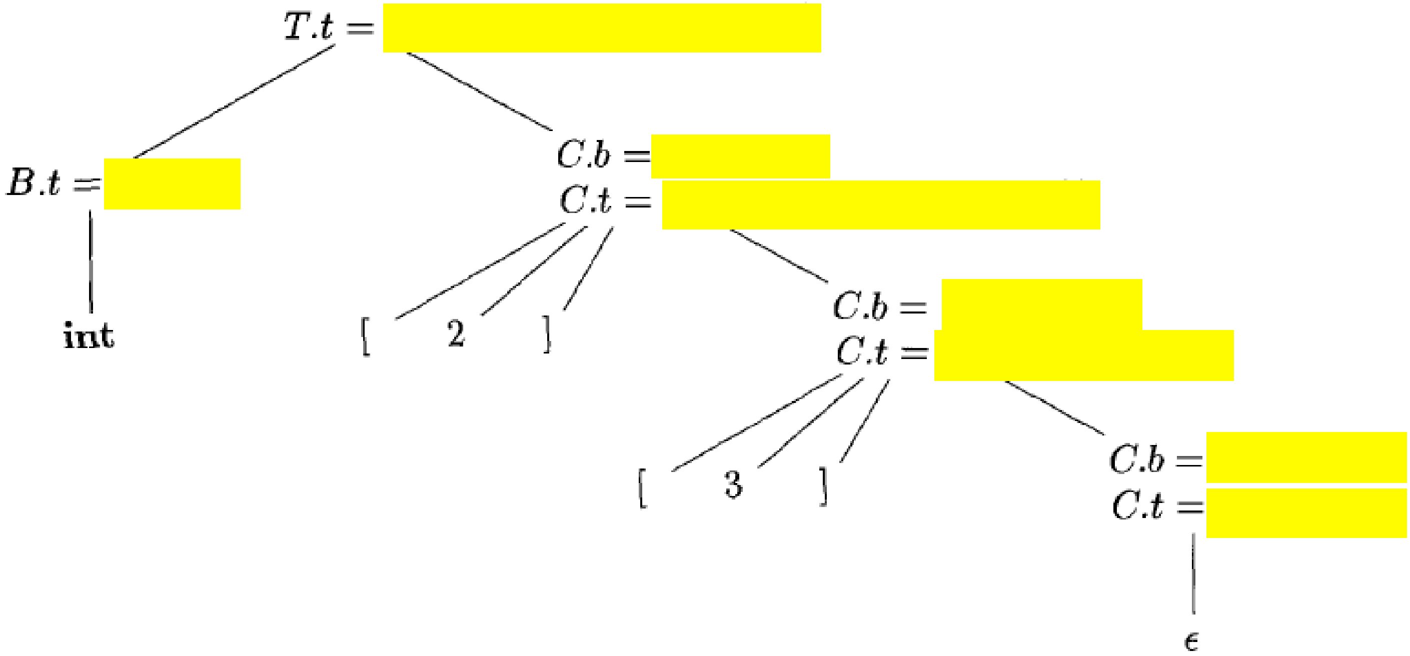
Node 9: addtype(a.entry,a_8) = addtype(a.entry,real)



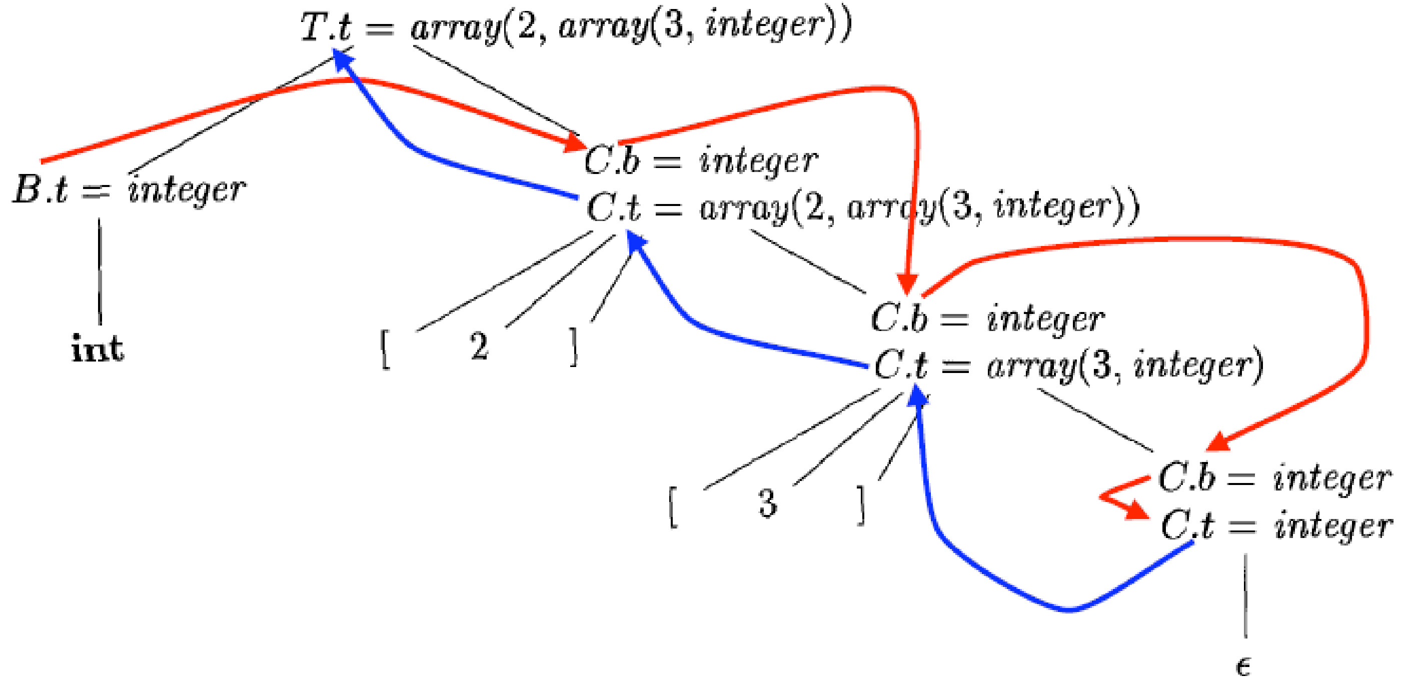
Input: “real a, b, c”

PRODUCTION	SEMANTIC RULES
$T \rightarrow B\ C$	$T.t = C.t$
	$C.b = B.t$
$B \rightarrow$ **int**	$B.t = integer$
$B \rightarrow$ **float**	$B.t = float$
$C \rightarrow [\ \textbf{num}\ ]\ C_1$	$C.t = array(\textbf{num}.val,\ C_1.t)$
	$C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

`int[2][3] ≡ array(2,array(3,integer))`



PRODUCTION	SEMANTIC RULES
$T \rightarrow B\ C$	$T.t = C.t$ $C.b = B.t$
$B \rightarrow$ **int**	$B.t = integer$
$B \rightarrow$ **float**	$B.t = float$
$C \rightarrow [\ \textbf{num}\ ]\ C_1$	$C.t = array(\textbf{num}.val,\ C_1.t)$ $C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$



Attribute evaluators

- **Tree Walkers:** Traverse the parse or syntax tree in one pass or multiple passes at compile time
  - Capable of evaluating any noncircular attribute grammar
  - An attribute grammar is circular if an attribute depends on itself
  - Can decide the circularity in exponential time
  - Too complex to be used in practice
- **Rule-Based Methods:** The compiler writer analyses the attribute grammar and fixes an evaluation order at compiler-construction time
  - Trees can still be used for attribute evaluation
  - Almost all reasonable grammars can be handled this way
  - Used practically in all compilers

L-attributed grammars

- **Motivation:** parsing and semantic analysis in one pass in top-down parsers (recursive descent and LL parsers)
- **Definition:** An attribute grammar is L-attributed if each inherited attribute of Xi, 1 <= i <= n, on the right-hand side of $X_0 \to X_1\ X_2\ \ldots\ X_m$, depends only on:
  - the attributes of the symbols $X_1, X_2, \ldots, X_{i-1}$ to the left of X_i in the production, and
  - the inherited attributes of X_0
  - The L: the information flowing from left to right

L-attributed grammars (cont)

```
void dfvisit (AST N) {
    for ( each child M of N from left to right ) {
        evaluate inherited attributes of M;
        dfvisit(m);
    }
    evaluate synthesised attributes of N
}
```

- All attributes can be evaluated in one pass
- Parsing and semantic analysis can be done together (say, in a recursive descent parser) without using a tree

S-attributed grammars

- **Motivation:** parsing and semantic analysis in one pass in bottom-up parsers
- **Definition:** An attribute grammar is S-attributed if it uses synthesised attributes only
- The information always flow up in the tree
- Every S-attributed grammar is L-attributed

Syntax-directed translation scheme (SDT)

- Context-free grammar
- Can implement Syntax Directed Definition (SDD)
- Program fragments embedded within production bodies → **semantic actions**
  - called semantic rules
  - can appear anywhere within the production body

Syntax-directed translation in LL parsing

- One-pass design
- Method:
  - Each non terminal symbol A is associated with an evaluation function, void ParseA(Symbol A);
  - Each terminal symbol is associated with an input string matching function

Syntax-directed translation in LL parsing (cont)

- Suppose that A is the LHS of
  A-> α_1 | α_2 | . . . | α_n
- Then ParseA function will be:

```
void ParseA(Symbol A, Rule r, ...)
{
    if(r==A->α1)
        call semantic processing function of A->α1
    else if(r==A->α2)
        call semantic processing function of A->α2
    . . .
    else if(r==A->αn)
        call semantic processing function of A->αn
}
```

Syntax-directed translation in LL parsing (cont)

- Loop up the LL table to get the rule that A will be expanded
- When expanding A using the right hand side:
  - The current element is a terminal
  - Current element is a non-terminal: call the corresponding function
    - parameters: left siblings and A's inherited attributes
  - Current element is a semantic action: evaluate it

Example

E -> T $\{R.i := T.val\}$

R $\{E.val := R.s\}$

R -> +

T $\{R_1.i := R.i + T.val\}$

R_1 $\{R.s := R_1.s\}$

R -> ε $\{R.s := R.i\}$

T -> (E) $\{T.val := E.val\}$

T -> num $\{T.val := num.val\}$

Example (cont)

```
void ParseE(...)
{
        // E ->      T {R.i:=T.val}
        //           R {E.val:=R.s}
        ParseT(...); R.i := T.val
        ParseR(...); E.val := R.s
}
```

```
void ParseR(...)
{               // case 1
                //R ->      +
                //          T {R1.i:=R.i+T.val}
                //          R1 {R.s:=T.val+R1.i}
                if(rule=R->+TR1)
                {           match('+');
                            ParseT(...);
                            R1.i:=R.i+T.val;
                            ParseR(...);
R.s:=R1.s

                }
                // R ->ε {R.s:=R.i}
                else if(rule=R->ε)
                {
                            R.s:=R.i
                }
 }
```

Example (cont)

First(E)=First(T) = {(,num}

First(R) = {ε,+}

Follow(R) = {$,)}

	num	+	(	)	$
E	E->TR		E->TR		
T	T->num		T->(E)		
R		R->+TR		R->ε	R->ε

Example (cont)

Stack	Input	Production	Semantic rule
$E	6+4$	E->TR	
${E.val:=R.s}R{R.i:=T.val} T	6+4$	T->6	
${E.val:=R.s}R{R.i:=T.val} {T.val:=num.val}6	6+4$		
${E.val:=R.s}R{R.i:=T.val} {T.val:=num.val}	+4$		T.val=6
${E.val:=R.s}R{R.i:=T.val}	+4$		R.i=T.val=6
${E.val:=R.s}R	+4$	R->+TR$_1$	
${E.val:=R.s}{R.s:=$R_1$.s}R$_1${$R_1$.i:=R.i+T.val}T+	+4$		
${E.val:=R.s}{R.s:=$R_1$.s}R$_1${$R_1$.i:=R.i+T.val}T	4$	T->4	
${E.val:=R.s}{R.s:=$R_1$.s}R$_1${$R_1$.i:=R.i+T.val} {T.val:=num.val}4	4$		
${E.val:=R.s}{R.s:=$R_1$.s}R$_1${$R_1$.i:=R.i+T.val} {T.val:=num.val}	$		T.val=4
${E.val:=R.s}{R.s:=$R_1$.s}R$_1${$R_1$.i:=R.i+T.val}	$		R_1.i:=R.i+T.val=10
${E.val:=R.s}{R.s:=$R_1$.s}R$_1$	$	R -> ε	
${E.val:=R.s}{R.s:=$R_1$.s}{R.s:=R.i}	$		R.s:=R.i=10
${E.val:=R.s}{R.s:=$R_1$.s}	$		R.s:=R_1.s=10
$ {E.val:=R.s}	$		E.val=R.s=10
$	$		

	Production	Semantic Rules
1)	$D \rightarrow T\ L$	$L.inh = T.type$
2)	$T \rightarrow \textbf{int}$	$T.type = \text{integer}$
3)	$T \rightarrow \textbf{float}$	$T.type = \text{float}$
4)	$L \rightarrow L_1\ ,\ \textbf{id}$	$L_1.inh = L.inh$ $addType(\textbf{id}.entry, L.inh)$
5)	$L \rightarrow \textbf{id}$	$addType(\textbf{id}.entry, L.inh)$

Exercise:
turn the L-attributed SDD into an SDT

2) Build the parse-tree with semantic actions for: “**real id1 , id2 , id3**”

Summary

- Attribute grammars, Attribute types
- Syntax directed translation
- Syntax directed translation in LL parsing