Principles of Programming Languages

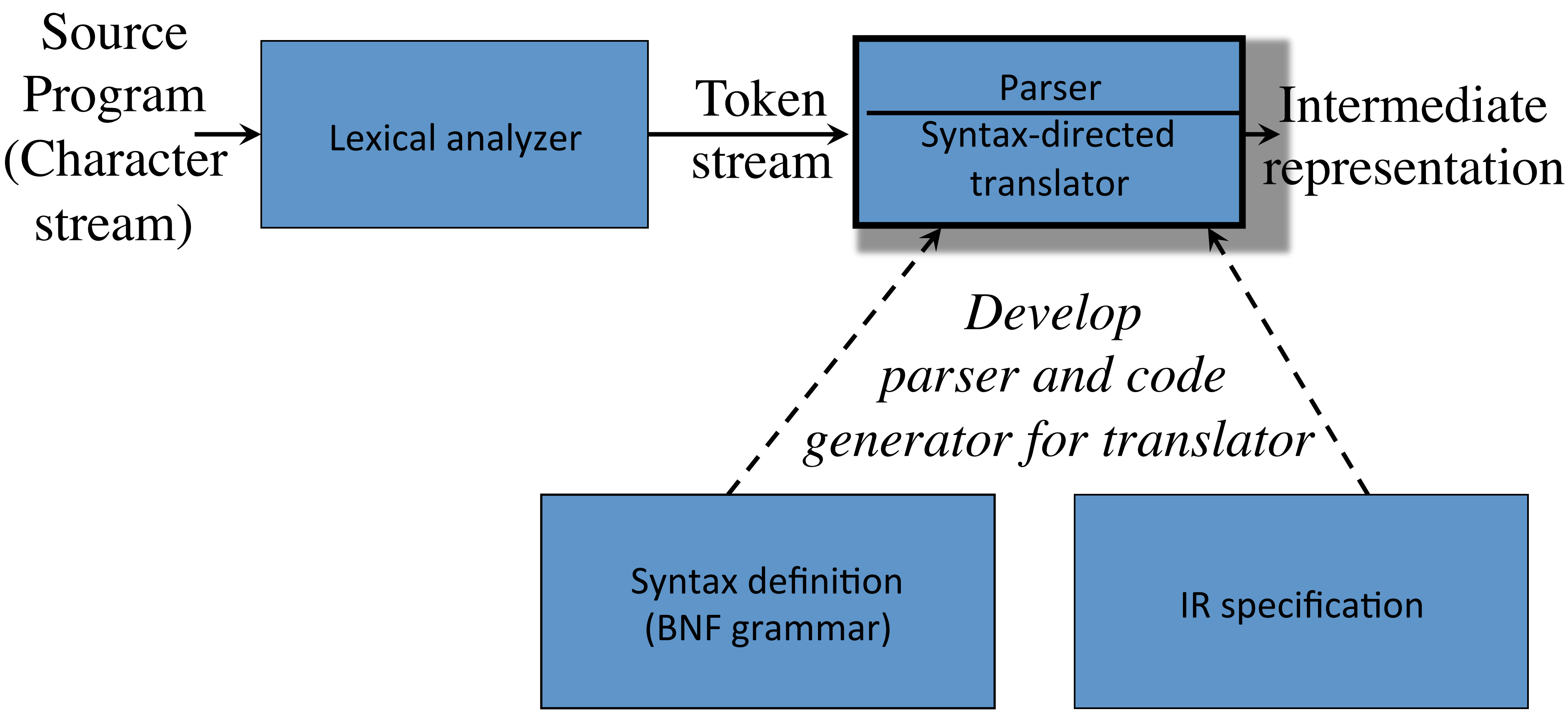
http://www.di.unipi.it/~andrea/Didattica/PLP-14/

Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 11

- Syntax-Directed Translation

The Structure of the Front-End



Syntax-Directed Translation

- Briefly introduced in the first lectures
- General technique to “manipulate” programs, based on context-free grammars
- Tightly bound with parsing
- Will be used for static analysis (type checking) and (intermediate) code generation
- Several other uses:
  - Generation of abstract syntax trees
  - Evaluation of expressions
  - Implementation of Domain Specific Languages (see example on typesetting math formulas in the book)
  - …
- Partly supported by parser generators like Yacc

Syntax-Directed Translation

- We will consider:
  - Syntax-Directed Definitions (*Attribute Grammars)*
  - Syntax-Directed Translation Schemes
  - Translation evaluated on the parse tree
  - Translation evaluated during parsing

Syntax-Directed Definitions

- A *syntax-directed definition* (or *attribute grammar*) is a context free grammar where:
  - Terminals and nonterminals have *attributes* holding values
  - Productions have associated *semantic rules,* which set the values of attributes
- Each grammar symbol can have any number of attributes
- Attribute *val* of symbol *X* is denoted *X.val*
- A semantic rule of a production $A \to \alpha$ has the form

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

  where $b, c_1, c_2, \ldots, c_k$ are attributes of *A* or of the symbols in α

Attributes

- An *annotated parse tree* is a parse tree where all attributes of nodes have the corresponding value
- Attribute values may represent
  - Numbers (literal constants)
  - Strings (literal constants)
  - Intermediate program representations
    - (pointers to) nodes of abstract syntax tree
    - (pointers to) strings representing the intermediate code
  - 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

Synthesized vs. Inherited Attributes

- Given a production $A \rightarrow \alpha$ and a semantic rule $b := f(c_1, c_2, \ldots, c_k)$
  - if *b* is an attribute of *A* then it is a *synthesized* attribute of *A*
  - if *b* is an attribute of a symbol B in α then it is an *inherited* attribute *B*
- **Note**: terminal symbols have only synthesized attributes

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

inherited (L.in)

synthesized (T.type)

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 single bottom-up traversal
- Given a parse tree, any *postorder depth-first traversal* algorithm can be used to execute the semantic rules and assign attribute values
- In some cases attributes can be evaluated during parsing, without building the parse tree explicitly
- [Yacc/Bison only support S-attributed definitions]

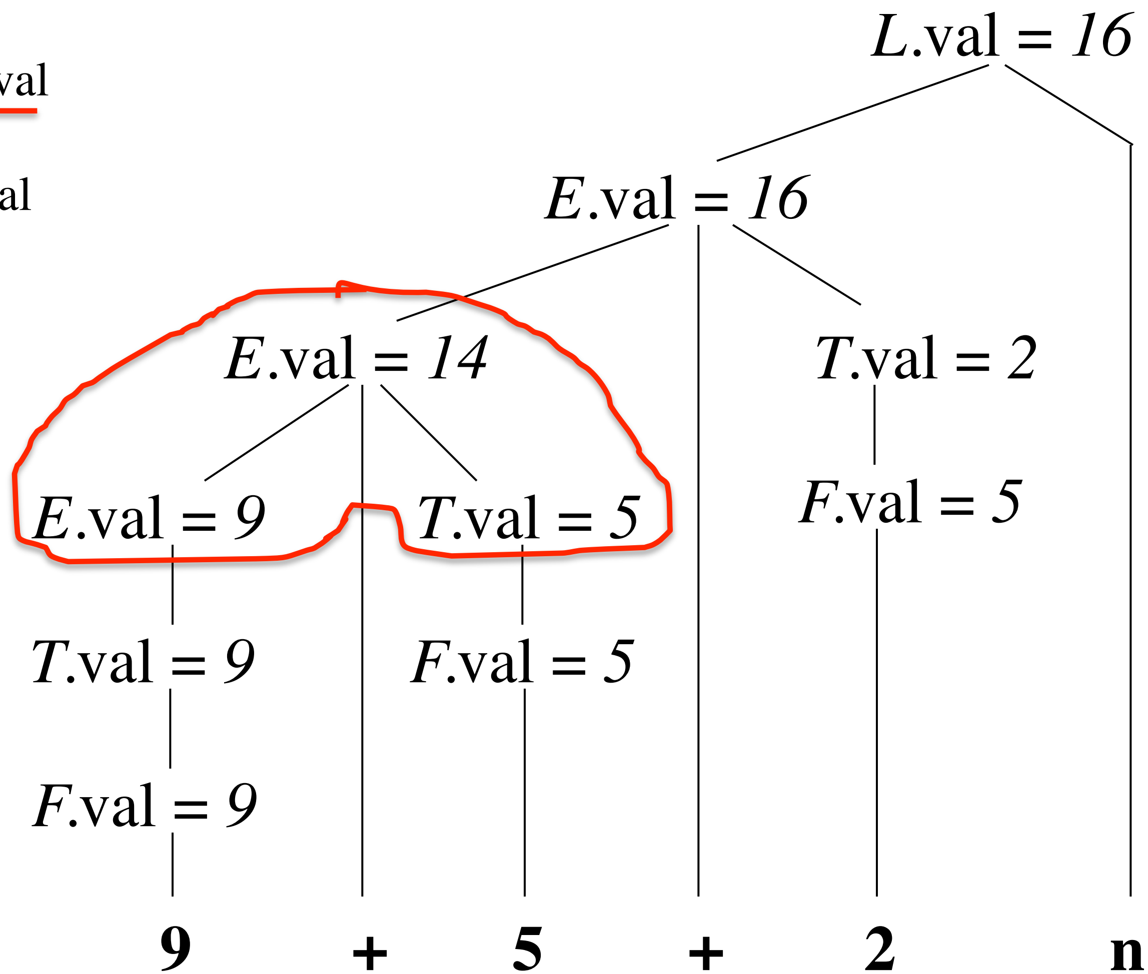
Example: evaluating expressions with synthesized attributes

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

A *Syntax-Directed Definition* (*SDD*) or *Attribute Grammar*

Example: An Annotated Parse Tree

Productions	Semantic Rule
$L \rightarrow E$ **n**	L.val := E.val
$E \rightarrow E_1 + T$	E.val := E_1.val + T.val
$E \rightarrow T$	E.val := T.val
$T \rightarrow T_1 * F$	T.val := T_1.val * F.val
$T \rightarrow F$	T.val := F.val
$F \rightarrow (E)$	F.val := E.val
$F \rightarrow$ **digit**	F.val := **digit**.lexval

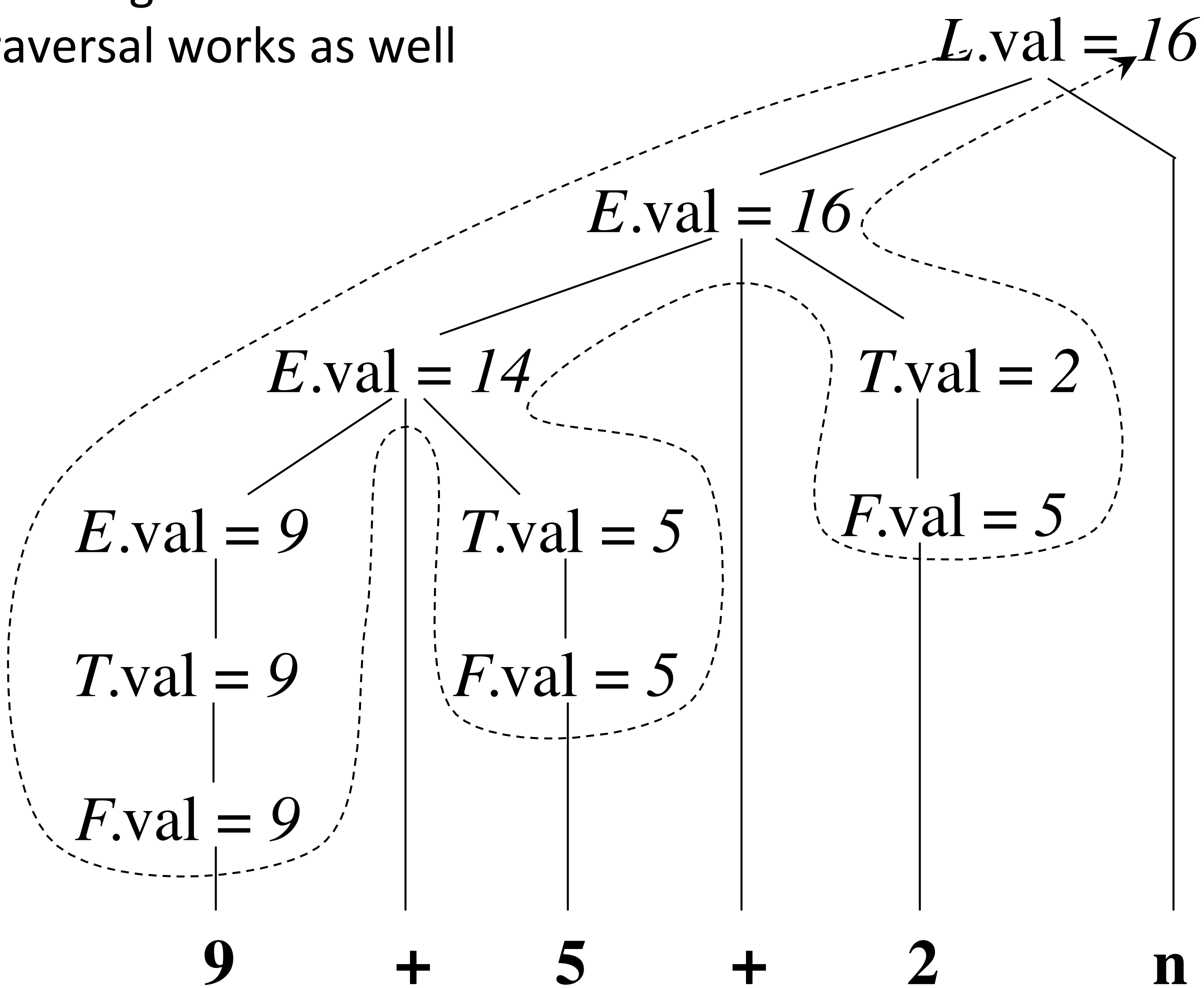


Annotating a Parse Tree with Depth-First Post-Order Traversals

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

Depth-First Traversals (Example)

Note: right-to-left traversal works as well

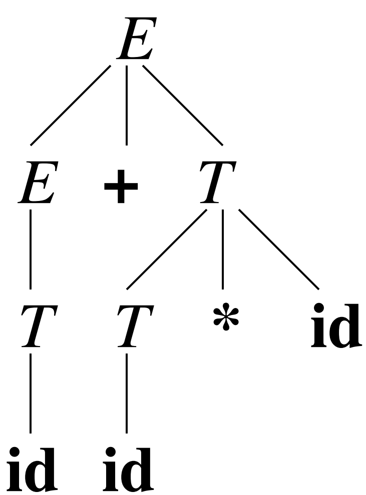


Semantic Rules

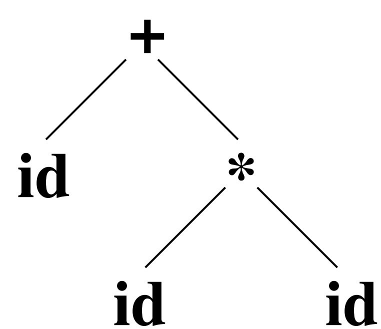
L.val := *E*.val
E.val := E_1.val + *T*.val
E.val := *T*.val
T.val := T_1.val * *F*.val
T.val := *F*.val
F.val := *E*.val
F.val := **digit**.lexval

Example: generation of Abstract Syntax Trees

- 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



Concrete syntax tree

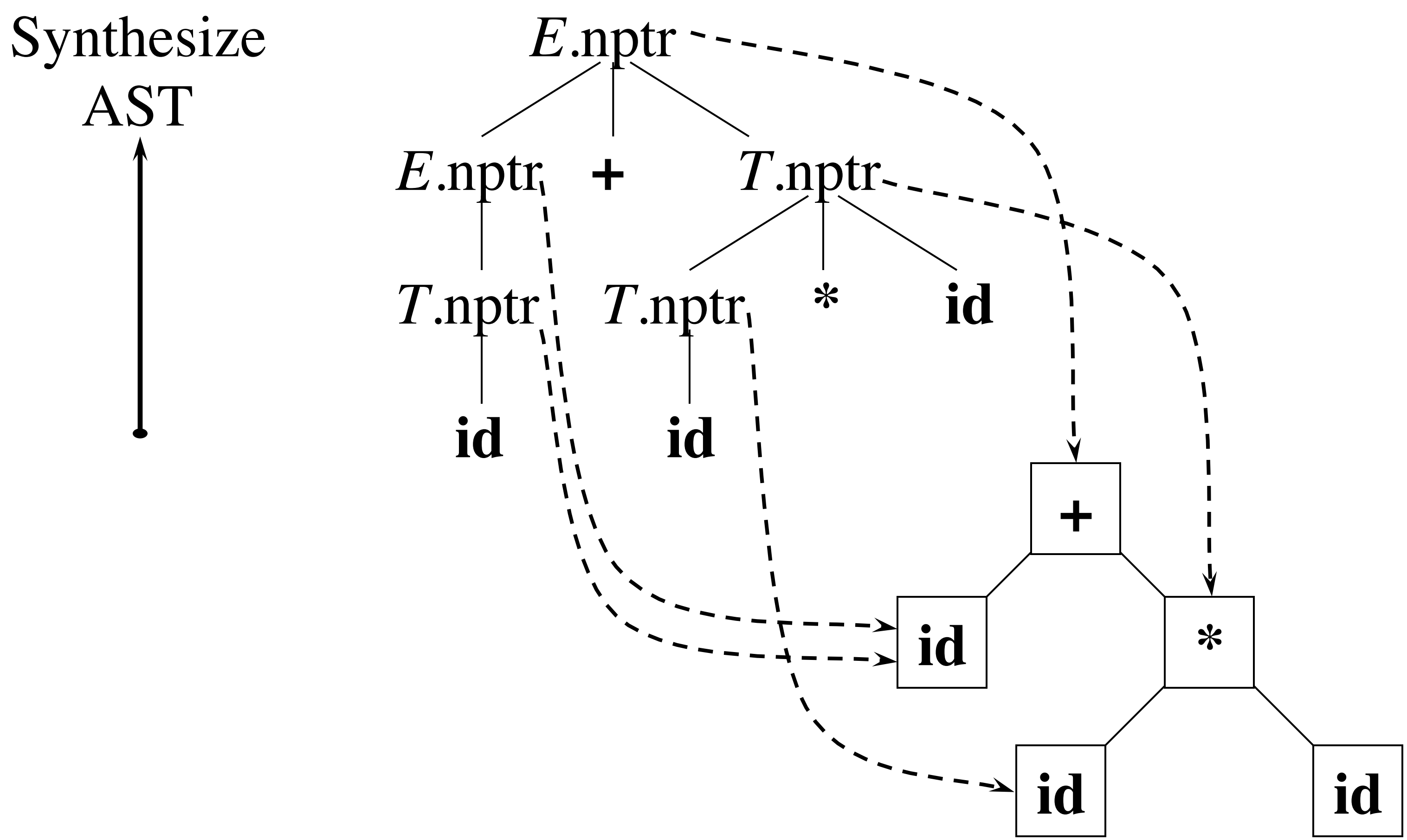


Abstract syntax tree

S-Attributed Definitions for Generating Abstract Syntax Trees

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)

Generating Abstract Syntax Trees



Example Attribute Grammar with Synthesized + Inherited Attributes

- Grammar generating declaration of typed variables
- *Limited side-effect:* The attributes add typing information to the symbol table via side effects
- Inherited attributes allow to handle information that does not respect the tree structure

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

Evaluation order of attributes

- In presence of inherited attributes, it is not obvious in what order the attributes should be evaluated
- Attributes of a nonterminal in a production may depend in an arbitrary way on attributes of other symbols
- The evaluation order must be consistent with such dependencies
- If the dependencies are circular, there is no way to evaluate the attributes
- Side-effect rules are interpreted as definition of dummy synthesized attributes of the head symbol

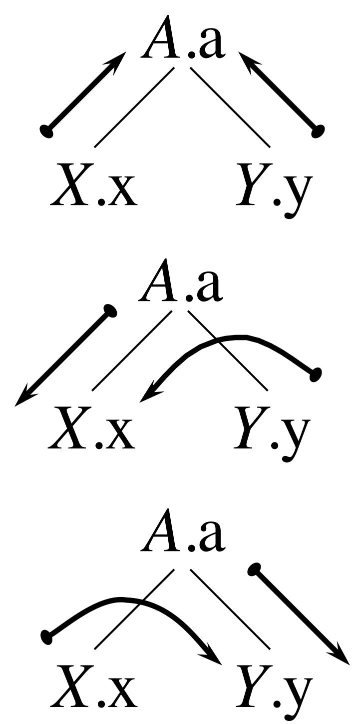
Dependency Graphs for Attributed Parse Trees

- Given a parse tree, consider as nodes all its attributes
- Represent graphically the dependencies induced by semantic rules

$A \rightarrow X\ Y$

Direction of

value dependence



$A.\mathrm{a} := f(X.\mathrm{x}, Y.\mathrm{y})$

synthesized

$X.\mathrm{x} := f(A.\mathrm{a}, Y.\mathrm{y})$

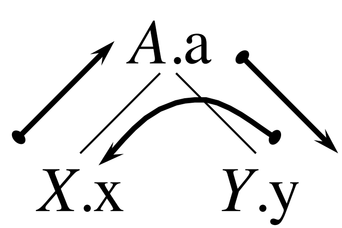
inherited

$Y.\mathrm{y} := f(A.\mathrm{a}, X.\mathrm{x})$

inherited

Dependency Graphs with Cycles?

- Edges in the dependency graph determine the evaluation order for attribute values
- Dependency graphs cannot be cyclic: no evaluation order is possible



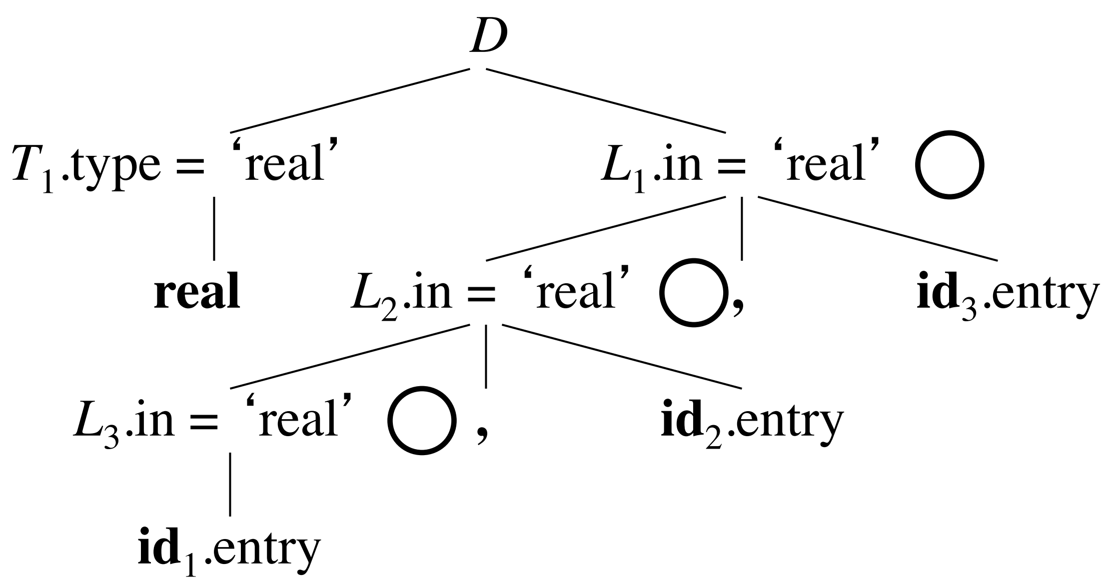
$$A.\mathrm{a} := f(X.\mathrm{x})$$
$$X.\mathrm{x} := f(Y.\mathrm{y})$$
$$Y.\mathrm{y} := f(A.\mathrm{a})$$

Error: cyclic dependence

Example Annotated Parse Tree

$D \rightarrow T\ L$	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)

real id, id, id

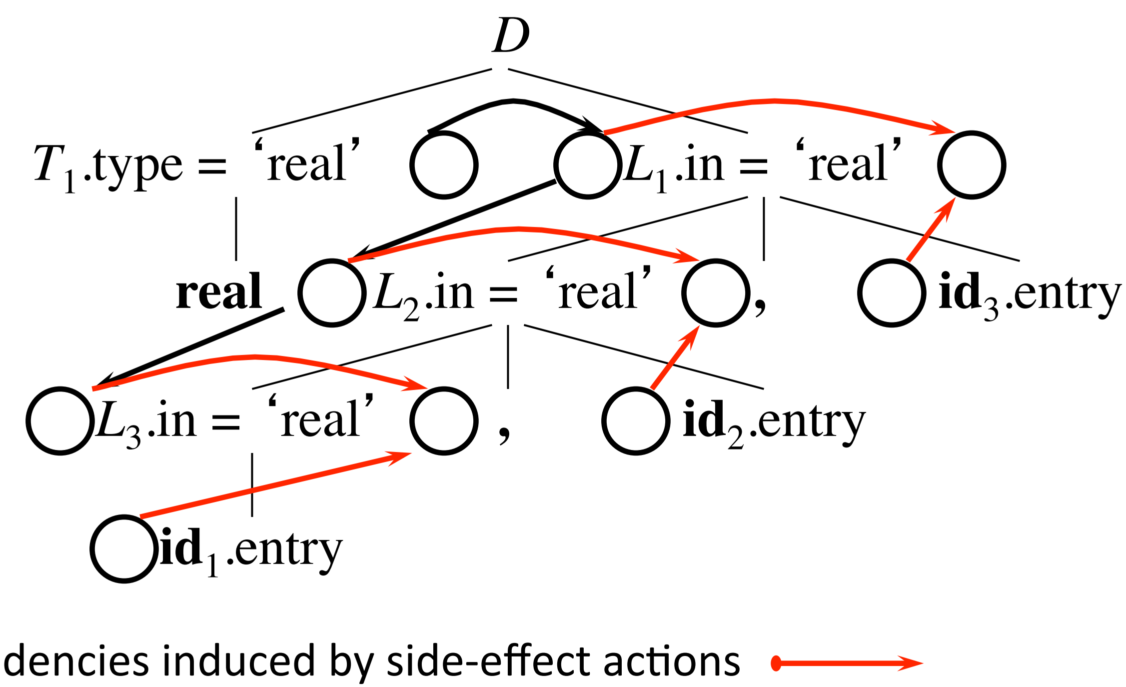


○ Dummy synthesized attribute of *L* induced by side-effect actions

Example Annotated Parse Tree with Dependency Graph

$D \rightarrow T\ L$	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)

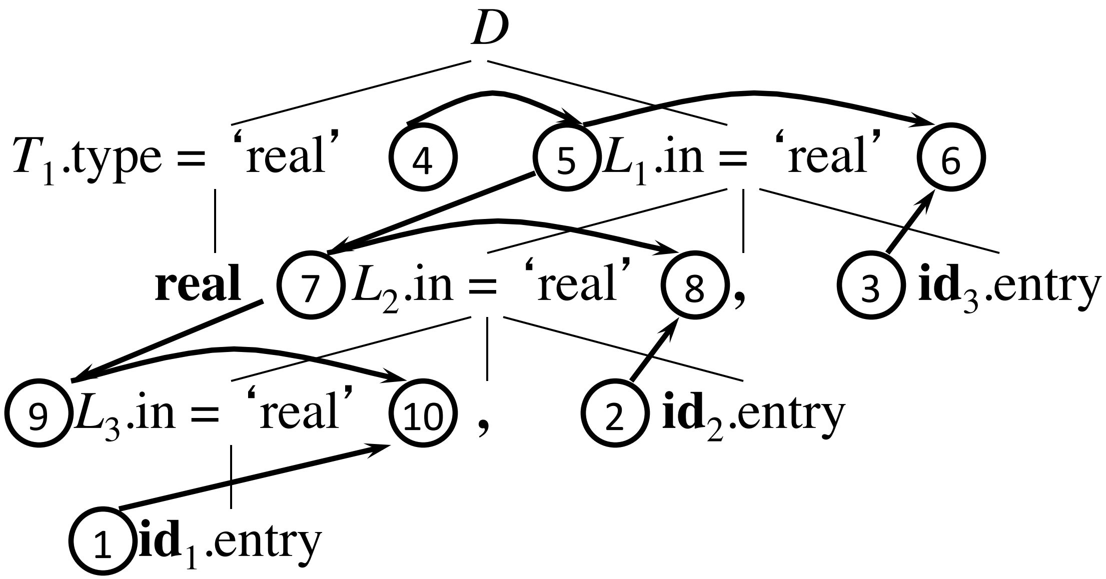
real id, id, id



Evaluation Order

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

Example Parse Tree with Topologically Sorted Actions



Topological sort:

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)

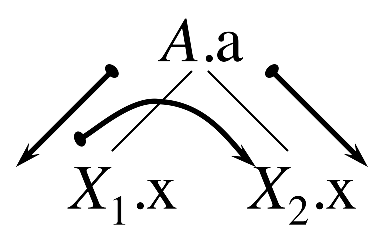
Evaluation Methods

- *Parse-tree methods*: for each input, build the parse tree and the dependency graph, and determine an evaluation order from a topological sort
- *Rule-base methods* the evaluation order is pre-determined from the semantic rules
- *Oblivious methods* the evaluation order is fixed and semantic rules must be (re)written to support the evaluation order (for example S-attributed definitions)

L-Attributed Definitions

- A syntax-directed definition is *L-attributed* if each <u>inherited</u> attribute of X_j on the right side of $A \to X_1 X_2 \dots X_n$ depends only on
  1. the attributes of the symbols $X_1, X_2, \dots, X_{j\text{-}1}$
  2. the inherited attributes of A
  3. the attributes of X_j, in a non-circular way

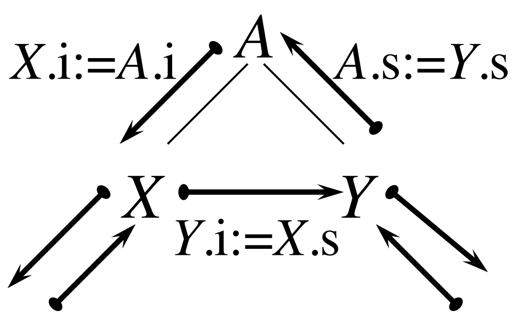
Possible dependences of inherited attributes



L-Attributed Definitions (cont'd)

- L-attributed definitions allow for a natural order of evaluating attributes: depth-first and left to right

$A \rightarrow X\ Y$



$X.\mathrm{i} := A.\mathrm{i}$
$Y.\mathrm{i} := X.\mathrm{s}$
$A.\mathrm{s} := Y.\mathrm{s}$

- Note: every S-attributed syntax-directed definition is also L-attributed (since it doesn't have any inherited attribute)

Syntax-Directed Translation Schemes

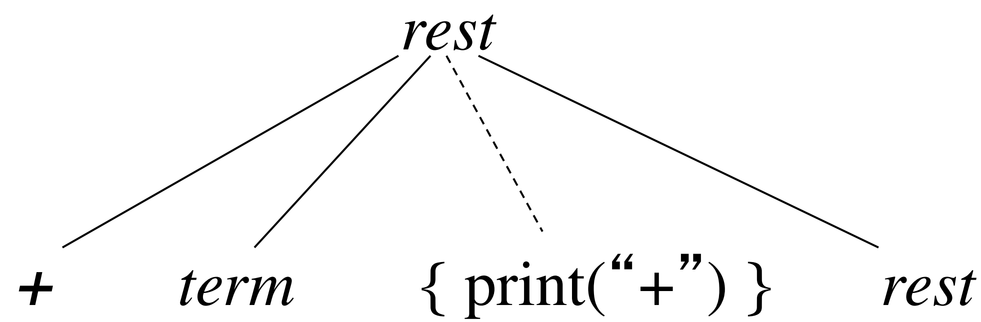
- Translation Schemes provide an alternative notation for Syntax-Directed Definitions and are more expressive in general
- The semantic rules include arbitrary side-effects and can be suitably embedded into productions
- SDD's can be evaluated in *any order* compatible with the dependencies, SDT's should be evaluated left-to-right
- SDT's can always be implemented by building the parse tree first, and then performing the actions in left-to-right depth-first order
- In several cases they can be implemented during parsing, without building the whole parse tree first

Syntax-Directed Translation Schemes

- A production of a translation scheme and corresponding node in a parse tree:

rest → **+** *term* { print("+") } *rest*

Embedded semantic action



- Possible implementation:
  – Build the parse tree ignoring semantic actions
  – Add semantic actions in the right place below non-terminals
  – Visit the tree in ***depth-first, pre-order left-to-right*** traversal executing all actions

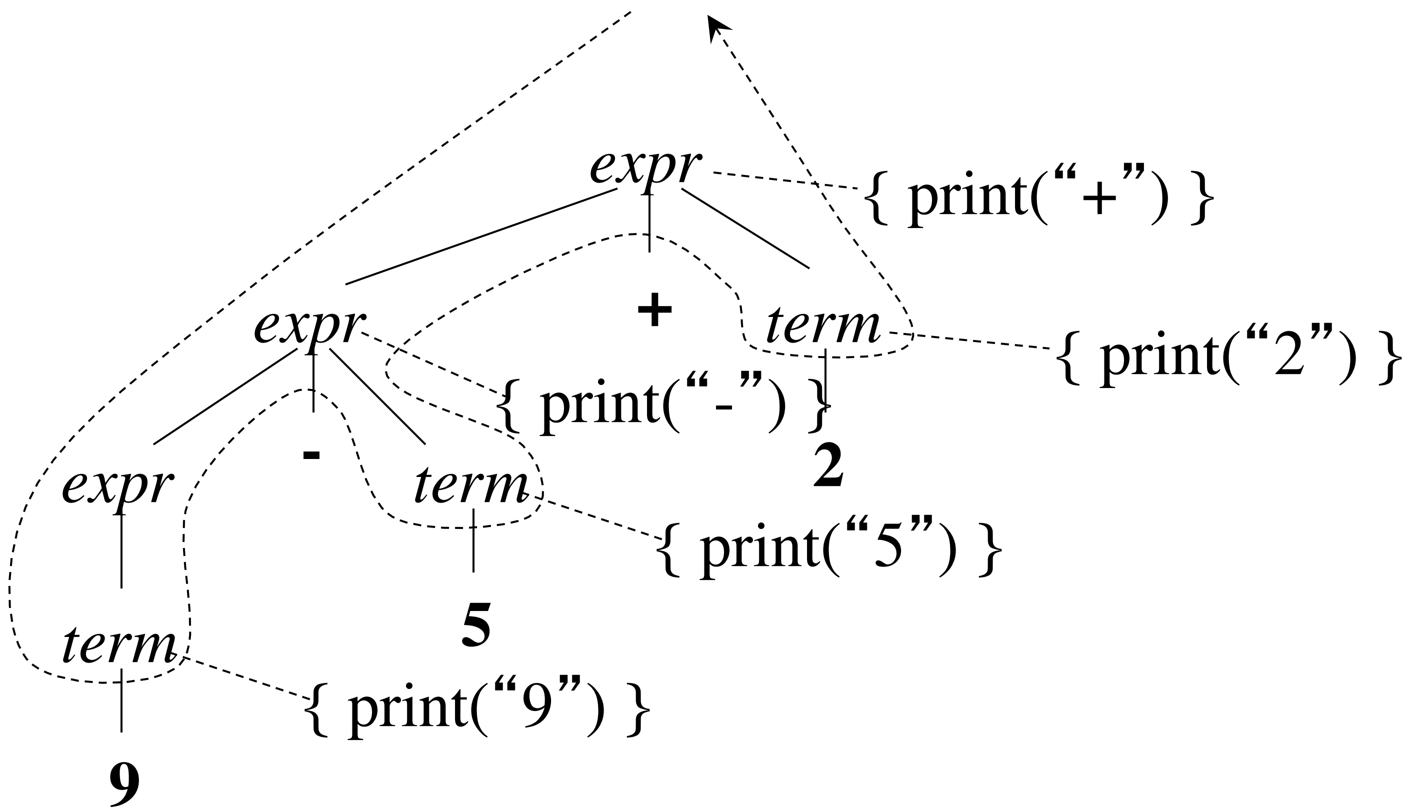
Postfix Translation Schemes

- If the grammar is LR (thus can be parsed bottom-up) and the SDD is S-attributed (synthesized attributes only), semantic actions can be placed at the end of the productions
- They are executed when the body is reduced to the head
- SDTs where all actions are at the end of productions are called *postfix SDTs*

Example Translation Scheme for Postfix Notation

expr → *expr* **+** *term*	{ print("+") }
expr → *expr* **-** *term*	{ print("-") }
expr → *term*	
term → **0**	{ print("0") }
term → **1**	{ print("1") }
…	…
term → **9**	{ print("9") }

Example Translation Scheme (cont'd)



Translates **9-5+2** into postfix **95-2+**

Implementation of Postfix SDTs

- Postfix SDTs can be implemented during LR parsing
- The actions are executed when reductions occur
- The attributes of grammar symbols can be put on the stack, together with the symbol or the state corresponding to it
- Since all attributes are synthesized, the attribute for the head can be computed when the reduction occurs, because all attributes of symbols in the body are already computed

Using Translation Schemes for L-Attributed Definitions

- An L-attributed SDD for a grammar that can be parsed top-down (LL) can be implemented using Translation Schemes

1. Embed actions that compute **inherited attributes** for nonterminal *A* immediately before *A*
   – Note that in left-to-right parsing, all attributes on which the inherited attribute may depend are evaluated already
2. Place actions that compute a **synthesized attribute** for the head of a production at the end of the body of that production

- We consider some ways of implementing the resulting SDTs during parsing, both top-down and bottom-up

Using Translation Schemes for L-Attributed Definitions

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

$D \rightarrow T$ { L.in := T.type } L

$T \rightarrow$ **int** { T.type := ‘integer’ }

$T \rightarrow$ **real** { T.type := ‘real’ }

$L \rightarrow$ { L_1.in := L.in } L_1 **, id** { *addtype*(**id**.entry, L.in) }

$L \rightarrow$ **id** { *addtype*(**id**.entry, L.in) }

Implementing L-Attributed Definitions in Recursive-Descent Parsers

Recap of Recursive Descent Parsing

- Grammar must be LL(1)
- Every nonterminal has one (recursive) procedure
- When a nonterminal has multiple productions, the input look-ahead is used to choose one
- Note: the procedures have no parameters and no result

expr → *term rest*
rest → **+** *term rest*
| **-** *term rest*
| ε
term → **id**

procedure *rest*();
begin
  if *lookahead* in FIRST(**+** *term rest*) **then**
    match(‘**+**’); *term*(); *rest*()
  else if *lookahead* in FIRST(**-** *term rest*) **then**
    match(‘**-**’); *term*(); *rest*()
  else if *lookahead* in FOLLOW(*rest*) **then**
    return
  else error()
end;

Implementing L-Attributed Definitions in Recursive-Descent Parsers

- Attributes are passed as arguments to procedures (*inherited*) or returned (*synthesized*)
- Procedures store computed attributes in local variables

D → *T* { *L*.in := *T*.type } *L*
T → **int** { *T*.type := 'integer' }
T → **real** { *T*.type := 'real' }

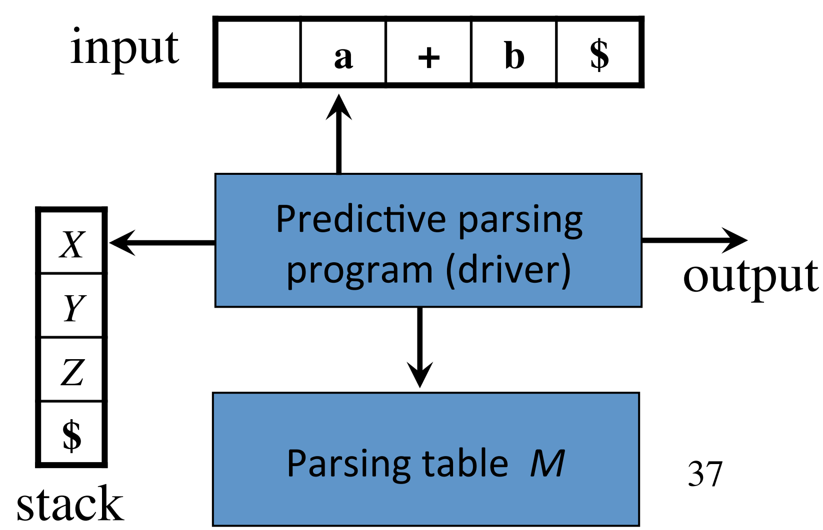
```
void D()
{ Type Ttype = T();
  Type Lin = Ttype;
  L(Lin);
}
Type T()
{ Type Ttype;
  if (lookahead == INT)
  { Ttype = TYPE_INT;
    match(INT);
  } else if (lookahead == REAL)
  { Ttype = TYPE_REAL;
    match(REAL);
  } else error();
  return Ttype;
}
void L(Type Lin)
{ … }
```

Output: synthesized attribute (return Ttype)

Input: inherited attribute (Lin)

Implementing L-Attributed Definitions in Top-Down Table-Driven Parsers

- The stack will contain, besides grammar symbols, *action-records* and *synthesize-records*
- Inherited attributes of *A* are placed in *A*'s record
  - The code computing them is in a record above *A*
- Synthesized attributes of *A* are placed in a record just below *A*
- It may be necessary to make copies of attributes to avoid that they are popped when still needed



Implementing L-Attributed Definitions for **LL grammars** in Bottom-Up Parsers

- Remove any embedded action with *marking nonterminal:* $A \rightarrow \alpha \{ act \} \beta$ becomes

  $A \rightarrow \alpha N \beta$

  $N \rightarrow \varepsilon \{ act' \}$

  where act':

  - Copies as inherited attributes of *N* any attribute of *A, α* needed by *act*
  - Computes attributes like *act*, making them synthesized for N
- Fact: if the start grammar was LL, the new one is LR
- Note: *act'* accesses attributes out of its production! This works, as they are (deeper) in the LR stack