Principles of Programming Languages

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

Prof. Andrea Corradini
Department of Computer Science, Pisa

Lesson 10

- Continuation of the course
- Syntax-Directed Translation (1)

Continuation of the course

- [Nov-Dec 2014] 22 h
  - Introduction to compilers
  - Lexical analysis
  - Parsing
- [Feb-May 2015] ~50 h
  - Syntax directed translation
  - Intermediate code generation
  - Code generation

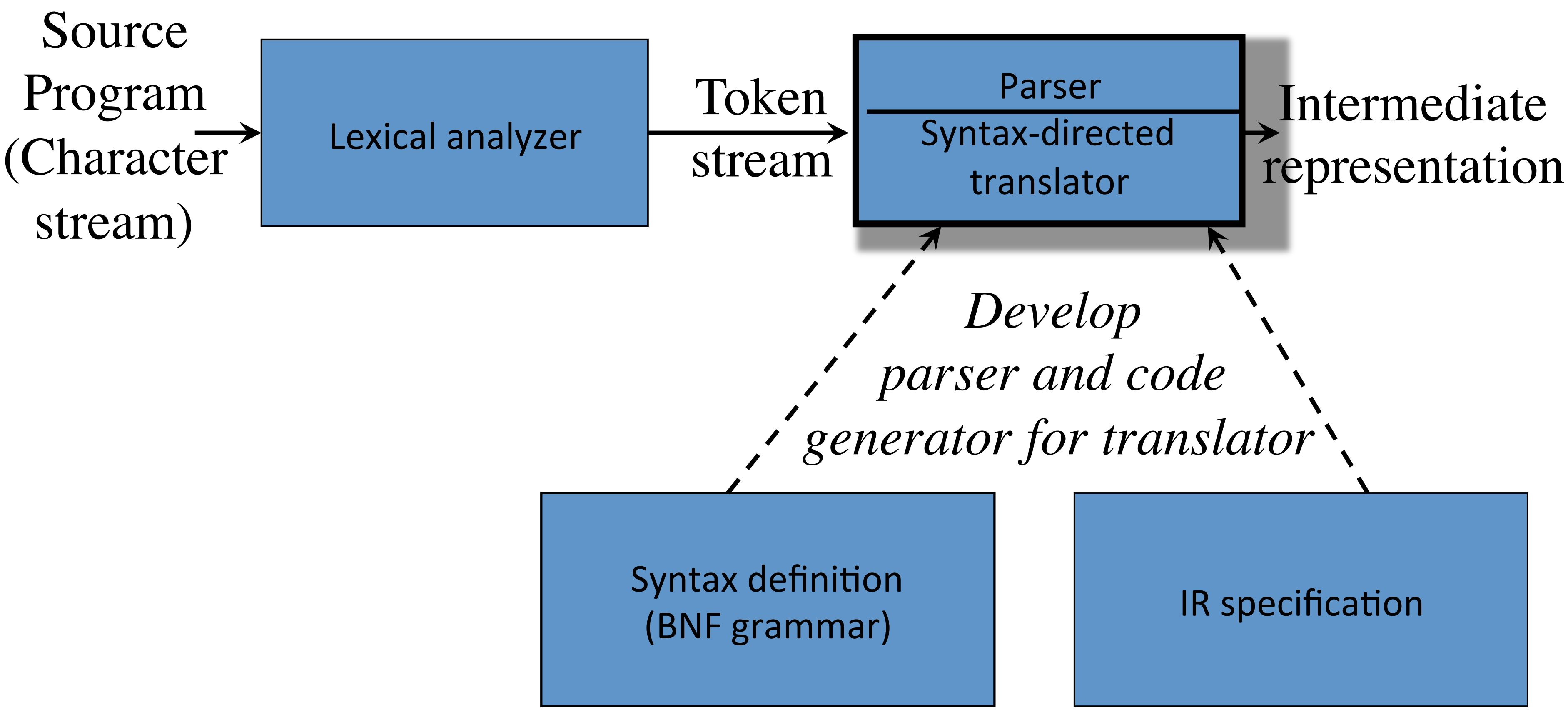
  ===================================
  - Concepts of Programming Languages
  - <to be detailed…>

  ===================================
  - May 27-29: 2nd Mid-Term Exam
  - Can be taken by everybody

Continuation of the course (2)

- Office hours: **Wednesday, 4-6 pm**
- 9 Credits vs. 12 Credits: still a problem for somebody?
- Important: no lectures on
  - Friday, March 6
  - Tuesday, March 17
  - Friday, March 20
- Need to recover several lectures with 3 lectures per week
  - Possible days and hours: **Thursday, 2-4 pm**

The Structure of the Front-End



Source Program (Character stream)

Lexical analyzer

Token stream

Parser

Syntax-directed translator

Intermediate representation

Develop parser and code generator for translator

Syntax definition (BNF grammar)

IR specification

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 Definitions

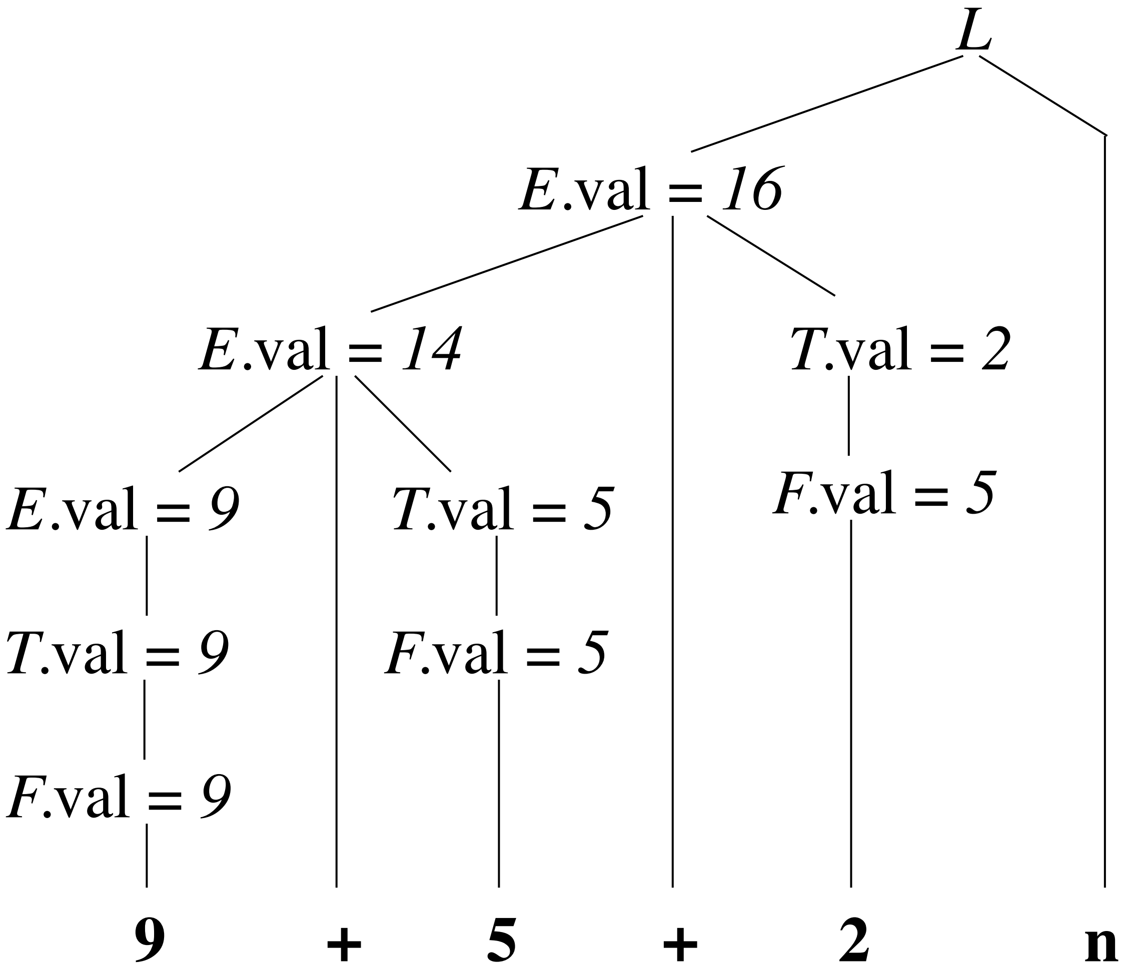
- A *syntax-directed definition* (or *attribute grammar*) binds a set of *semantic rules* to productions
- Terminals and nonterminals have *attributes* holding values, which are set by the semantic rules
- A *depth-first (postorder) traversal* algorithm traverses the parse tree executing semantic rules to assign attribute values
- After the traversal is complete the attributes contain the translated form of the input

Example: evaluating expressions with *synthesized* attributes

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

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

Example: An Annotated Parse Tree


L

$E.\text{val} = 16$

$E.\text{val} = 14$

$T.\text{val} = 2$

$E.\text{val} = 9$

$T.\text{val} = 5$

$F.\text{val} = 5$

$T.\text{val} = 9$

$F.\text{val} = 5$

$F.\text{val} = 9$

9 **+** **5** **+** **2** **n**

Productions

$L \rightarrow E\ \mathbf{n}$

$E \rightarrow E_1 + T$

$E \rightarrow T$

$T \rightarrow T_1 * F$

$T \rightarrow F$

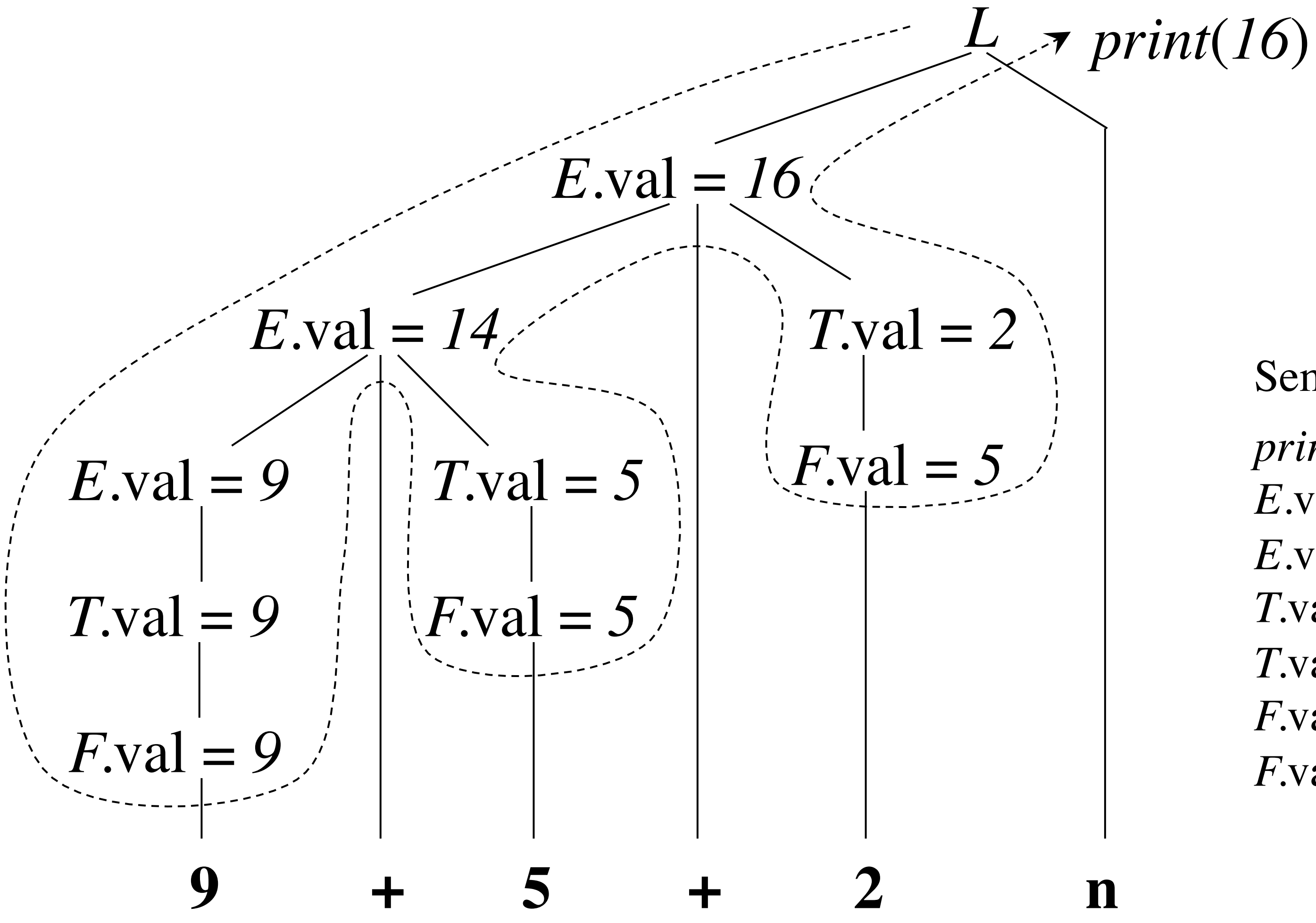
$F \rightarrow (\ E\ )$

$F \rightarrow \mathbf{digit}$

Annotating a Parse Tree with Depth-First 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)



Semantic Rules

print(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

Attributes

- Each grammar symbol can have any number of attributes
- Attribute values typically 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

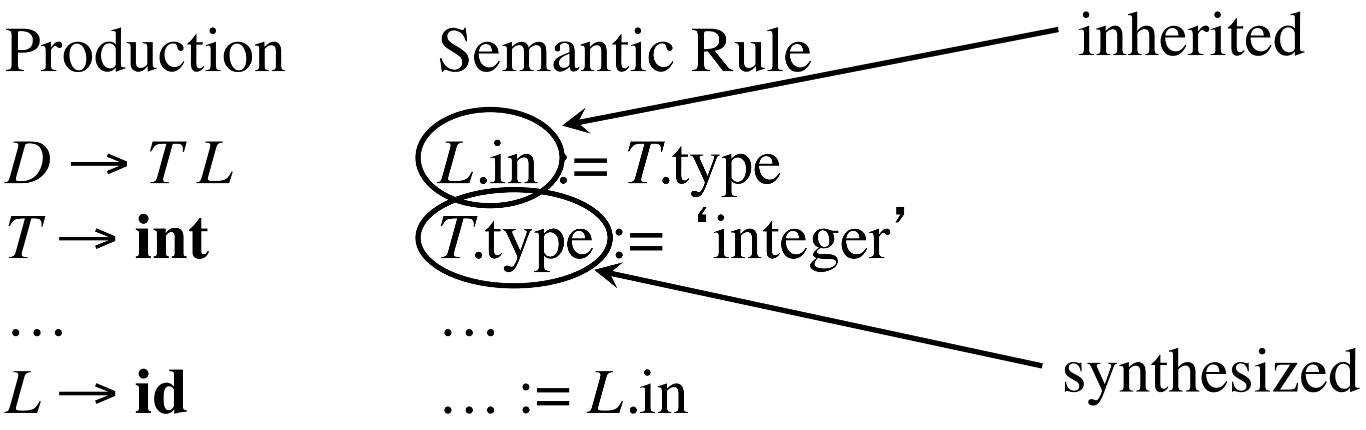
  $A \rightarrow \alpha$

  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 Versus Inherited Attributes (cont’d)



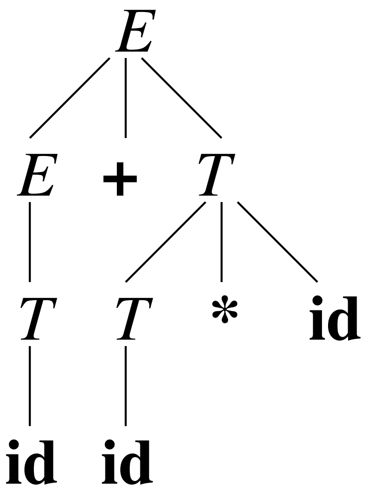
Production	Semantic Rule
$D \rightarrow T\ L$	$\boxed{L\text{.in}}$:= T.type
$T \rightarrow$ **int**	$\boxed{T\text{.type}}$:= ‘integer’
…	…
$L \rightarrow$ **id**	… := L.in

S-Attributed Definitions

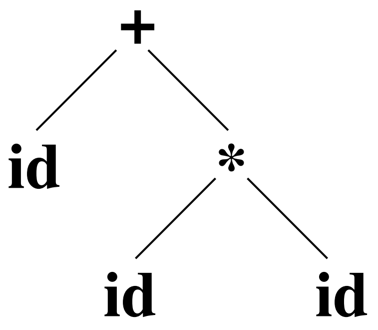
- A syntax-directed definition that uses synthesized attributes exclusively is called an *S-attributed definition* (or *S-attributed grammar*)
- A parse tree of an S-attributed definition can be annotated with a single bottom-up traversal
- [Yacc/Bison only support S-attributed definitions]

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

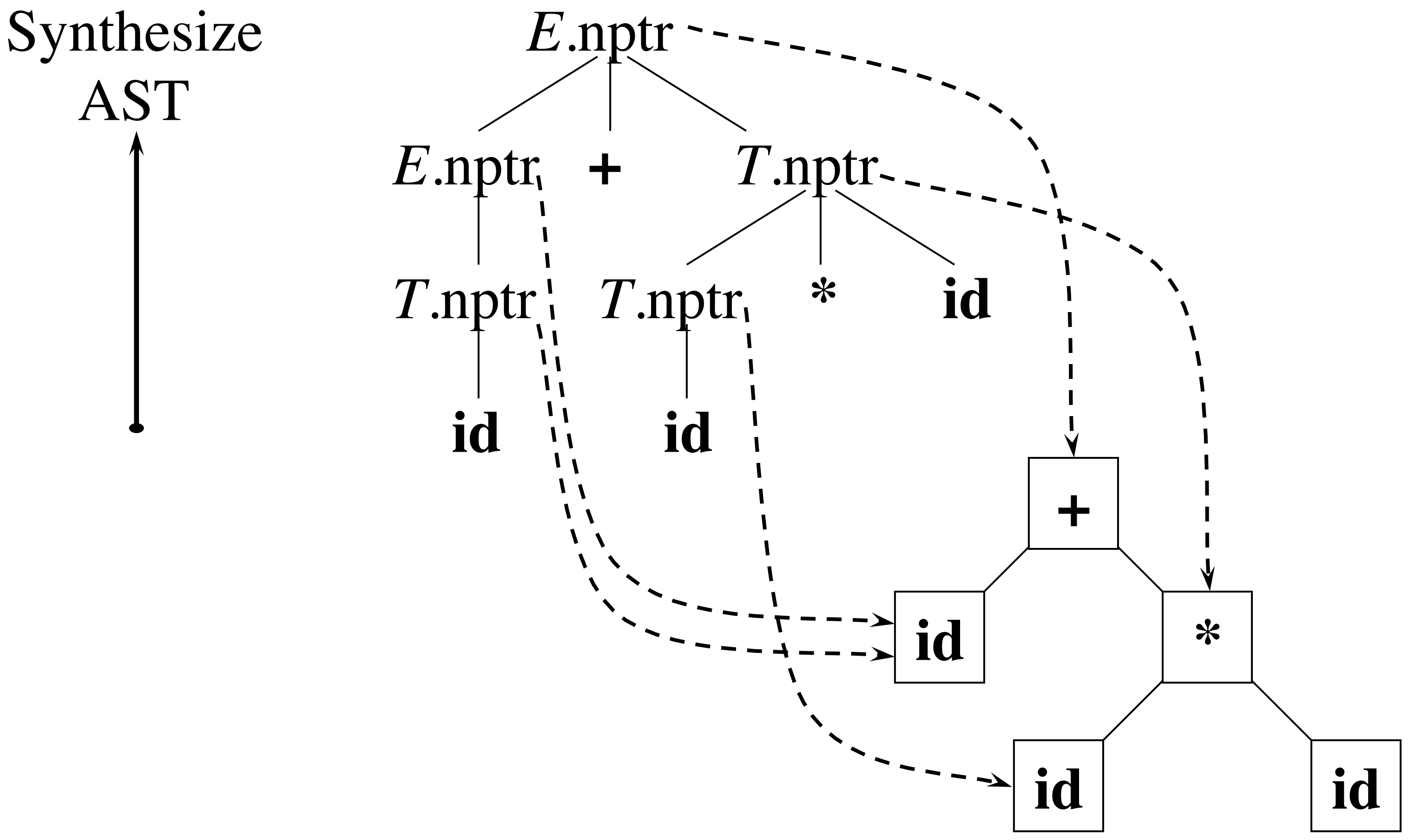


Abstract syntax tree

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
- The attributes add typing information to the symbol table via side effects

Production	Semantic Rule
$D \rightarrow T\ L$	$L.\mathrm{in} := T.\mathrm{type}$
$T \rightarrow \mathbf{int}$	$T.\mathrm{type} :=$ ‘integer’
$T \rightarrow \mathbf{real}$	$T.\mathrm{type} :=$ ‘real’
$L \rightarrow L_1\ \mathbf{,\,id}$	$L_1.\mathrm{in} := L.\mathrm{in};\ \mathit{addtype}(\mathbf{id}.\mathrm{entry}, L.\mathrm{in})$
$L \rightarrow \mathbf{id}$	$\mathit{addtype}(\mathbf{id}.\mathrm{entry}, L.\mathrm{in})$

Synthesized: $T.\mathrm{type}$, $\mathbf{id}.\mathrm{entry}$
Inherited: $L.\mathrm{in}$

Evaluation order of attributes

- In presence of inherited attributes, it is not obvious in which order
- the attributes can be evaluated

Grammar generating declaration of typed variables

- The attributes add typing information to the symbol table

via side effects

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

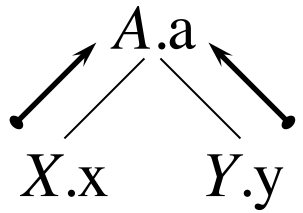
Inherited: L.in

Evaluation order of attributes

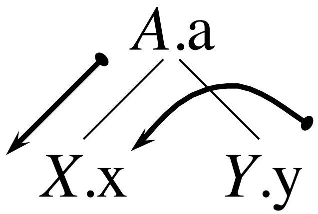
- In presence of inherited attributes, it is not obvious in what order the attributes should be evaluated
- Attributes of a nonterminal in a production may depend in an arbitrary way on attributes of other symbols
- The evaluation order must be consistent with such dependencies

Dependency Graphs for Attributed Parse Trees

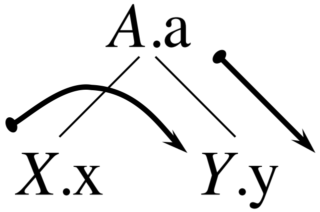
$A \to X\ Y$



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



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



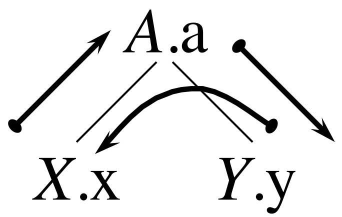
$Y.\mathrm{y} := f(A.\mathrm{a}, X.\mathrm{x})$

Direction of

value dependence

Dependency Graphs with Cycles?

- Edges in the dependency graph determine the evaluation order for attribute values
- Dependency graphs cannot be cyclic

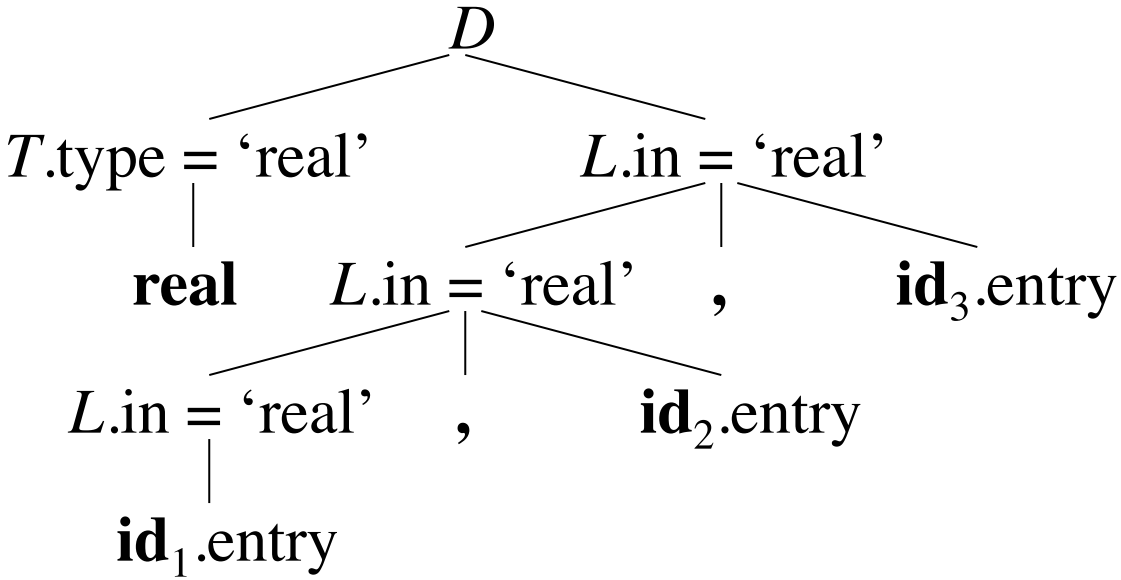


A.a := f(X.x)
X.x := f(Y.y)
Y.y := f(A.a)

Error: cyclic dependence

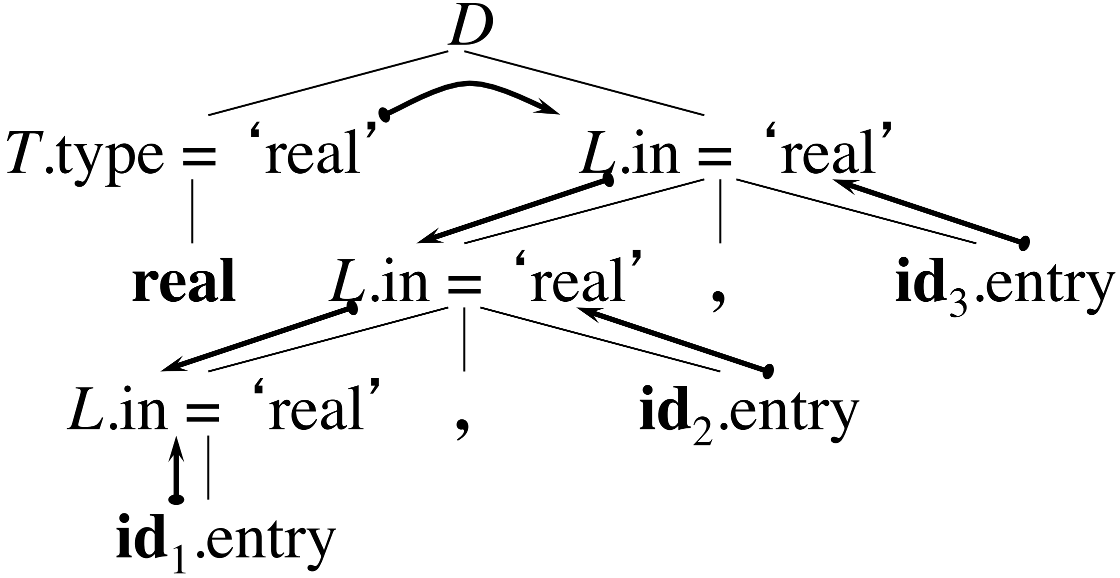
Example Annotated Parse Tree

$D \rightarrow T\ L$	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)



Example Annotated Parse Tree with Dependency Graph

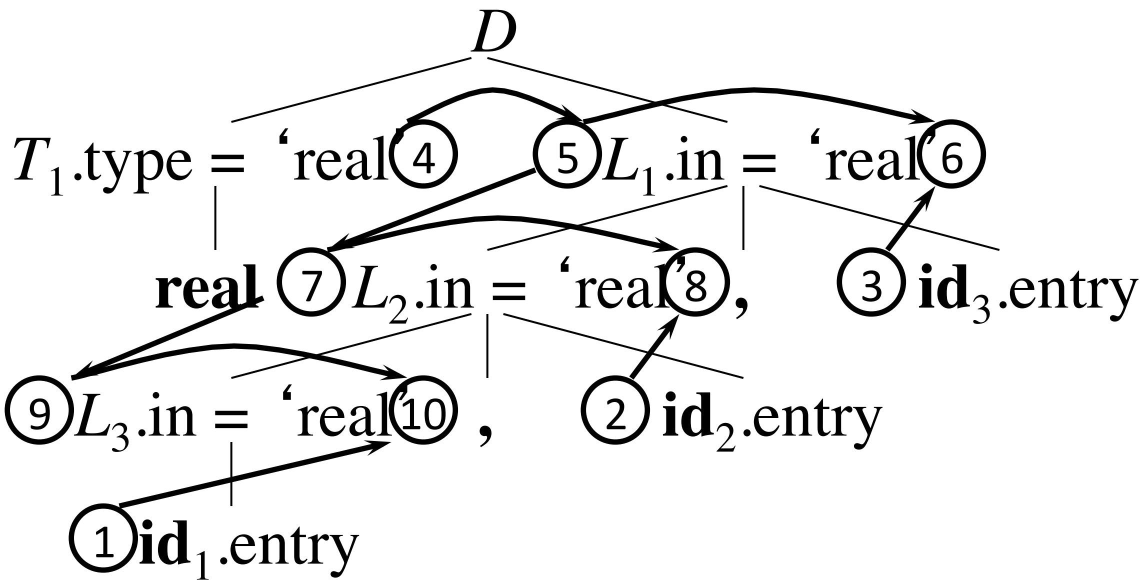
$D \rightarrow T\ L$	L.in := T.type
$T \rightarrow$ **int**	T.type := ‘integer’
$T \rightarrow$ **real**	T.type := ‘real’
$L \rightarrow L_1$ **, id**	L_1.in := L.in; *addtype*(**id**.entry, L.in)
$L \rightarrow$ **id**	*addtype*(**id**.entry, L.in)



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 $m_i \rightarrow m_j$ 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)

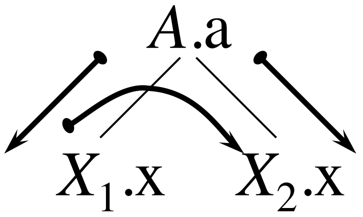
Evaluation Methods

- *Parse-tree methods* determine an evaluation order from a topological sort of the dependency graph constructed from the parse tree for each input
- *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 inherited attribute of X_j on the right side of $A \rightarrow X_1\ X_2 \ldots X_n$ depends only on
  1. the attributes of the symbols $X_1, X_2, \ldots, X_{j-1}$
  2. the inherited attributes of A

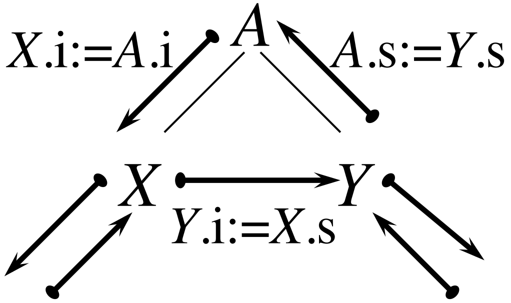
Possible dependences of inherited attributes



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.i:=A.i

A

A.s:=Y.s

X

Y

Y.i:=X.s

X.i := A.i

Y.i := X.s

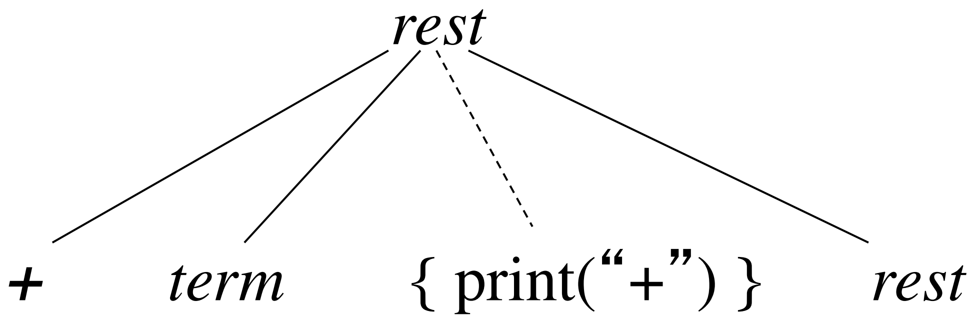
A.s := Y.s

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

Syntax-Directed Translation Schemes

- A *translation scheme* is a CF grammar embedded with *semantic actions*

$$rest \rightarrow \mathbf{+}\ term\ \underbrace{\{\ \mathrm{print}(“+”)\ \}}_{\text{Embedded semantic action}}\ rest$$



Syntax-Directed Translation Schemes

- Translation Schemes are an alternative notation for Syntax-Directed Definitions
- The semantic rules can be suitably embedded into productions
- 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

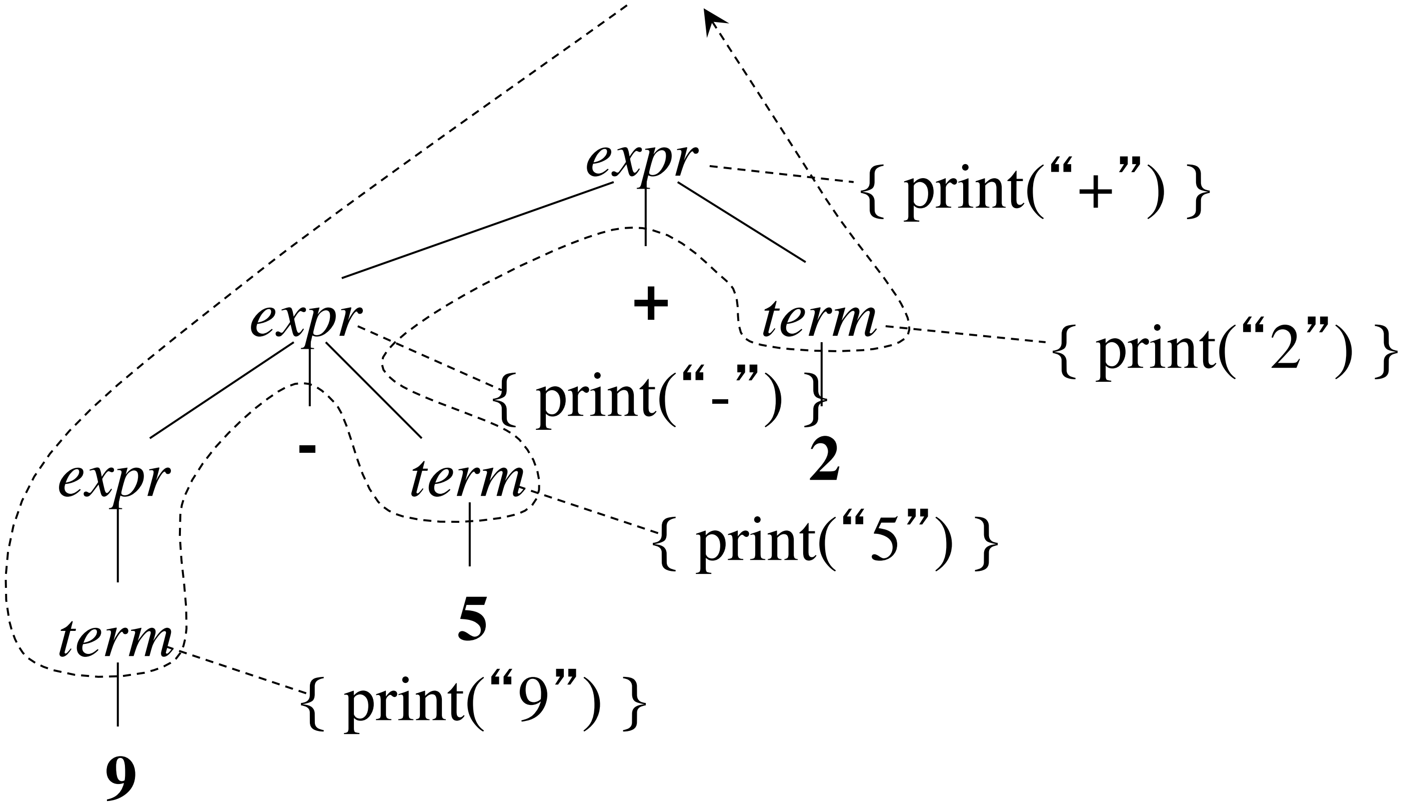
Postfix Translation Schemes

- 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
- These 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