

Chapter 4: Semantic analysis

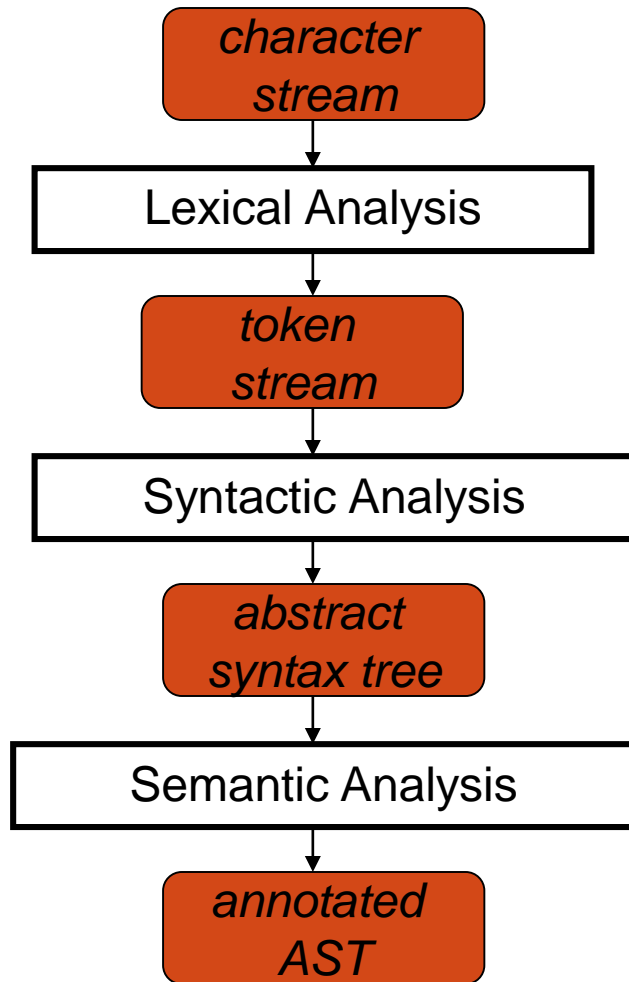
By Birku L.

Contents

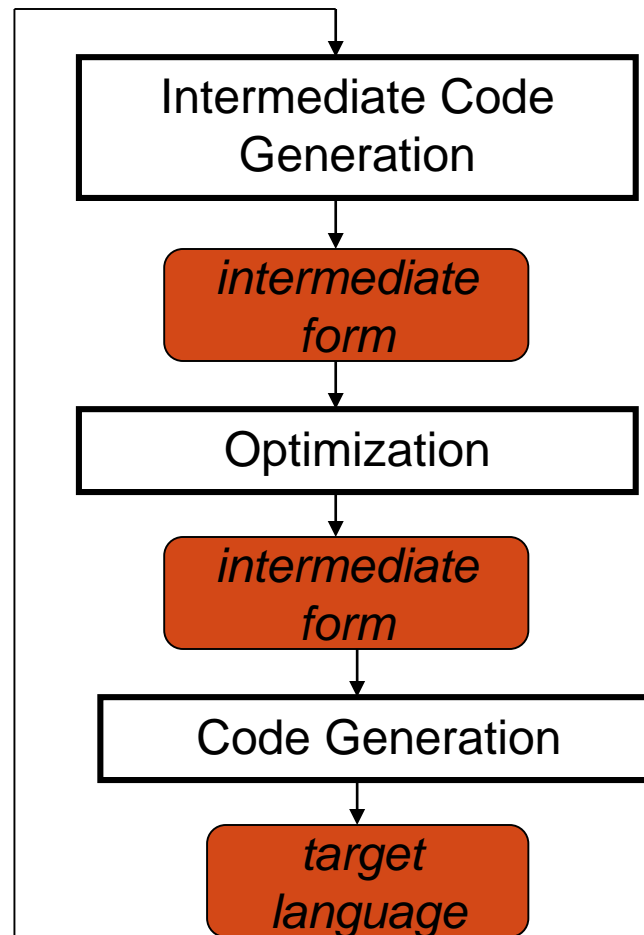
- Introduction
- Semantic analysis
- Syntax-Directed Translation (SDT)
 - Syntax Directed Definitions
 - Synthesized attributes
 - Inherited attributes
 - Implementing Attribute
 - Dependency Graph
 - S-Attributed Definitions
 - L-Attributed Definitions
 - Translation Schemes

Introduction

Analysis of input program
(**front-end**)



Synthesis of output program
(**back-end**)



Introduction...

- Program is **lexically well-formed**:
 - Identifiers have valid names.
 - Strings are properly terminated.
 - No stray characters.
- Program is **syntactically well-formed**:
 - Class declarations have the correct structure.
 - Expressions are syntactically valid.
- **Does this mean that the program is legal?**

Semantic analysis

```
class MyClass implements MyInterface {  
    string myInteger;  
    void doSomething() {  
        int[] x = new string;  
        x[5] = myInteger * y;  
    }  
    void doSomething() {  
    }  
    int fibonacci(int n) {  
        return doSomething() + fibonacci(n - 1);  
    }  
}
```

Semantic analysis...

```
class MyClass implements MyInterface {  
    string myInteger;  
  
    void doSomething() {  
        int[] x = new string;  
        x[5] = myInteger * y;  
    }  
    void doSomething() {  
    }  
    int fibonacci(int n) {  
        return doSomething() + fibonacci(n - 1);  
    }  
}
```

Interface not declared

Can't multiply strings

Wrong type

Variable not declared

Can't redefine functions

Can't add void

No main function

Semantic analysis...

- Ensure that the program has a **well-defined meaning**.
- Verify properties of the program that aren't caught during the earlier phases:
 - *Variables are declared before they're used.*
 - *Expressions have the right types.*
 - *Classes don't inherit from nonexistent base classes*
 - *Type consistence;*
 - *Inheritance relationship is correct;*
 - *A class is defined only once;*
 - *A method in a class is defined only once;*
 - *Reserved identifiers are not misused;*
 - ...
- Once we finish semantic analysis, we know that the user's input program is **legal**.

What Does Semantic Analysis Involve?

- Semantic analysis typically involves:
 - **Type checking** – Data types are used in a manner that is consistent with their definition (i. e., only with compatible data types, only with operations that are defined for them, etc.)
 - **Label Checking** – Labels references in a program must exist
 - **Flow control checks** – control structures must be used in their proper fashion (e.g. no breaks outside a loop or switch statement, etc.)
 - **Uniqueness checks** - Certain names must be unique, Many languages require variable declarations
 - **Logical checks** - Program is syntactically and semantically correct, but does not do the “correct” thing

What Does Semantic Analysis Involve?...

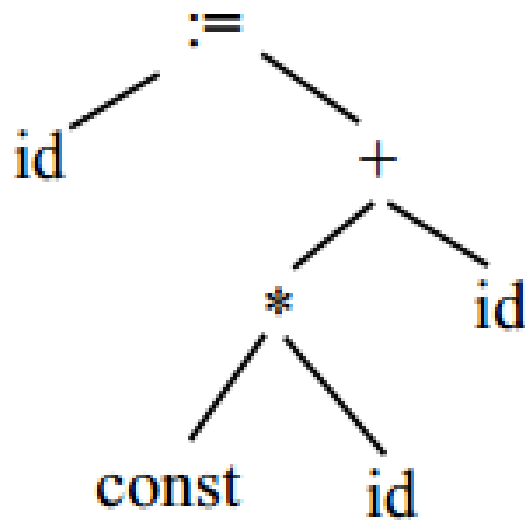
- Semantic analysis also gathers useful information about program for later phases:
 - E.g. count how many variables are in scope at each point.
- Why can't we just do this during parsing?
- Limitations of CFGs
 - How would you prevent duplicate class definitions?
 - How would you differentiate variables of one type from variables of another type?
 - How would you ensure classes implement all interface methods?
- Actually, **semantic analysis** is not a separate module within a compiler.
 - It is usually a collection of procedures called at appropriate times by the parser as the grammar requires it.

What Does Semantic Analysis Produce?

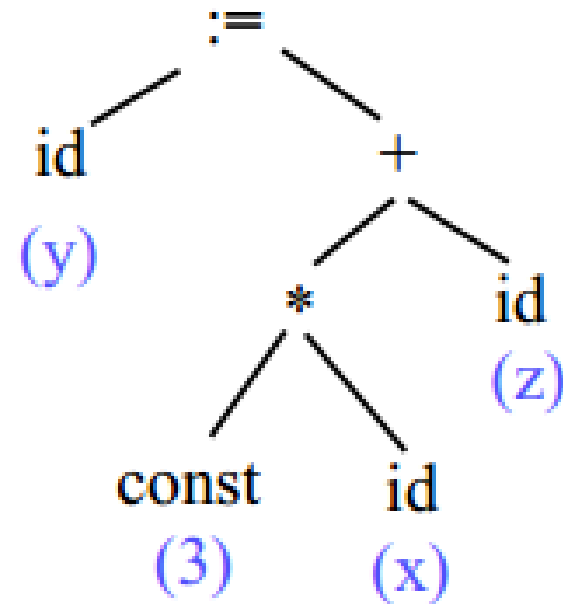
- Part of semantic analysis is producing some sort of representation of the program, either **object code** or an **intermediate representation** of the program.
- **One-pass compilers** will generate **object code** without using an intermediate representation;
 - code generation is part of the semantic actions performed during parsing.
- Other compilers will produce an **intermediate representation** during semantic analysis;
 - most often it will be an **annotated abstract syntax tree** or **quadruples**.
- The input for Semantic Analysis is *Abstract Syntax tree*

What Does Semantic Analysis Produce?...

- *Annotated Abstract Syntax tree* is parse-tree that shows the values of the attributes at each node.



parse tree



annotated parse tree

Syntax-Directed Translation (SDT)

- As for Lexical and Syntax analysis, also for Semantic Analysis we need both a *Representation Formalism* and an *Implementation Mechanism*
- One of *representation formalism* of semantic Analysis is *Syntax Directed Translations (SDT)*
- The Principle of SDT states that the meaning of an input sentence is related to its syntactic structure, i.e. to its Parse-Tree.
- By **SDTs** we indicate those formalisms for specifying translations for PL constructs guided by CFGs.
 - We associate **Attributes** to the **grammar symbols** representing the language constructs.
 - **Values for attributes** are computed by **Semantic Rules** associated with **grammar productions**.

Syntax-Directed Translation (SDT)...

- It is the combination of production and semantic rule

STD=Production+Semantic rule $A \rightarrow BC \quad \{f(B.V, C.V)\}$

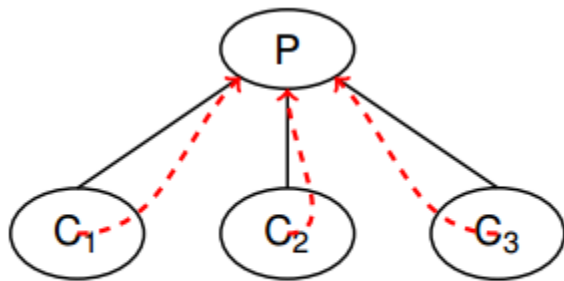
- Evaluation of Semantic Rules may:
 - Generate Code;
 - Insert information into the Symbol Table;
 - Perform Semantic Check;
 - Issue error messages ; etc.
- There are two notations for attaching semantic rules:
 - **Syntax Directed Definitions**
 - High-level specification hiding many implementation details (also called **Attribute Grammars**).
 - **Translation Schemes**
 - More implementation oriented: Indicate the order in which semantic rules are to be evaluated.

Syntax Directed Definitions(SDD)

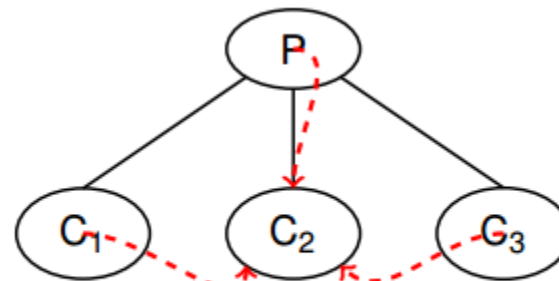
- SDD are a generalization of CFGs in which:
 - **Grammar symbols** have an associated set of **Attributes**;
 - **Productions** are associated with **Semantic Rules** for computing the values of attributes.
- Such formalism generates **Annotated Parse-Trees** where each node is of type record with a field for each attribute
 - e.g., **X.a** indicates the attribute **a** of the grammar symbol **X**.
- Attributes can represent anything we need: a string, a type, a number, a memory location , ... and they are two types

Two Types of Attributes

- **Synthesized attributes:** attribute values are computed from some attribute values of its children nodes
 - $P.\text{synthesized_attr} = f(C1.\text{attr}, C2.\text{attr}, C3.\text{attr})$
- **Inherited attributes:** attribute values are computed from attributes of the siblings and parent of the node
 - $C2.\text{inherited_attr} = f(P1.\text{attr}, C1.\text{attr}, C3.\text{attr})$



Synthesized attribute



Inherited attribute

S and L attributes

- S attribute only use Synthesized attribute where as L attribute used both synthesized and inherited attribute. The inherited attribute only inherit values from either parent or left siblings only
- Example
 $A \rightarrow BCD \{B.V=A.V, C.V=B.V, D.V=C.V\}$ but impossible to describe like $\{C.V=D.V\}$
- In S attribute Semantic actions are placed at the end of the production $A \rightarrow BC \{.....\}$ where as in L attribute the semantic are placed at anywhere on RHS like $A \rightarrow \{.....\}BC \mid B\{.....\}C \mid BC\{...\}$
- In S attribute attributes are evaluated during bottom up parsing where as in L attribute by traversing parse tree depth first, left to right

Syntax-Directed Definitions (SDD): Example

Production

$L \rightarrow E \text{ return}$

$E \rightarrow E_1 + T$

$E \rightarrow T$

$T \rightarrow T_1 * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow \text{digit}$

Semantic Rules

$\{ \text{print}(E.\text{val}) \}$

$\{ E.\text{val} = E_1.\text{val} + T.\text{val} \}$

$\{ E.\text{val} = T.\text{val} \}$

$\{ T.\text{val} = T_1.\text{val} * F.\text{val} \}$

$\{ T.\text{val} = F.\text{val} \}$

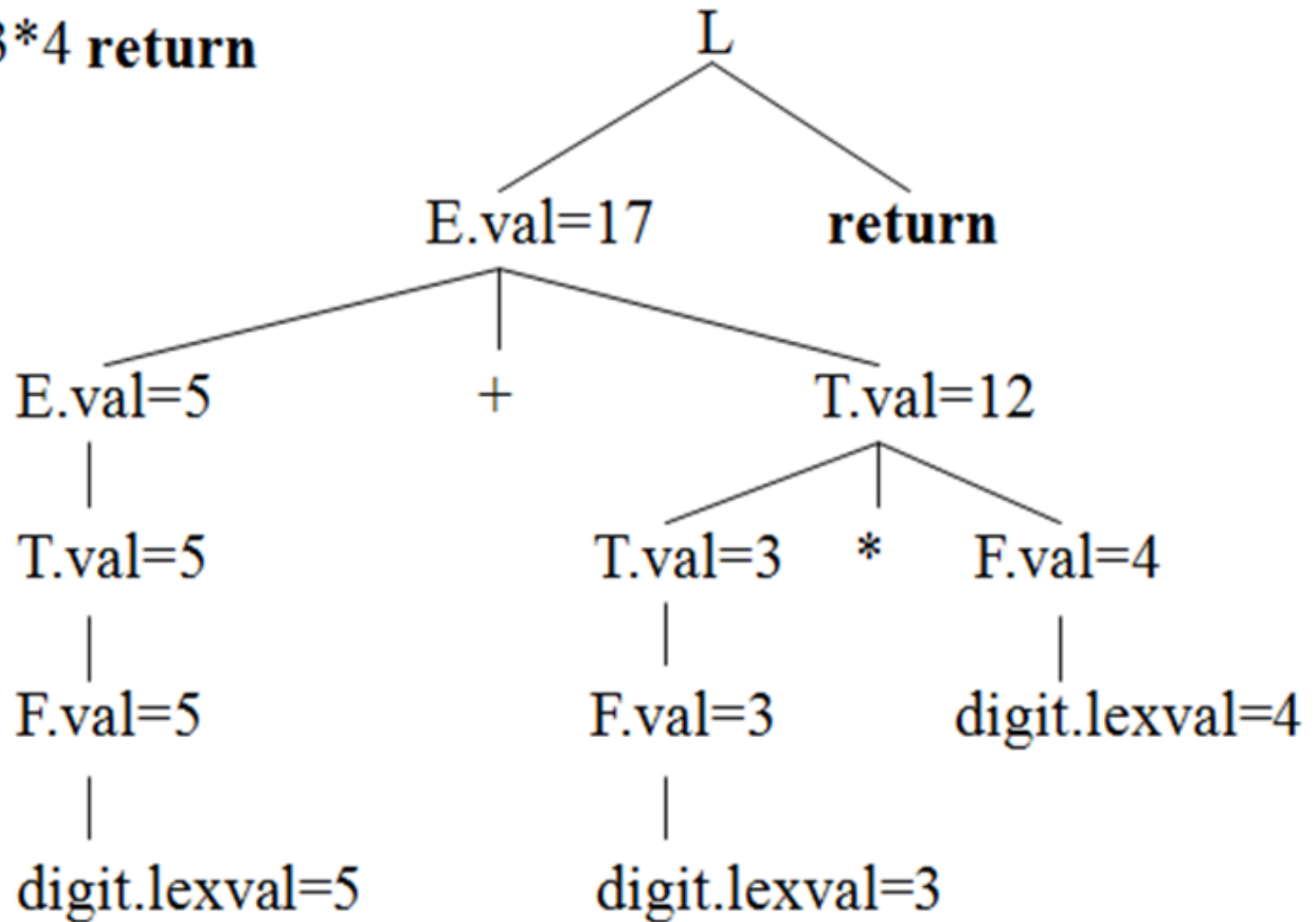
$\{ F.\text{val} = E.\text{val} \}$

$\{ F.\text{val} = \text{digit}.\text{lexval} \}$

- Symbols E, T, and F are associated with a synthesized attribute **val**.
- The token **digit** has a synthesized attribute **lexval** (it is assumed that it is evaluated by the lexical analyzer).

Annotated Parse Tree Example for $3*5+4$ return

Input: $5+3*4$ **return**



S-Attributed Definitions

- **An S-Attributed Definition** is a SDD that uses only Synthesized attributes
- **Evaluation Order:**
 - Semantic rules in a S-Attributed Definition can be evaluated by a *bottom-up*, or *post-order* traversal of the parse-tree.
- An S-attributed SDD can be implemented naturally in conjunction with an LR parser.
- **Example** – The arithmetic grammar at slide number 16 is an example of an S-Attributed.

Inherited Attributes

- **Inherited Attributes** are useful for expressing the dependence of a construct on the context in which it appears.
- It is always possible to rewrite a syntax directed definition to use only synthesized attributes, but it is often more natural to use both synthesized and inherited attributes.
- **Evaluation Order:** Inherited attributes can be evaluated by a **pre-order** traversal of the parse-tree, but
 - Unlike synthesized attributes, the order in which the inherited attributes of the children are computed is important.
 - Inherited attributes of the children can depend from both left and right siblings!

Inherited Attributes: Example

Production

$D \rightarrow T L$

$T \rightarrow \text{int}$

$T \rightarrow \text{real}$

$L \rightarrow L_1, \text{id}$

$L \rightarrow \text{id}$

Semantic Rules

$\{ L.in = T.type \}$

$\{ T.type = \text{integer} \}$

$\{ T.type = \text{real} \}$

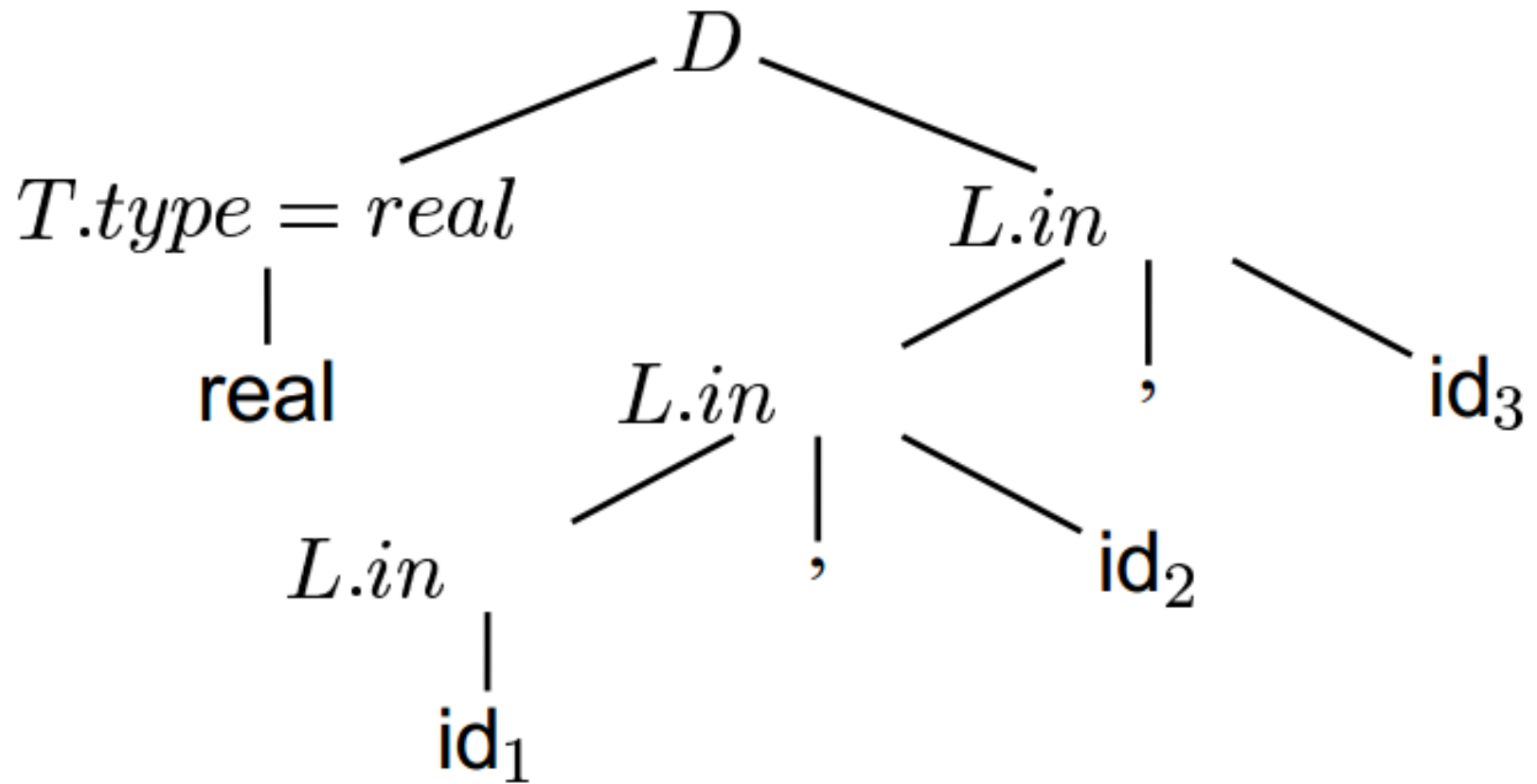
$\{ L_1.in = L.in, \text{addtype}(\text{id.entry}, L.in) \}$

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

- Symbol T is associated with a synthesized attribute *type*.
- Symbol L is associated with an inherited attribute *in*.
- The production $L \rightarrow L_1, \text{id}$ distinguishes the two occurrences of L .

Annotated Parse Tree Example

- The annotated parse-tree for the input **real id1, id2, id3** is

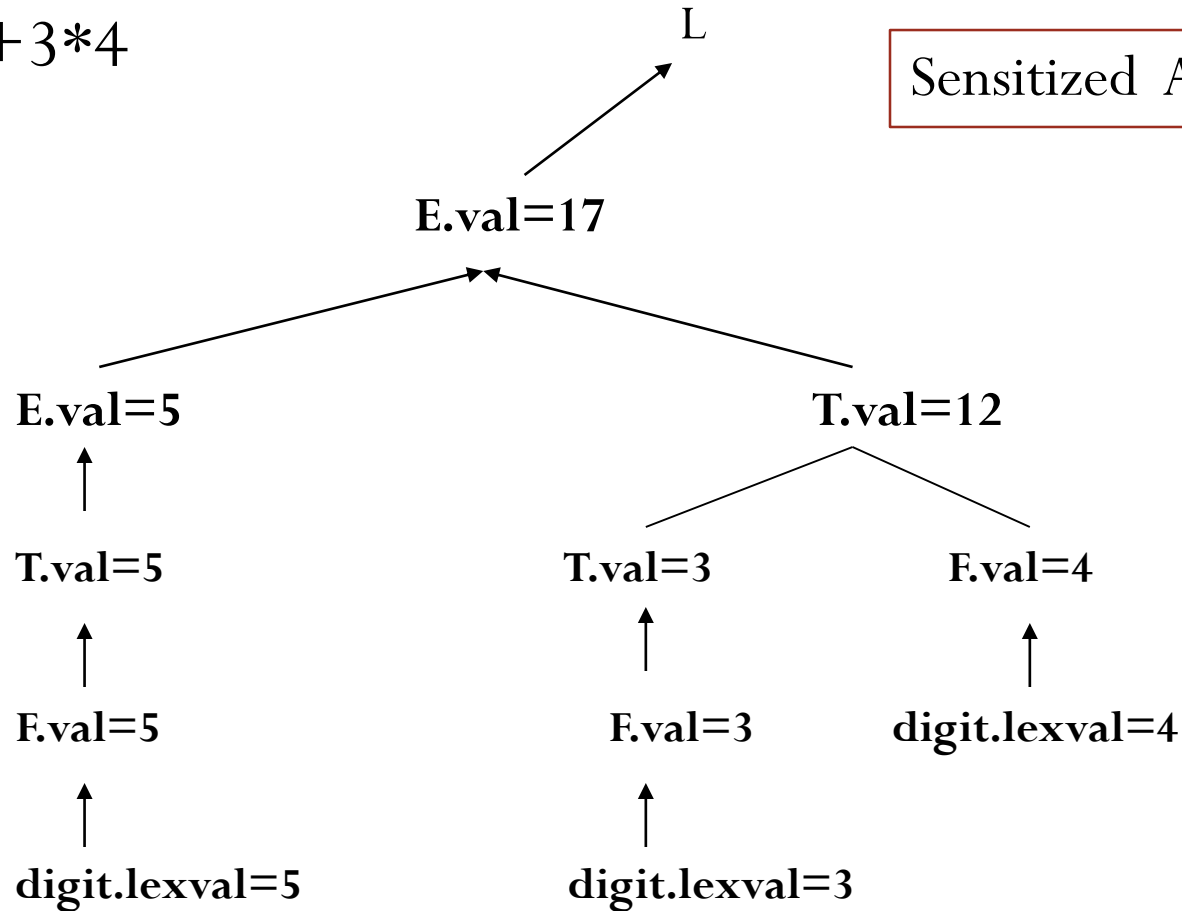


Dependency Graph

- Implementing a SDD consists primarily in finding an order for the evaluation of attributes
- **Dependency Graphs**(one of implementation mechanisms of semantic analysis) are the most general procedure to evaluate syntax directed translations with both synthesized and inherited attributes.
- A Dependency Graph shows the interdependencies among the attributes of the various nodes of a parse-tree.
 - There is a node for each attribute;
 - If attribute **b** depends on an attribute **c** there is a link from the node for **c** to the node for **b** (**b** \leftarrow **c**)
- If an attribute **b** depends from another attribute **c** then we need to fire the semantic rule **c** for first and then the semantic rule for **b**

Dependency Graph...

Input: $5 + 3 * 4$

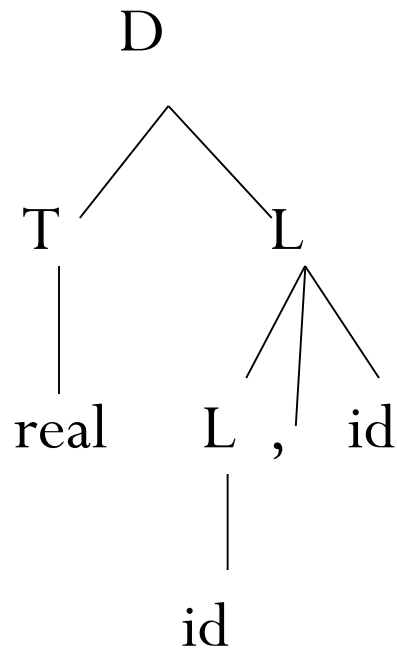


A *dependency graph* suggests possible *evaluation orders* for an annotated parse-tree.

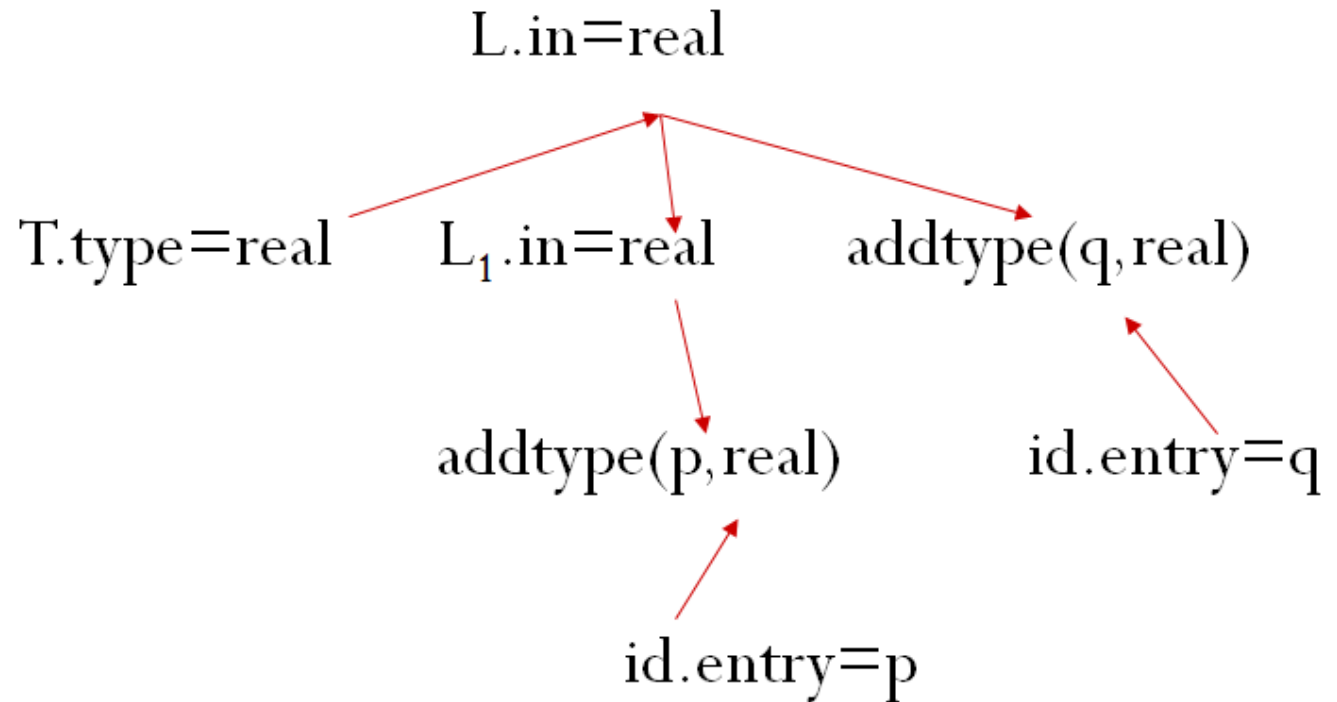
Dependency Graph...

Input: real p, q

Inherited Attributes



parse tree



dependency graph

Implementing Attribute Evaluation: General Remark

- Attributes can be evaluated by building a *dependency graph* at compile-time and then finding a topological sort
- Disadvantages
 - fails if the dependency graph has a cycle: We need a test for non-circularity;
 - is time consuming due to the construction of the dependency graph.
- **Alternative Approach** - design the SDD in such away that attributes can be evaluated with a fixed order (method followed by many compilers).
- Formalisms for which an attribute evaluation order can be fixed at compiler construction time are known as *Strongly Non-Circular Syntax Directed Definitions*.
 - S-Attributed and L-Attributed Definitions examples.

Evaluation of S-Attributed Definitions

- Synthesized Attributes can be evaluated by a **bottom-up** parser as the input is being analyzed avoiding the construction of a dependency graph.
- The parser keeps the values of the synthesized attributes in its stack.
- When ever a reduction $A \rightarrow \alpha$ is made, the attribute for A is computed from the attributes of α which appear on the stack.
- Thus, a translator for an **S-Attributed Definition** can be implemented by extending the stack of an **LR-Parser**

L-Attributed Definitions

- **L-Attributed Definitions** contain both synthesized and inherited attributes but do not need to build a dependency graph to evaluate them
- This means that they can also be evaluated during the parsing
- A syntax-directed definition is **L-attributed** if each inherited attribute of X_j , where $1 \leq j \leq n$, 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, \dots, X_{j-1} to the left of X_j in the production and
 2. the inherited attribute of A
- **Note that:** Every S-attributed definition is L-attributed, the restrictions only apply to inherited attribute

L-Attributed Definitions...

- **L-Attributed Definitions** are a class of syntax directed definitions whose attributes can always be evaluated by single traversal of the parse-tree of the inherited attributes (not to synthesized attributes)
- The following procedure evaluate L-Attributed Definitions by mixing PostOrder (synthesized) and PreOrder (inherited) traversal

Algorithm L-Eval(n : Node). *Input:* Parse-Tree node from an L-Attribute Definition. *Output:* Attribute evaluation.

Begin

 For each child m of n , from left-to-right Do Begin;

 evaluate inherited attributes of m ;

 L-Eval(m)

 End;

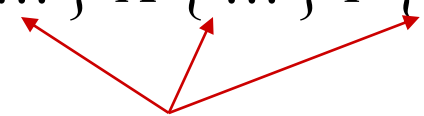
 evaluate synthesized attributes of n

End.

Translation Schemes

- In a syntax-directed definition, we do not say anything about the evaluation times of the semantic rules
 - when the semantic rules associated with a production should be evaluated?
- A **translation scheme** is a context-free grammar in which:
 - attributes are associated with the grammar symbols and
 - semantic actions enclosed between braces $\{\}$ are inserted within the right sides of productions.

Ex: $A \rightarrow \{ \dots \} X \{ \dots \} Y \{ \dots \}$



Semantic Actions

- Translation schemes indicate the order in which semantic rules and attributes are to be evaluated

Translation Schemes for S-attributed Definitions

- If our syntax-directed definition is S-attributed, the construction of the corresponding translation scheme will be simple.
- Each associated semantic rule in a S-attributed syntax-directed definition will be inserted as a semantic action into the end of the right side of the associated production.

Production

$E \rightarrow E1 + T$

Semantic Rule

$E.val = E1.val + T.val \rightarrow$ a production
of a syntax directed definition

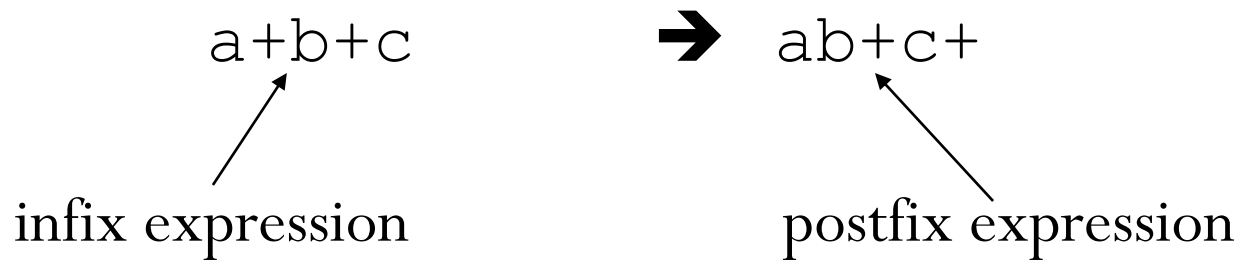


$E \rightarrow E1 + T$

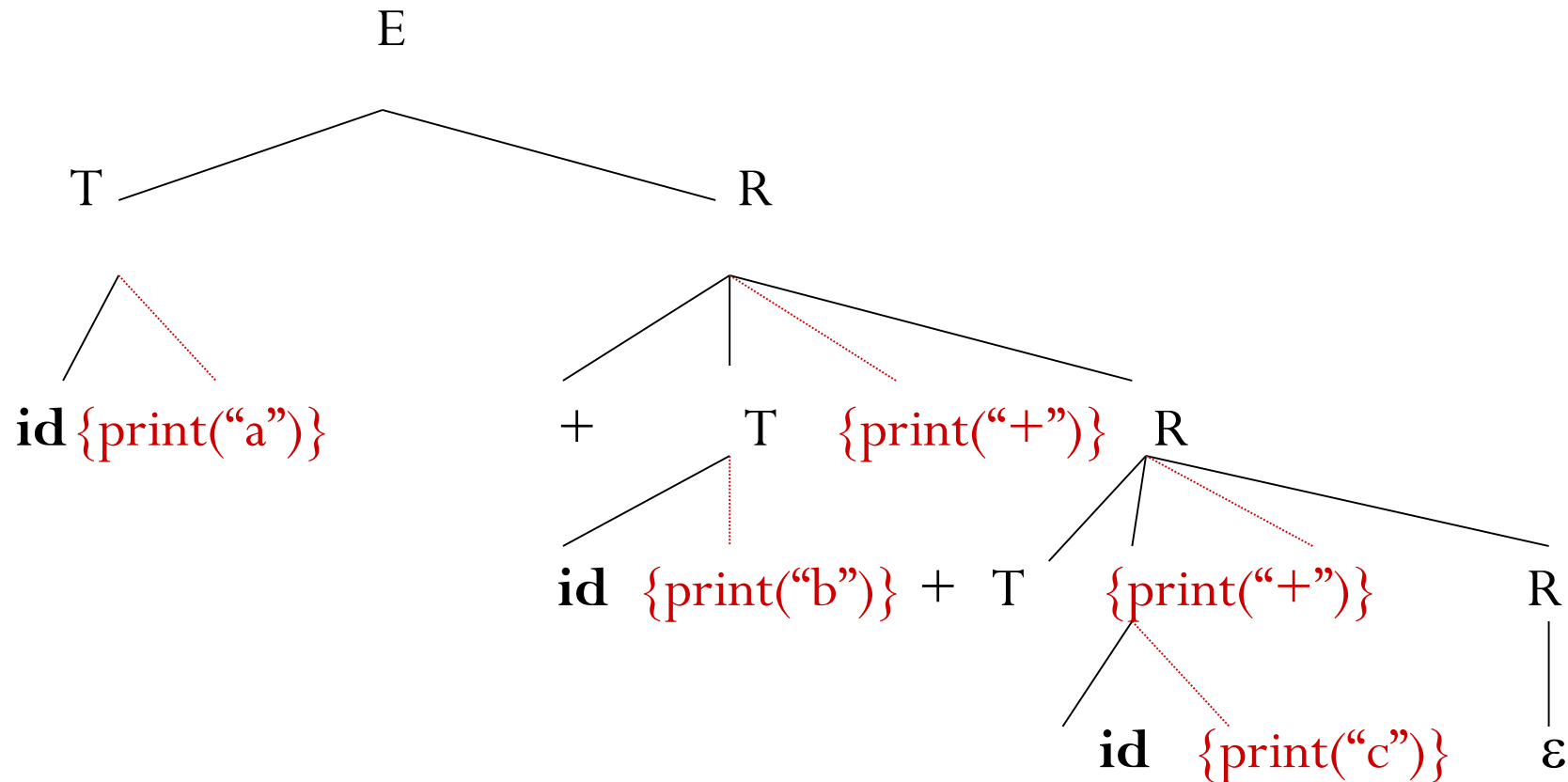
$\{ E.val = E1.val + T.val \} \rightarrow$ the
production of the corresponding translation scheme

A Translation Scheme Example

- A simple translation scheme that converts infix expressions to the corresponding postfix expressions.

$$E \rightarrow T R$$
$$R \rightarrow + T \{ \text{print}(\text{"+"}) \} R_1$$
$$R \rightarrow \varepsilon$$
$$T \rightarrow \mathbf{id} \{ \text{print}(\mathbf{id.name}) \}$$


A Translation Scheme Example (cont.)



The **depth first traversal** of the parse tree (executing the semantic actions in that order) will produce the **postfix representation** of the infix expression.

Inherited Attributes in Translation Schemes

- If a translation scheme has to contain both synthesized and inherited attributes, we have to observe the following rules:
 - An inherited attribute of a symbol on the RHS of a production must be computed in a semantic action before that symbol.
 - A semantic action must not refer to a synthesized attribute of a symbol to the right of that semantic action.
 - A synthesized attribute for the non-terminal on the left can only be computed after all attributes it references have been computed
- With a L-attributed SDD, it is always possible to construct a corresponding translation scheme which satisfies these three conditions

Designing translation schemes

$D ::= T L$	$L.in := T.type$
$T ::= \mathbf{int}$	$T.Type := \text{integer}$
$T ::= \mathbf{real}$	$T.type := \text{real}$
$L ::= \mathbf{id}, L1$	$L1.in := L.in; \text{Addtype}(\text{id.entry}, L.in)$
$L ::= \mathbf{id}$	$\text{Addtype}(\text{id.entry}, L.in)$

- Every attribute value must be available when referenced
 - S-attribute of left-hand symbol computed at end of production
 - L-attribute of right-hand symbol computed before the symbol
 - S-attribute of right-hand symbol referenced after the symbol

```
D ::= T { L.in := T.type } L
T ::= int { T.Type := integer }
T ::= real { T.type := real }
L ::= id , { Addtype(id.entry, L.in) } { L1.in := L.in } L1
L ::= id { Addtype(id.entry, L.in) }
```

Top-down translation

```
void parseD()
  { Type t = parseT(); }
  parseL(t);
}

Type parseT
  { switch (currentToken()) {
    case INT: return TYPE_INT;
    case REAL: return TYPE_REAL;
  }
}

void parseL(Type in)
  {
    SymEntry e = parseID();
    AddType(e, in);
    if (currentToken() == COMMA) {
      parseTerminal(COMMA);
      parseL(in)
    }
  }
}
```

Reading assignment

Type Checking