# Principles of Compiler Design

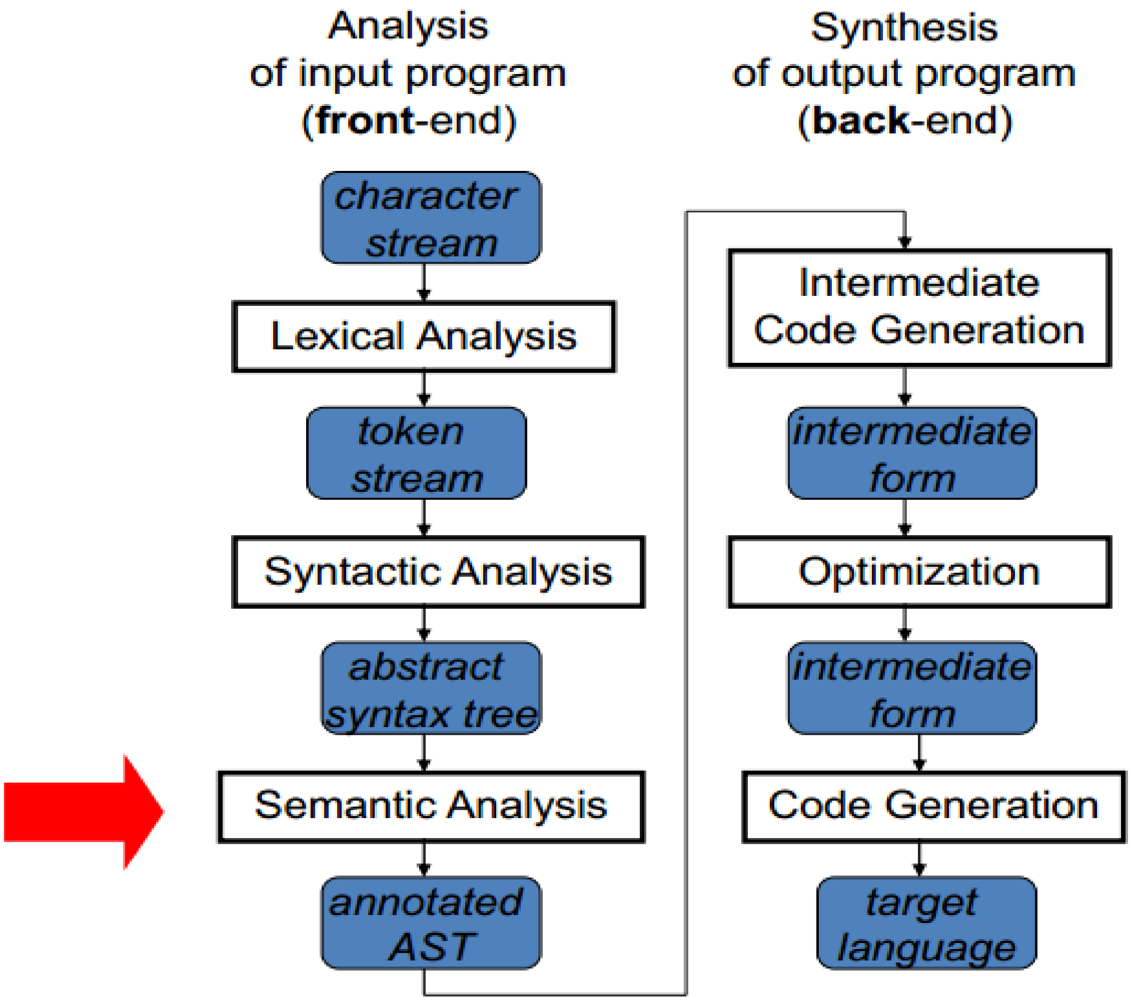
## Chapter 4

## Syntax Directed Translation

# Outline

- Syntax-Directed Translation
- Syntax Directed Definitions
- Attribute definitions
- Construction of syntax trees
- Bottom-up Evaluation of S-attributed Definitions
- L-attributed definitions

# Compiler Phases






3

# *Introduction to semantic analysis*

- Semantic Analysis phase is the *third phase of the compiler* .
- Semantic Analysis *checks the source program* for ***semantic errors.***
- It uses the ***hierarchical structure determined*** by the syntax analysis phase ***to identify*** the operators and operands of expressions and statement.
- Semantic analysis performs ***type checking***, i.e., it checks that, whether each operator has operands that are permitted by the source language specification.

Example: If a real numbers is used to index an array i.e., a[1.5] then the compiler will *report an error* . This error is handled during ***semantic analysis***.

# ***Semantic Errors***

- ✓The primary source of semantic error are *undeclared names* and *type incompatibilities*.
- ✓Semantic errors can be ***detected*** both at compile *time* and *at run time*.
- ✓The ***errors*** will be declaration or scope of variables.

Example: ***Undeclared*** or ***multiply-declared*** identifiers.

Example: Type incompatibilities between operators and operands and between formal and actual parameters are another common source of semantic errors that can be **detected** at *compile time*.

# Syntax-Directed Translation

- **Grammar symbols** are associated with **attributes** to *associate information* with the programming language constructs that they represent.
- Values of these ***attributes are evaluated*** by the **semantic rules** associated with the production rules.
- Evaluation of these semantic rules:
  - may generate intermediate codes
  - may put information into the symbol table
  - may perform type checking
  - may issue error messages
  - may perform some other activities
  - in fact, they may perform almost any activities.
- An attribute may hold almost any thing.
  - a string, a number, a memory location, a complex record.
- SDT = Grammar + Semantic Rules

# Syntax-Directed Definitions and Translation Schemes

- When we associate **semantic rules** with productions, we use two notations:
  - **Syntax-Directed Definitions**
  - **Translation Schemes**
- **Syntax-Directed Definitions:**
  - give high-level specifications for translations
  - hide many implementation details such as order of evaluation of semantic actions.
  - We associate a production rule with a **set of semantic actions**, and we do not say when they **will be evaluated**.
- **Translation Schemes:**
  - indicate the order of evaluation of semantic actions associated with a **production rule**.
  - In other words, **translation schemes** give a little bit information about implementation details.

# Syntax-Directed Definitions

- A syntax-directed definition is a generalization of a context-free grammar in which:
  - Each grammar symbol is associated with a set of attributes.
  - This set of attributes for a grammar symbol is partitioned into ***two*** subsets called **synthesized** and **inherited** attributes of that grammar symbol.
  - Each production rule is associated with a set of **semantic rules.**
- ***Semantic rules*** set up dependencies between attributes which can be represented by a ***dependency graph***.
- This ***dependency graph*** determines the evaluation order of these semantic rules.
- Evaluation of a semantic rule defines the value of an attribute. But a **semantic rule** may also have some ***side effects*** such as ***printing*** a value.

# Annotated Parse Tree

- A parse tree showing the values of attributes at each node is called an **annotated parse tree**.
- The process of computing the attributes values at the nodes is called **annotating** (or **decorating**) of the parse tree.
- Of course, the order of these computations depends on the **dependency graph induced** by the semantic rules.

# Syntax-Directed Definition

- In a ***syntax-directed definition***, each production $A \rightarrow \alpha$ is associated with a set of semantic rules of the form:

  $b = f(c_1, c_2, \ldots, c_n)$ where $f$ is a function,

  and $b$ can be one of the followings:

  - $b$ is a **synthesized attribute** of A and $c_1, c_2, \ldots, c_n$ are attributes of the grammar symbols in the production ( $A \rightarrow \alpha$ ). i.e. the value of a synthesized attribute is computed from the values of attributes at the children of that node in the parse tree and itself

    OR

  - $b$ is an **inherited attribute** one of the grammar symbols in $\alpha$ (on the right side of the production), and $c_1, c_2, \ldots, c_n$ are attributes of the grammar symbols in the production ( $A \rightarrow \alpha$ ). i.e. The value of an **inherited attribute** is computed from the values of attributes at the siblings and/or parent of that node in the parse tree.

# Attribute Grammar

- So, a semantic rule $b=f(c_1,c_2,\ldots,c_n)$ indicates that the attribute b *depends on* attributes $c_1,c_2,\ldots,c_n$.
- In a **syntax-directed definition**, a semantic rule may just evaluate a value of an attribute or it may have some side effects such as printing values.

- An **attribute grammar** is a syntax-directed definition in which the functions in the semantic rules cannot have side effects (they can only evaluate values of attributes).

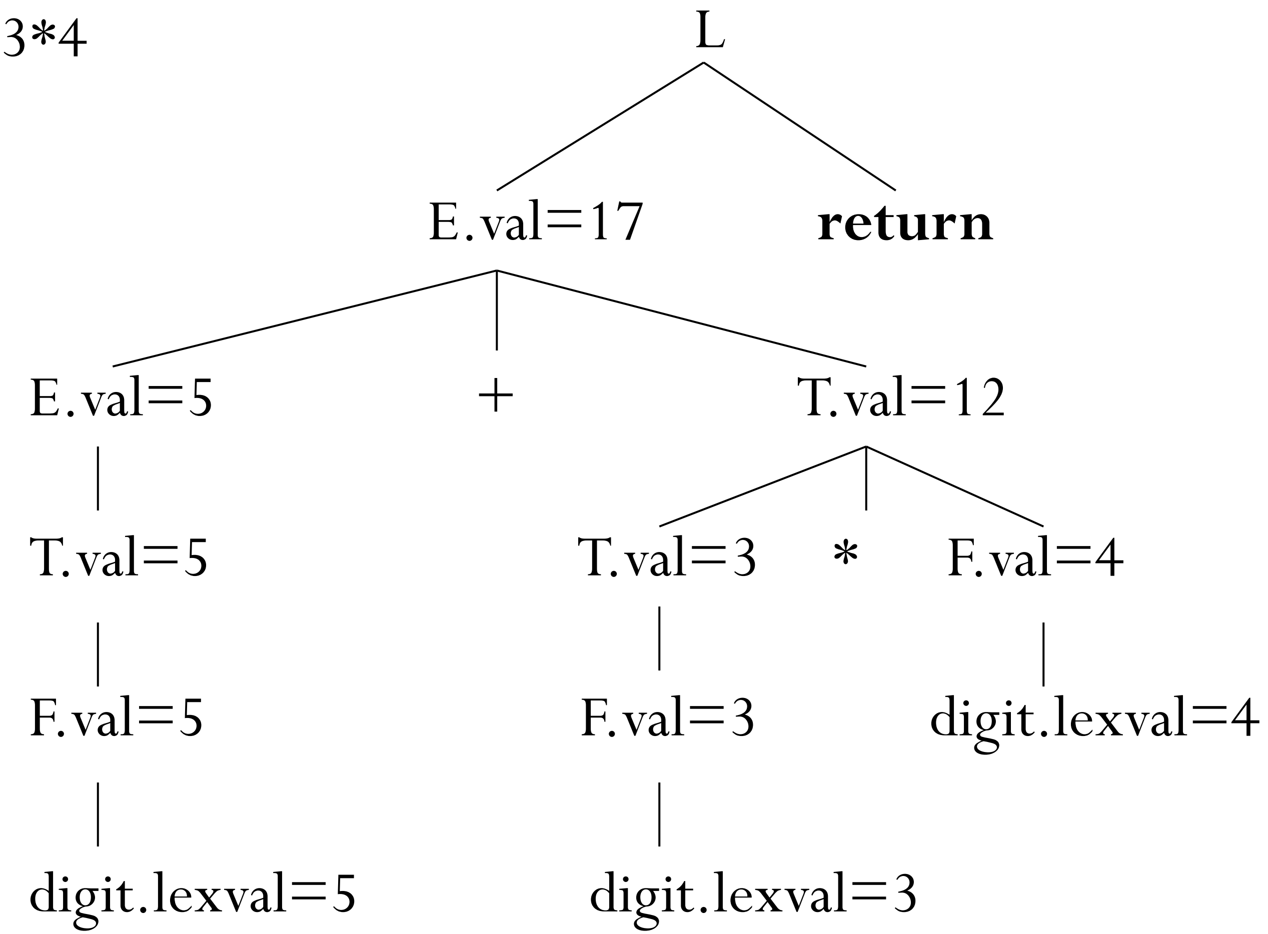
# Syntax-Directed Definition - Example

| **Production** | **Semantic Rules** |
|---|---|
| $L \rightarrow E$ **return** | print(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 |

- 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).

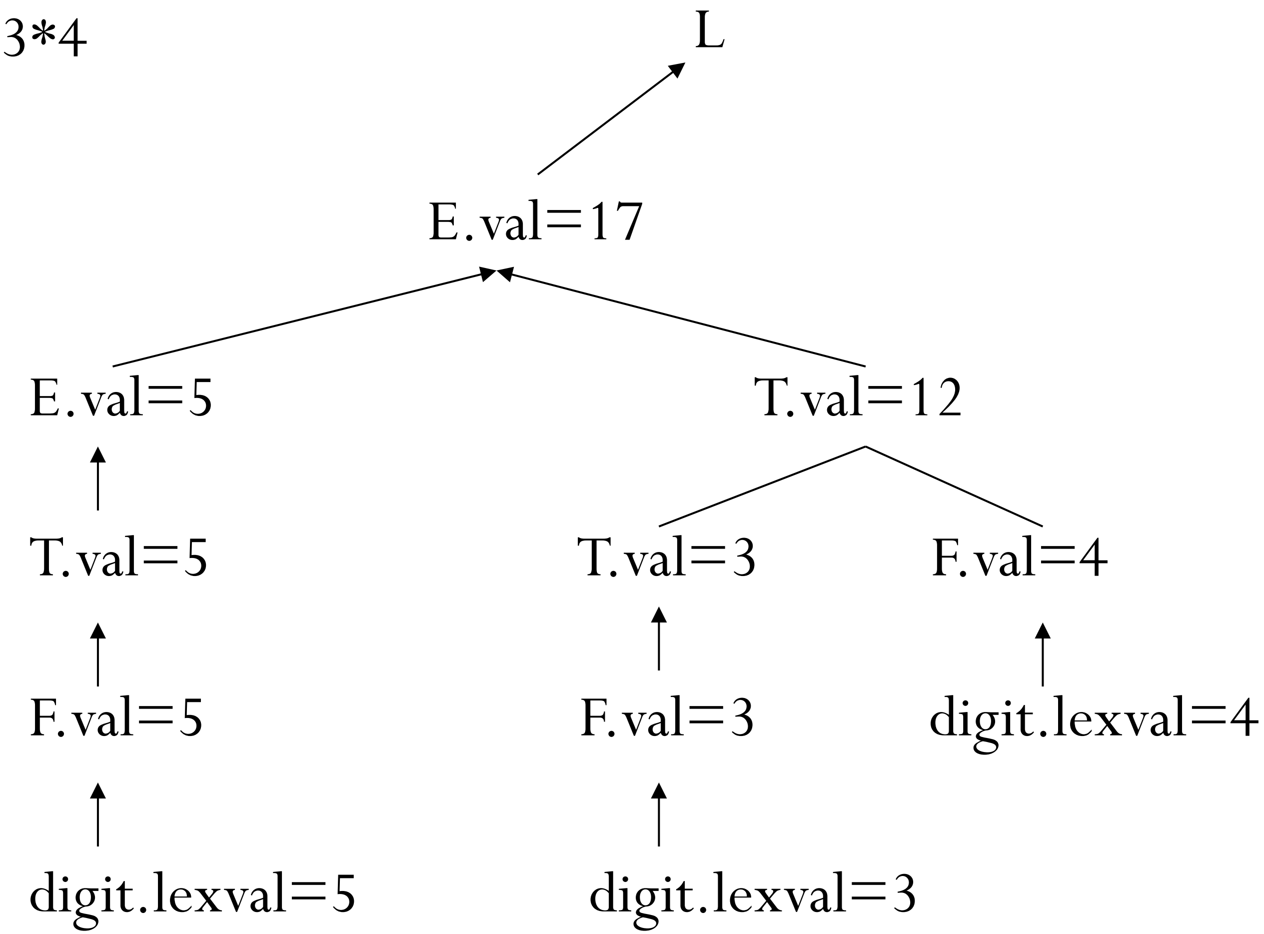
# Annotated Parse Tree - Example

Input: 5+3*4



# Dependency Graph - Example

Input: 5+3*4



# Syntax-Directed Definition – Example2

| Production | Semantic Rules |
| --- | --- |
| $E \rightarrow E_1 + T$ | E.loc=newtemp(), E.code = $E_1$.code \|\| T.code \|\| add $E_1$.loc,T.loc,E.loc |
| $E \rightarrow T$ | E.loc = T.loc, E.code=T.code |
| $T \rightarrow T_1 * F$ | T.loc=newtemp(), T.code = $T_1$.code \|\| F.code \|\| mult $T_1$.loc,F.loc,T.loc |
| $T \rightarrow F$ | T.loc = F.loc, T.code=F.code |
| $F \rightarrow ( E )$ | F.loc = E.loc, F.code=E.code |
| $F \rightarrow$ **id** | F.loc = **id**.name, F.code=“ ” |

- Symbols E, T, and F are associated with synthesized attributes *loc* and *code*.
- The token **id** has a synthesized attribute *name* (it is assumed that it is evaluated by the lexical analyzer).
- It is assumed that || is the string concatenation operator.

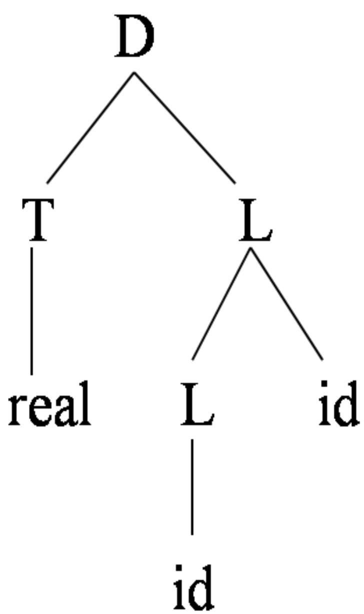
# Syntax-Directed Definition – Inherited Attributes

| **Production** | **Semantic Rules** |
|---|---|
| $D \rightarrow T\ L$ | $L.in = T.type$ |
| $T \rightarrow$ **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) |

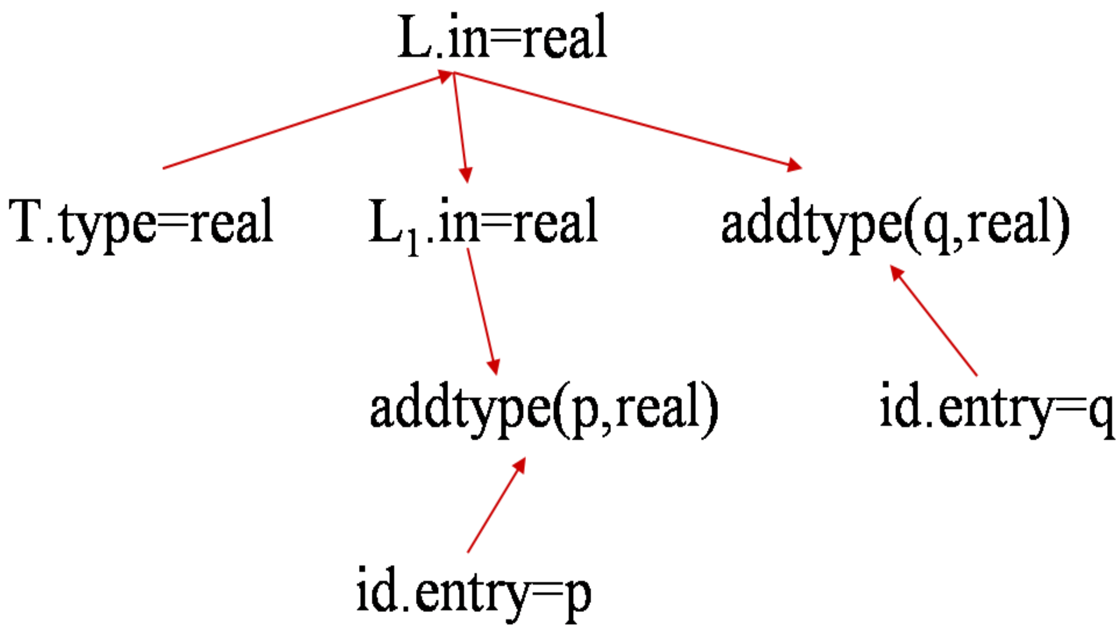
- Symbol T is associated with a synthesized attribute *type*.
- Symbol L is associated with an inherited attribute *in*.

# A Dependency Graph – Inherited Attributes

Input: `real p q`




*parse tree*




*dependency graph*

# S-Attributed Definitions

- Syntax-directed definitions are used to specify syntax-directed translations.
- To create a translator for an arbitrary syntax-directed definition can be difficult.
- We would like to evaluate the semantic rules during parsing (i.e. in a single pass, we will parse and we will also evaluate semantic rules during the parsing).
- We will look at two sub-classes of the syntax-directed definitions:
  - **S-Attributed Definitions**: only synthesized attributes used in the syntax-directed definitions.
  - **L-Attributed Definitions**: in addition to synthesized attributes, we may also use inherited attributes in a restricted fashion.
- To implement S-Attributed Definitions and L-Attributed Definitions are easy (we can evaluate semantic rules in a single pass during the parsing).
- Implementations of S-attributed Definitions are a little bit easier than implementations of L-Attributed Definitions

# L-Attributed Definitions

- S-Attributed Definitions can be efficiently implemented.
- We are looking for a larger (larger than S-Attributed Definitions) subset of syntax-directed definitions which can be efficiently evaluated.

  ➔ **L-Attributed Definitions**

- L-Attributed Definitions can always be evaluated by the depth first visit of the parse tree.
- This means that they can also be evaluated during the parsing.

# L-Attributed Definitions

- 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 $A \rightarrow X_1X_2 \ldots X_n$ depends only on:
  1. The attributes of the symbols $X_1, \ldots, X_{j-1}$ to the left of $X_j$ in the production and
  2. the inherited attribute of A

- Every S-attributed definition is L-attributed, the restrictions only apply to the inherited attributes (***not to synthesized attributes***).

# 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:* $A \longrightarrow \{ \ldots \}\ X\ \{ \ldots \}\ Y\ \{ \ldots \}$

Semantic Actions

# Translation Schemes

- When designing a translation scheme, some restrictions should be observed to ensure that an attribute value is available when a semantic action refers to that attribute.
- These restrictions (motivated by L-attributed definitions) ensure that a semantic action does not refer to an attribute that has not yet computed.
- In translation schemes, we use *semantic action* terminology instead of *semantic rule* terminology used in syntax-directed definitions.
- The position of the semantic action on the right side indicates when that semantic action will 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 | Semantic Rule | |
|---|---|---|
| $E \rightarrow E_1 + T$ | $E.val = E_1.val + T.val$ | ➔ a production of a syntax directed definition |

$$\Downarrow$$

$E \rightarrow E_1 + T \; \{ \; E.val = E_1.val + T.val \; \}$ ➔ the production of the corresponding translation scheme

# A Translation Scheme Example

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

$E \rightarrow T\ R$

$R \rightarrow +\ T$ { print("+") } $R_1$

$R \rightarrow \varepsilon$

$T \rightarrow$ **id** { print(**id**.name) }

`a+b+c` ➔ `ab+c+`

infix expression postfix expression

# Self-Review Questions

1. Define the following
    1. Syntax Directed Definition
    2. Dependency graph
    3. Annotated parse tree
    4. Synthesized Attribute
    5. Inherited Attribute

2. Compare and Contrast S-Attributed SDT and L-Attributed SDT

# Endof ch4…

Thank You

?