A Simple Syntax-Directed Translator

Course Code: CSC3220 Course Title: Compiler Design

Dept. of Computer Science
Faculty of Science and Technology

Lecturer No:	**4**	**Week No:**	**4**	**Semester:**	**Summer**
Lecturer:	*Md Masum Billah; billah.masumcu@aiub.edu*				

Lecture Outline

1. Quiz1
2. Learning Objectives
3. Context-free grammar
4. Derivation
5. Ambiguity
6. Associativity of operators
7. Precedence of operators
8. Books and References

Objective and Outcome

Objective:

- To explain the Context Free Grammar (CFG) with example.
- To demonstrate derivation or derivation tree from a CFG
- To elaborate ambiguity and ambiguous grammar.
- To explain associativity and precedence of operator

Outcome:

- After this lecture the student will able to demonstrate CFG
- Student will be capable of derivation from CFG
- Student will be able to differentiate if a grammar is ambiguous or not.
- After this lecture student will learn associativity and precedence of operator

Context-free grammar

- In this section, we introduce a notation — the **"context-free grammar,"** or **"grammar"** for short — that is used to specify the syntax of a language.
- A grammar naturally describes the hierarchical structure of most programming language constructs.
- For example, an if-else statement in Java can have the form

  if (expression) statement else statement

Definition

A grammar consists of:

- a set of variables (also called non terminals), one of which is designated the start variable; It is customary to use upper-case letters for variables;
- a set of terminals (from the alphabet); and
- a list of productions (also called rules).
- A designation of one of the non terminals as the *start* symbol.

Formal Definition

One can provide a formal definition of a context free grammar. It is a 4-tuple (V, Σ, S, P) where:

- V is a finite set of variables;
- Σ is a finite alphabet of terminals;
- S is the start variable; and
- P is the finite set of productions. Each production has the form $V \rightarrow (V \cup \Sigma)*$

Example: 0^n1^n Here is a grammar:

$S \rightarrow 0S1$

$S \rightarrow \varepsilon$

S is the only variable. The terminals are 0 and 1. There are two productions.

Derivation

- A grammar derives strings by beginning with the start symbol and repeatedly replacing a nonterminal by the body of a production for that nonterminal.
- The terminal strings that can be derived from the start symbol form the language defined by the grammar.

Derivation

$S \rightarrow 0S1$
$S \rightarrow \varepsilon$

The string 0011 is in the language generated.

The derivation is:
$S \Longrightarrow 0S1 \Longrightarrow 00S11 \Longrightarrow 0011$
For compactness, we write
$S \rightarrow 0S1 \mid \varepsilon$
where the vertical bar means or.

Consider the CFG
$S \rightarrow 0S1S \mid 1S0S \mid \varepsilon$
The string 011100 is generated:
$S \Longrightarrow 0S1S \Longrightarrow 01S \Longrightarrow 011S0S \Longrightarrow 0111S0S0S$
$\Longrightarrow 01110S0S \Longrightarrow 011100S \Longrightarrow 011100$

Derivation

- This CFG generates sentences as composed of noun- and verb-phrases:

S → NP VP
NP → the N
VP → V NP
V → sings | eats
N → cat | song | canary

This generates “the canary sings the song”, but also “the song eats the cat”.
This CFG generates all “legal” sentences, not just meaningful ones.

Parse Trees and Ambiguity

- A parse tree pictorially shows how the start symbol of a grammar derives a string in the language. If nonterminal A has a production A -> XYZ, then a parse tree may have an interior node labeled A with three children labeled X, Y, and Z, from left to right.

- A grammar can have more than one parse tree generating a given string of terminals. Such a grammar is said to be ambiguous.

Derivation

string ->*string* + *string* | *string* - *string* | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

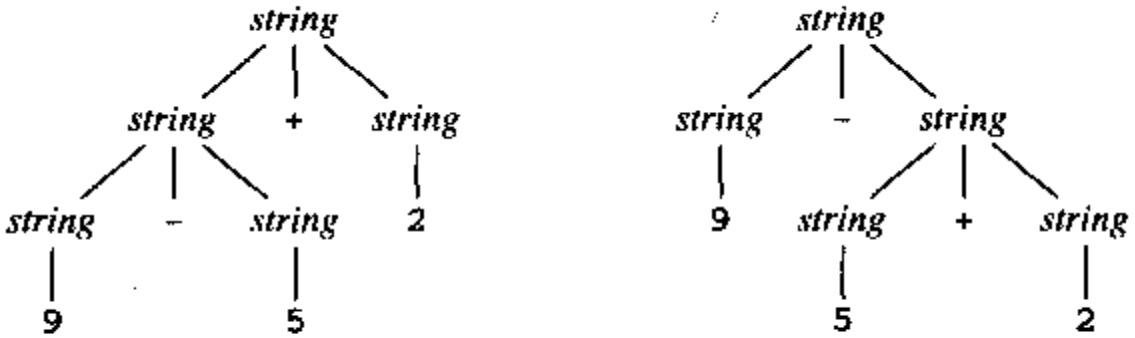
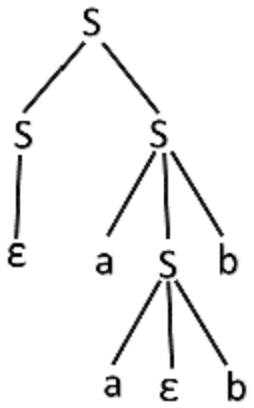


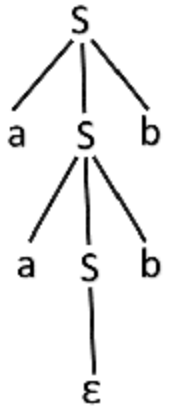
Fig. 2.3. Two parse trees for 9–5+2.

Check if the given grammar G is ambiguous or not.

S → aSb | SS
S → ε

For the string "aabb" the above grammar can generate two parse trees





Since there are two parse trees for a single string "aabb", the grammar G is ambiguous.

Associativity of operators

left-associative

- By convention, 9+5+2 is equivalent to (9+5)+2 and 9 - 5 - 2 is equivalent to (9 - 5) - 2 .
- When an operand like 5 has operators to its left and right, conventions are needed for deciding which operator applies to that operand.
- We say that the operator + associates to the left, because an operand with plus signs on both sides of it belongs to the operator to its left. In most programming languages the four arithmetic operators, addition, subtraction, multiplication, and division are left-associative.

Right-associative

- Some common operators such as exponentiation are right-associative.
- As another example, the assignment operator = in C and its descendants is right associative; that is, the expression a=b=c is treated in the same way as the expression a=(b=c).
- Strings like a=b=c with a right-associative operator are generated by the following grammar:

list→ list -digit|letter|digit
digit→ 0|1|2|……9|0
right ->• letter = right | letter
letter -> a |

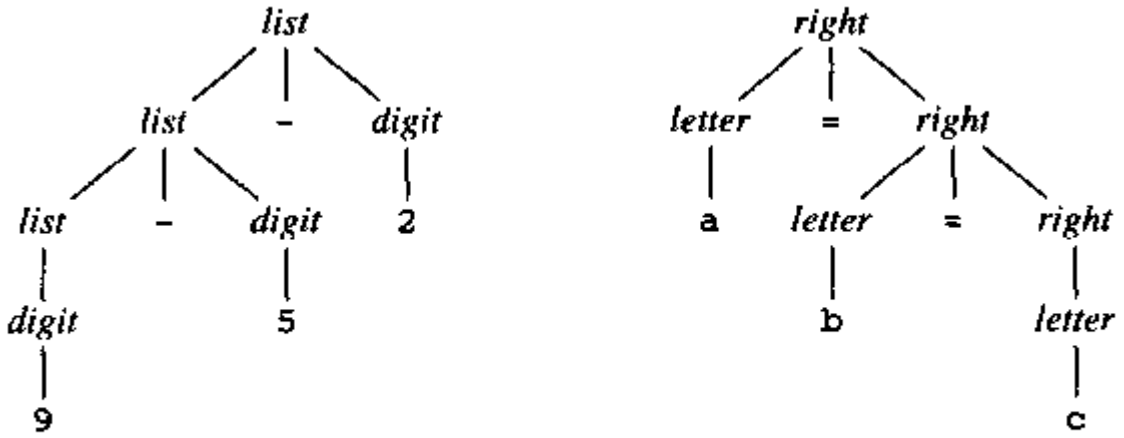


Fig. 2.4. Parse trees for left- and right-associative operators.

Precedence of operators

- Consider the expression 9+5*2.
- There are two possible interpretations of this expression: (9+5)*2 or 9+(5*2).
- The associativity rules for + and * apply to occurrences of the same operator, so they do not resolve this ambiguity.
- We say that * has higher precedence than + if * takes its operands before + does.
- In ordinary arithmetic, multiplication and division have higher precedence than addition and subtraction. Therefore, 5 is taken by * in 9+5*2.

Lecture References

1. Compilers-Principles, techniques and tools (2nd Edition) *V. Aho, Sethi and D. Ullman*

References/ Books

1. Compilers-Principles, techniques and tools (2nd Edition) *V. Aho, Sethi and D. Ullman*
2. Principles of Compiler Design (2nd Revised Edition 2009) *A. A. Puntambekar*
3. Basics of Compiler Design *Torben Mogensen*

Syntax Directed Translation

Course Code: CSC3220 Course Title: Compiler Design

Dept. of Computer Science
Faculty of Science and Technology

Lecturer No:	**5**	**Week No:**	**5**	**Semester:**	**Summer**
Lecturer:	*Md Masum Billah, billah.masumcu@aiub.edu*				

Lecture Outline

1. Syntax Directed Translation
2. Syntax Directed Definition
3. Synthesized Attribute
4. Inherited Attribute
5. Syntax Directed Translation Scheme
6. Class Exercises

Objectives and Outcomes

Objectives:

- Understand the Semantics of the language.
- Understand the evaluation process of input by the compiler.

Outcomes:

- Students should be able to understand the annotated parse tree.
- Students will analyze how to construct the input from infix to postfix by the compiler.

Syntax Directed Translation

Syntax-directed translation is done by attaching rules or program fragments to productions in a grammar. For example, consider an expression *expr* generated by the production

expr -» expr + term

Here, *expr* is the sum of the two subexpressions *expr* and *term.* (The subscript in *expr* is used only to distinguish the instance of *expr* in the production body from the head of the production). We can translate *expr* by exploiting its structure, as in the following pseudo-code:

translate *expr;*
translate *term;*
handle +;

Syntax Directed Translation

There are two notations for attaching semantic rules:

- Syntax Directed Definitions
- Syntax Directed Translation Schemes

Syntax Directed Definitions

Syntax Directed Definitions are a generalization of context-free grammars in which:

- Grammar symbols have an associated set of **Attributes**;
- Productions are associated with **Semantic Rules** for computing the values of attributes.

Important Note: Such formalism generates **Annotated Parse Tree** where each node of the tree is a record with a field for each attribute(e.g **expr.t** indicates the attribute **t** of the grammar symbol **expr)**

Syntax Directed Definitions

Here Semantic Rule is applied for one production of a context free grammar.

Production	Semantic Rule
$expr \rightarrow expr + term$	$expr.t := expr.t \,\|\|\, term.t \,\|\|\, \text{`+'}$

This production derives an expression containing a plus operator. The left operand of the plus operator is given by *expr* and the right operand by *term.* The semantic rule associated with this production constructs the value of attribute expr.t by concatenating the postfix forms expr.t and term.t of the left and right operands, respectively, and then appending the plus sign. This rule is a formalization of the definition of "postfix expression“. The symbol **||** in the semantic rule is the operator for string concatenation.

Syntax Directed Definitions

Example:

Production	Semantic Rule
$expr \rightarrow expr + term$	$expr.t := expr.t \,\|\|\, term.t \,\|\|\, '+'$
$expr \rightarrow expr - term$	$expr.t := expr.t \,\|\|\, term.t \,\|\|\, '-'$
$expr \rightarrow term$	$expr.t := term.t$
$term \rightarrow 0$	$term.t := '0'$
$term \rightarrow 1$	$term.t := '1'$
….	….
$term \rightarrow 9$	$term.t := '9'$

Syntax Directed Definitions

We applied semantic rules for each production of a context-free grammar. Now we will see how semantic rules are embedded in parse tree.

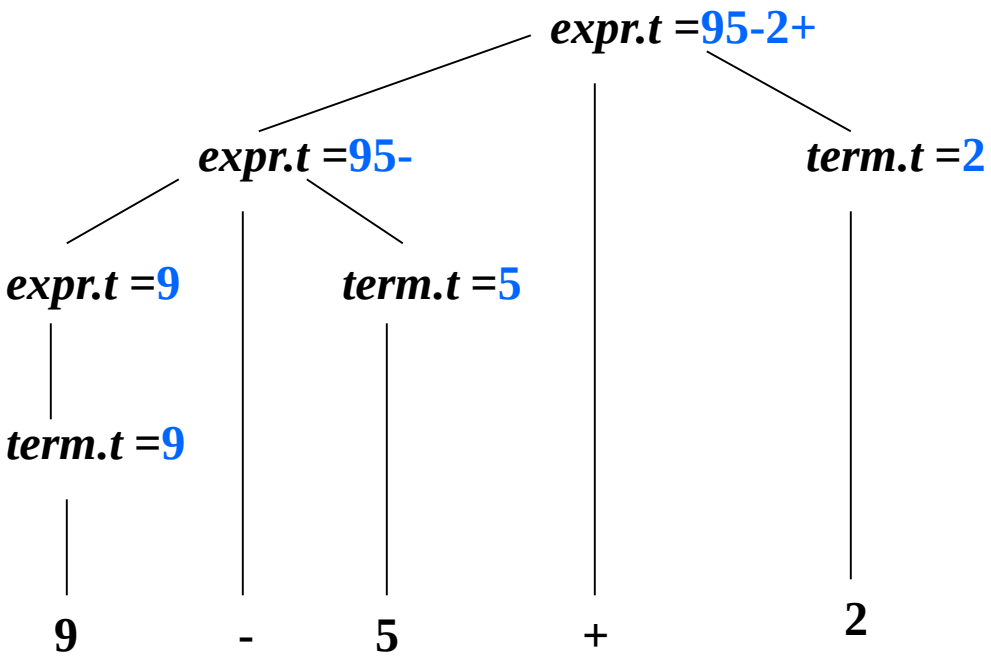


Fig: Annotated Parse Tree

Syntax Directed Definitions

- It starts at the root and recursively visits the children of each node in left-to-right order.
- The semantic rules at a given node are evaluated once all descendants of that node have been visited.
- A parse tree showing all the attribute values at each node is called annotated parse tree.

Syntax Directed Definitions

Attribute Grammars: Each grammar symbol has an associated set of attributes. An attribute can represent anything we choose:

- The value of an expression when literal constants are used
- The data type of a constant, variable, or expression
- The location (or offset) of a variable in memory
- The translated code of an expression, statement, or function

We distinguish between two kinds of attributes:

I. Synthesized Attributes.
II. Inherited Attributes.

Syntax Directed Definitions

Synthesized Attributes: An attribute is synthesized if the attribute value of parent is determined from attribute values of children in the parse tree.

Example:

Production	Semantic Rule
$expr \rightarrow expr + term$	$expr.t := expr.t \,\|\| \, term.t \,\|\| \, '+'$
$expr \rightarrow expr - term$	$expr.t := expr.t \,\|\| \, term.t \,\|\| \, '-'$
$expr \rightarrow term$	$expr.t := term.t$
$term \rightarrow 0$	$term.t := '0'$
$term \rightarrow 1$	$term.t := '1'$
….	….
$term \rightarrow 9$	$term.t := '9'$

Syntax Directed Definitions

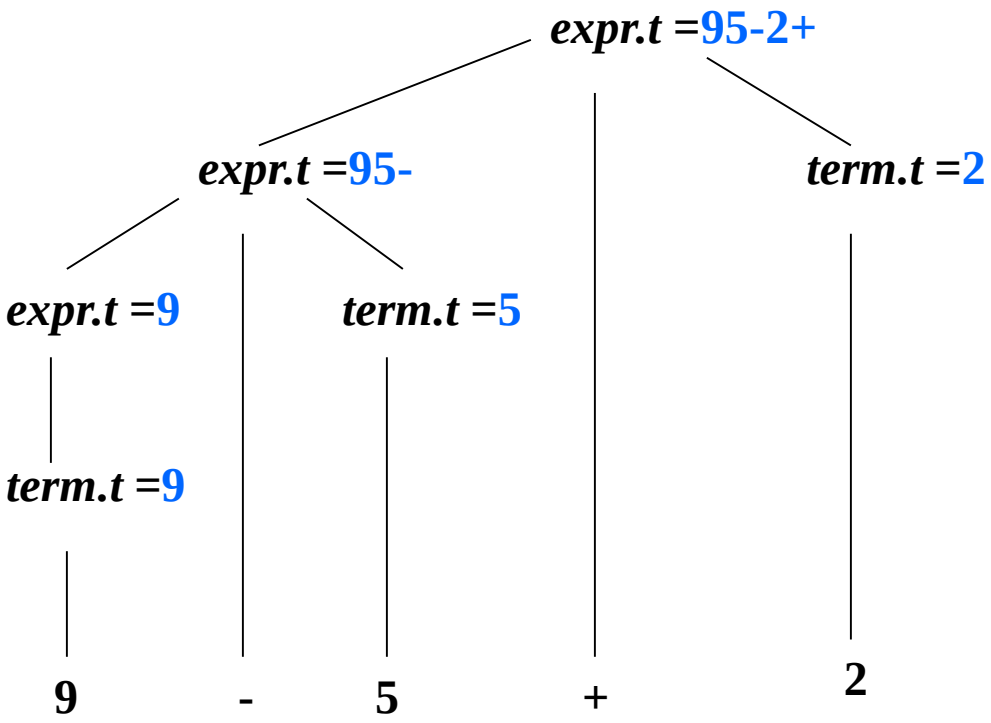


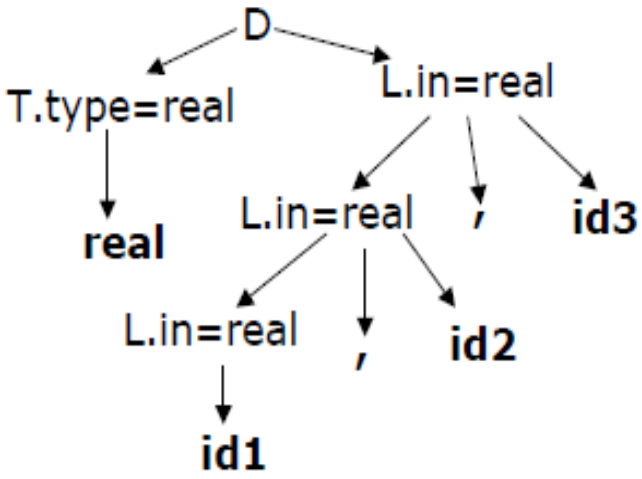
Fig: Annotated Parse Tree

Syntax Directed Definitions

Inherited Attributes: An attribute is inherited if the attribute value of a parse-tree node is determined from attribute values of its parent and siblings.

Example:

Production	Semantic rules
D::=T L	L.in:=T.type
T::= **int**	T.Type:=integer
T::=**real**	T.type:=real
L::=L1 **,id**	L1.in := L.in Addtype(id.entry,L.in)
L::=**id**	Addtype(id.entry,L.in)



Syntax Directed Translation Schemes

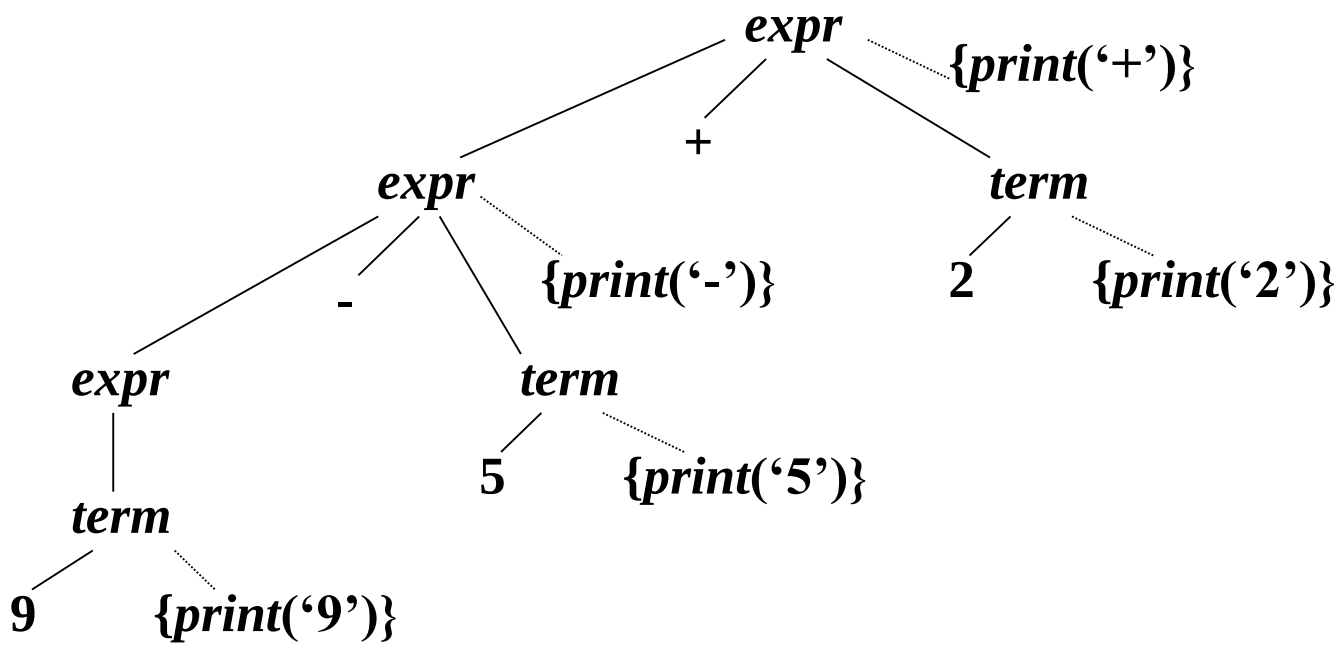
Syntax Directed Translation Schemes: A translation scheme is a context free grammar in which

- Attributes are associated with grammar symbols
- Semantic actions are enclosed between braces {} and are inserted within the right-hand side of productions.

Production	Semantic Action
$expr \rightarrow expr + term$	{*print*('+')}
$expr \rightarrow expr - term$	{*print*(' - ')}
$expr \rightarrow term$	{*print*( )}
$term \rightarrow 0$	{*print*('0')}
$term \rightarrow 1$	{*print*('1')}
….	….
$term \rightarrow 9$	{*print*('9')}

Syntax Directed Translation Schemes

A translation scheme is like a syntax-directed definition except the order of evaluation of the semantic rules is explicitly shown.



Class Exercises

1. Show the annotated parse tree for the following expressions.
   I. 2*3+4
   II. 2+3-4/5

Lecture References

A. Aho, R. Sethi and J. Ullman, ***Compilers: Principles, Techniques and Tools*** *(*The Dragon Book*)*, [Second Edition]

References

1. A. Aho, R. Sethi and J. Ullman, ***Compilers: Principles, Techniques and Tools(***The Dragon Book*)*, [Second Edition]

2. **Principles of Compiler Design** (2nd Revised Edition 2009) A. A. Puntambekar

3. Basics of Compiler Design Torben Mogensen