Simple Syntax-Directed Translation

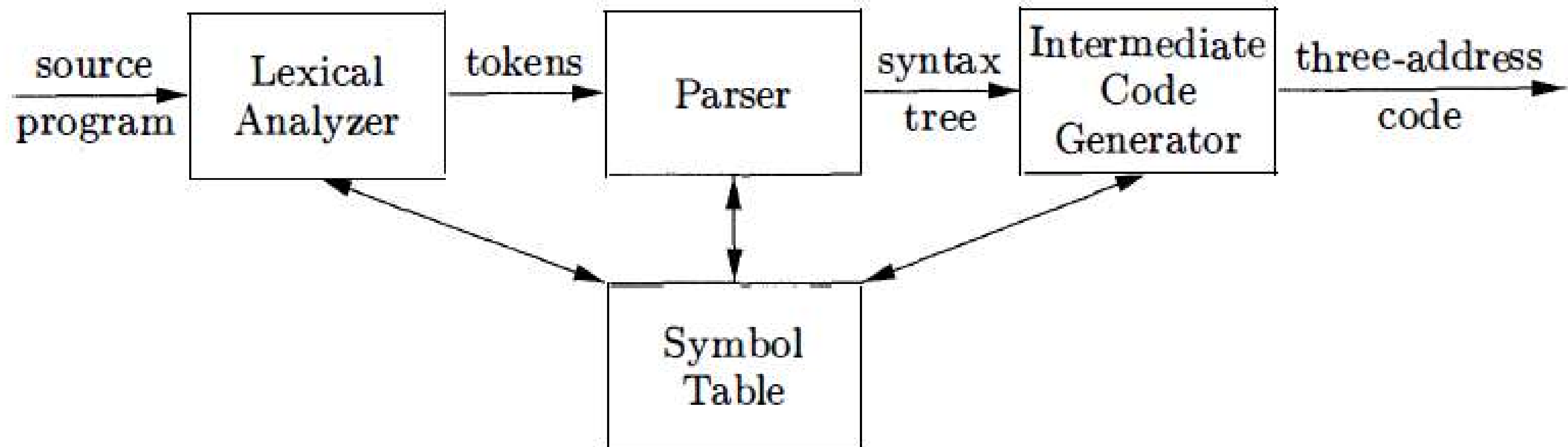
MD. Khorshed Alam

CSE, NDUB

Introduction

- The **analysis phase** of a compiler breaks up a source program into constituent pieces & produces an internal representation for it, called **intermediate code**.
- The **synthesis phase** translates the intermediate code into the target program.
- Analysis is organized around the "syntax" of the language to be compiled.
- **syntax** of a programming language describes the proper form of its programs,
- **semantics** of the language defines what its programs mean; that is, what each program does when it executes.
- For syntax: CFG or BNF

A model of a compiler front end



A model of a compiler front end

- **Lexical Analyzer**: allows a translator to handle multi-character constructs like identifiers, which are written as sequences of characters, but are treated as units called tokens during syntax analysis;

✓ **count + 1**, the identifier **count** is treated as a unit.

✓ The lexical analyzer in allows numbers, identifiers, and "white space" (blanks, tabs, and newlines) to appear within expressions.

- **Intermediate-code generation:**

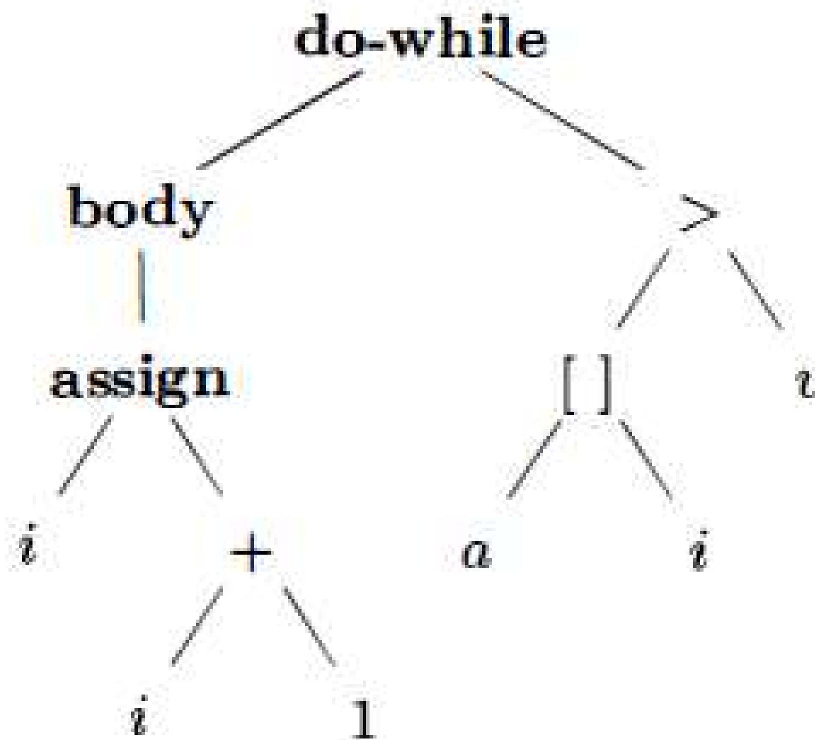
➢ abstract syntax trees or simply syntax trees, represents the hierarchical syntactic structure of the source program.

➢ three-address instructions

- **Parser:** produces a syntax tree, that is further translated into three-address code.

❑ **Some compilers combine parsing and intermediate-code generation into one component.**

Intermediate code for "**do i = i + 1 ; while (a [i] < v) ;** "



Abstract syntax tree

(a)

```
1: i = i + 1
2: t1 = a [ i ]
3: if t1 < v goto 1
```

Three-address code

(b)

Syntax Definition

- "**context-free grammar**," or "**grammar**" specify the syntax of a language.
- A grammar naturally describes hierarchical structure of language constructs.
- For example, an if-else statement in Java can have the form

if (expression) statement else statement

- An if-else statement is the *concatenation of the keyword* ***if****, an opening parenthesis, an expression, a closing parenthesis, a statement, the keyword* ***else****, and another statement.*
- Using the variable expr to denote an expression and the variable stmt to denote a statement, this structuring rule can be expressed as

stmt → if (expr) stmt else stmt

- in which the arrow may be read as "can have the form."
- Such a rule is called a production.
- Lexical elements- keyword **if** and the **parentheses** are called **terminals**.
- Variables- **expr** & **stmt** represent sequences of terminals and are called **non-terminals**.

Definition of Grammars

- **A context-free grammar (CFG) has four components:**
- 1. A set of ***terminal*** symbols, sometimes referred to as "tokens."
  - The terminals are the elementary symbols of the language defined by the grammar.
- 2. A set of ***nonterminals***, sometimes called "syntactic variables."
  - Each non-terminal represents a set of strings of terminals
- 3. A set of ***productions***,
  - each production consists of a non-terminal, called the head or left side of the production, an arrow, and a sequence of terminals and/or non-terminals , called the body or right side of the production

Definition of Grammars

- The intuitive intent of a production is to specify one of the written forms of a construct; if the head non-terminal represents a construct, then the body represents a written form of the construct .
- 4. A designation of one of the non-terminals as the start symbol

CFG = {T, NT, P, S}

Example

$$
\begin{array}{lcl}
list & \rightarrow & list + digit \\
list & \rightarrow & list - digit \\
list & \rightarrow & digit \\
digit & \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{array}
$$

- **Terminals**: **+ - 0 1 2 3 4 5 6 7 8 9**

➢ A string of terminals is a sequence of zero or more terminals.

➢ The string of zero terminals is called the **empty string (ε)**

- **Non-terminals:** ***list*** ***digit***
- **Start symbol:** ***list***

Derivations

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

Example

$$
\begin{array}{lll}
list & \rightarrow & list + digit \quad \textbf{(1)} \\
list & \rightarrow & list - digit \quad \textbf{(2)} \\
list & \rightarrow & digit \quad \textbf{(3)} \\
digit & \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \quad \textbf{(4)}
\end{array}
$$

- **9-5+2** is a ***list**:*

a) 9 is a ***list*** by production (3) , since 9 is a ***digit***.

b) 9-5 is a ***list*** by production (2) , since 9 is a **list** & 5 is a **digit**

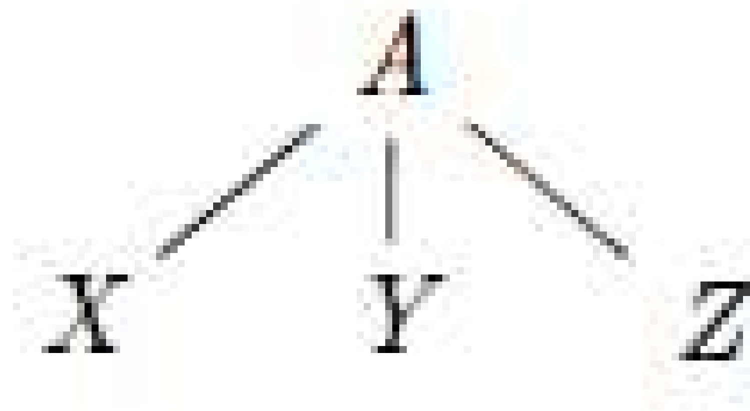
c) 9-5+2 is a ***list*** by production (1) , since 9-5 is a ***list*** and 2 is a ***digit***.

Parsing

- Parsing is the problem of taking a string of terminals and figuring out how to derive it from the start symbol of the grammar,
- if it cannot be derived from the start symbol of the grammar, then reporting syntax errors within the string.
- Parsing is one of the most fundamental problems in all of compiling
- A source program has multi-character lexemes that are grouped by the lexical analyzer into tokens, whose first components are the terminals processed by the parser.

Parse Trees

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



Properties of the Parse Tree

1. The root is labeled by the start symbol.
2. Each leaf is labeled by a terminal or by ε.
3. Each interior node is labeled by a non-terminal.
4. If A is the non-terminal labeling some interior node and $X_1, X_2, \bullet \bullet \bullet, X_n$ are the labels of the children of that node from left to right, then there must be a production $A \rightarrow X_1 X_2 \cdots X_n$.

- Here, $X_1, X_2, \ldots, X_n$ each stand for a symbol that is either a terminal or a non-terminal .
- As a special case, if $A \rightarrow \varepsilon$ is a production, then a node labeled A may have a single child labeled ε

Tree Terminology

- ❑A tree consists of one or more **nodes**. Nodes may have labels, which in this book typically will be grammar symbols.
  - ➢When we draw a tree, we often represent the nodes by these labels only.

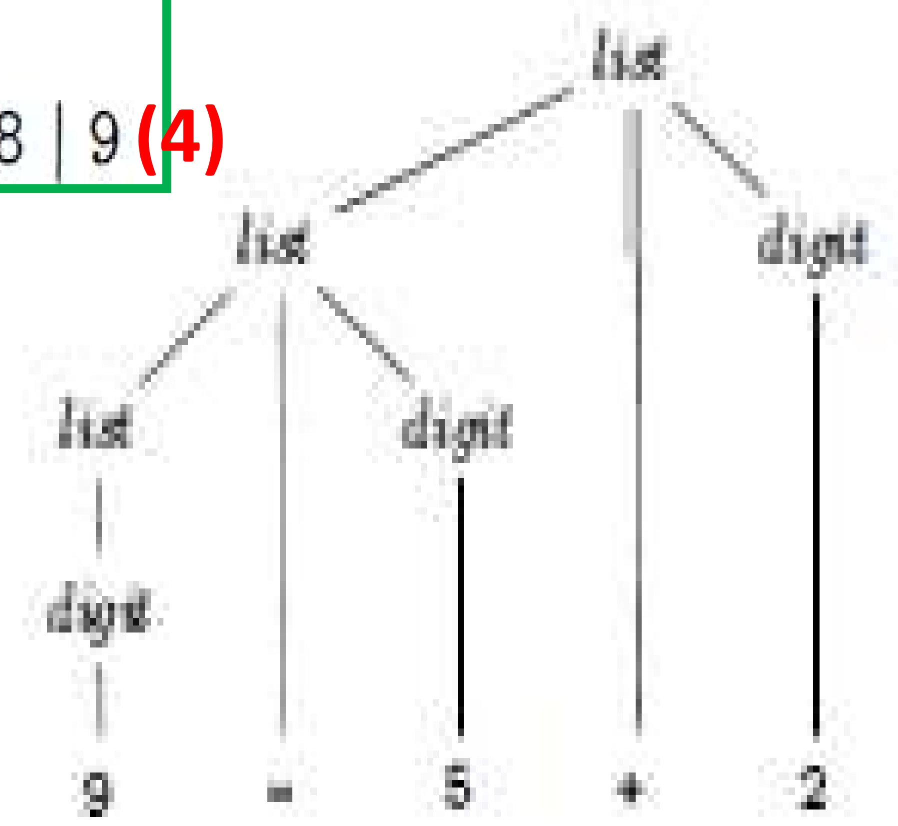
- ❑ Exactly one node is the **root**.
  - ➢All nodes except the root have a unique **parent**; the root has no parent.
  - ➢When we draw trees , we place the parent of a node above that node and draw an edge between them.
  - ➢The **root** is then the **highest (top)** node.

Tree Terminology

- If node N is the parent of node M, then M is a child of N. The children of one node are called **siblings**. They have an order, *from the left*, and when we draw trees , we order the children of a given node in this manner.
- A node with no children is called a **leaf**. Other nodes - those with one or more children - are ***interior nodes***.
- A **descendant** of a node N is either N itself, a child of N, a child of a child of N, and so on , for any number of levels. We say node N is an ***ancestor*** of node M if M is a descendant of N.

Example: 9-5+2

list	$\rightarrow$	*list* + *digit*	**(1)**
list	$\rightarrow$	*list* - *digit*	**(2)**
list	$\rightarrow$	*digit*	**(3)**
digit	$\rightarrow$	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9	**(4)**

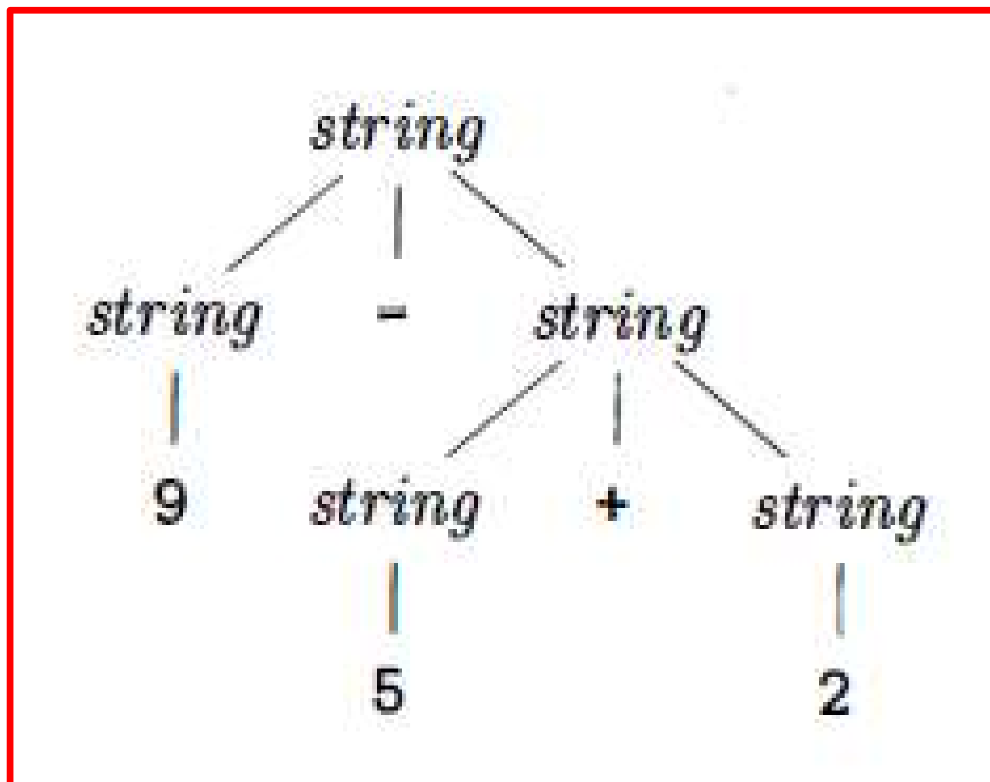


Ambiguity

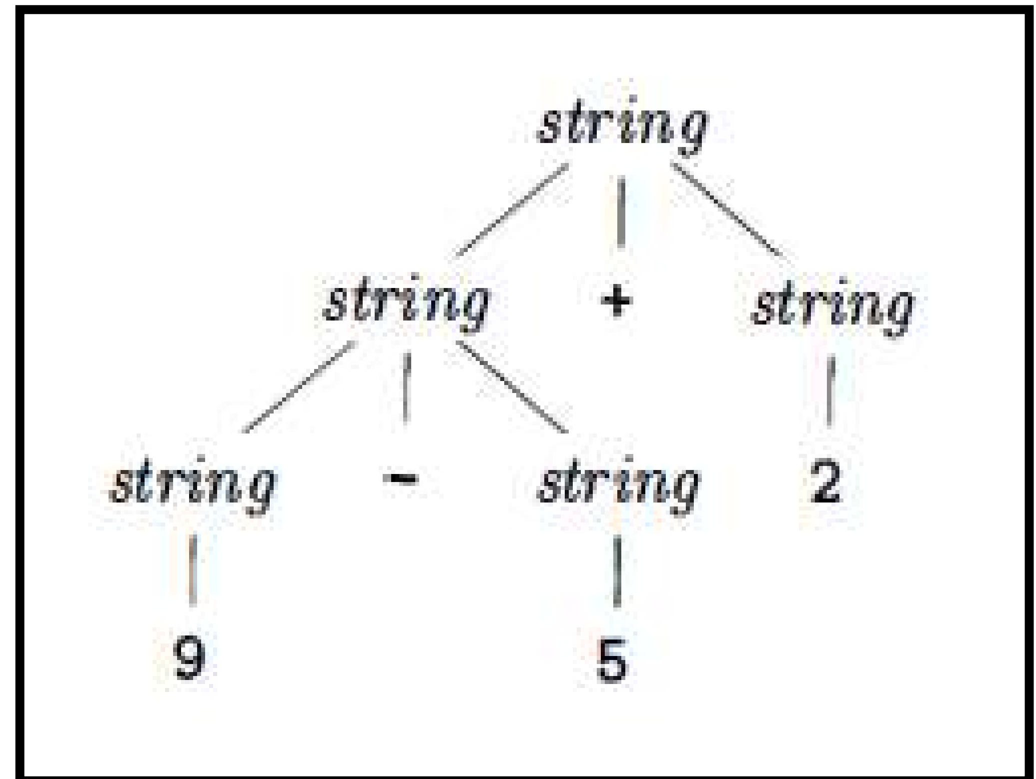
- **ambiguous grammar**: a grammar can have more than one parse tree generating a given string of terminals.
- Since a string with more than one parse tree usually has more than one meaning,
  - we need to design unambiguous grammars for compiling applications,
  - or to use ambiguous grammars with additional rules to resolve the ambiguities

- 9-5+2 has more than one parse tree with this grammar.
- Two ways: (9-5) +2 and 9- (5+2) .

$$string \rightarrow string + string \mid string - string \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$



- (9-5) +2



- 9- (5 + 2)

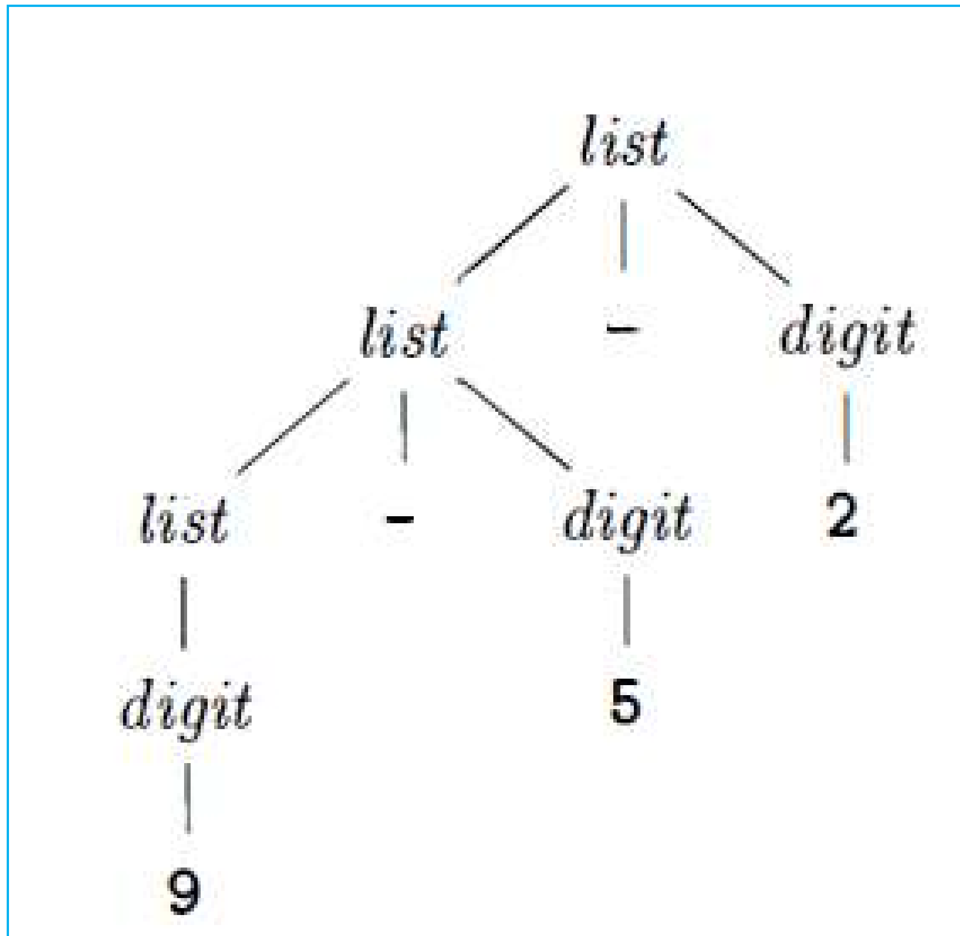
Associativity of Operators

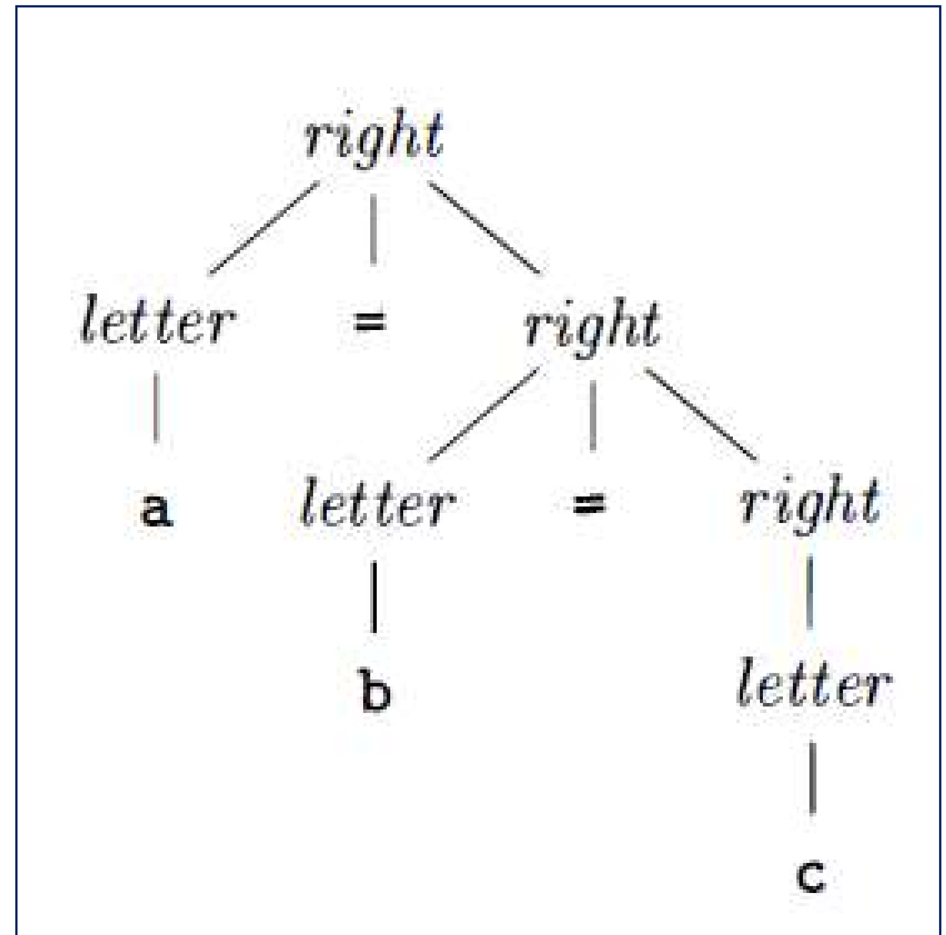
- By convention, 9+5+2 is equivalent to (9+5) +2
- 9-5-2 is equivalent to (9-5) -2.
- When an operand like 5 has operators to its left and right, conventions are needed for deciding which operator applies to that operand.
- operator + associates to the left, because an operand with plus signs on both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

Associativity of Operators

- exponentiation are right-associative.
- assignment operator = in C and its descendants is right associative;
  - the expression a=b=c is treated in the same way as the expression a= (b=c)
- **right → letter = right I letter**
- **letter → a I b I . . . I z**

Parse Tree for 9-5-2 and a=b=c





Left associative

Right associative

Precedence of Operators

- **9+5*2**

➢ 02 possible interpretations:

  ❑ **(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.
- Rules defining the relative precedence of operators are needed 'when more than one kind of operator is present

Precedence of Operators

- We say that ***** has higher precedence than **+** if * takes its operands before + does.
- In ordinary arithmetic, multiplication & division have higher precedence than addition & subtraction
- 9+5*2 ⇒ 9+ (5*2)
- 9*5+2 ⇒ (9*5) +2

Example 2.6 : A grammar for arithmetic expressions can be constructed from a table showing the associativity and precedence of operators. Operators on the same line have the same associativity and precedence:

left - associative: + -
left - associative: * /

- We create two nonterminals **expr** and **term** for the two levels of precedence, and an extra nonterminal **factor** for generating basic units in expressions.
- The basic units in expressions are presently digits and parenthesized expressions.

$$\text{factor} \rightarrow \textbf{digit} \mid (\text{expr})$$

- Binary operators, * and /, that have the highest precedence.
- Since these operators associate to the left, the productions are simiilar to those for lists that associate to the left.

$$
\begin{aligned}
Term \rightarrow\ & term * factor \\
\mid\ & term\ /\ factor \\
\mid\ & factor
\end{aligned}
$$

- Similarly, **expr** generates lists of terms separated by the additive operators

$$
\begin{aligned}
expr \rightarrow\ & expr + term \\
\mid\ & expr - term \\
\mid\ & term
\end{aligned}
$$

Resulting Grammars

expr → expr + term | expr – term | term

term → term * factor | term / factor | factor

factor → **digit |** (expr)

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

$$expr \rightarrow expr_1 + term$$

- *expr* is the sum of the two sub expressions $expr_1$ & *term*.
- (The subscript in $expr_1$ 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:
  - translate *$expr_1$* ;
  - translate *term*;
  - Handle *+*;

Attributes

- An attribute is any quantity associated with a programming Construct
- Examples: data types of expressions, the number of instructions in the generated code, or the location of the first instruction in the generated code for a construct , among many other possibilities.
- Since we use grammar symbols (nonterminals and terminals) to represent programming constructs, we extend the notion of attributes from constructs to the symbols that represent them.

(Syntax- directed) translation schemes

- A translation scheme is a notation for attaching program fragments to the productions of a grammar.
- The program fragments are executed when the production is used during syntax analysis.
- The combined result of all these fragment executions, in the order induced by the syntax analysis, produces the translation of the program to which this analysis/synthesis process is applied.

Postfix Notation

- The postfix notation for an expression E:
  - 1. If E is a variable or constant , then the postfix notation for E is E itself.
  - 2. If E is an expression of the form E_1 **op** E_2 , where **op** is any binary operator, then the postfix notation for E is E_1\`$E_2$\` **op**, where E_1\` & $E_2$\` are the postfix notations for E_1 & E_2
  - 3. If E is a parenthesized expression of the form (E1), then the postfix notation for E is the same as the postfix notation for E_1 .

- The postfix notation for **(9-5)+2 ≡ 95-2+**
  - The translations of 9, 5, and 2 are the constants themselves, by rule (1).
  - The translation of 9-5 is 95- by rule (2)
  - The translation of (9-5) is the same by rule (3)
- Having translated the parenthesized subexpression, we may apply rule (2) to the entire expression, with (9-5) in the role of E_1 and 2 in the role of E_2 , to get the result **95-2+**

- **9 - (5+2) ≡ 952+-**
- 5+2 is first translated into 52+, and this expression becomes the second argument of the minus sign.

Postfix Notation

- No parentheses are needed in postfix notation, because the position and arity (number of arguments) of the operators permits only one decoding of a postfix expression.
  - **Trick**
  - repeatedly scan the postfix string from the left, until you find an operator.
  - Then, look to the left for the proper number of operands, and group this operator with its operands.
  - Evaluate the operator on the operands, and replace them by the result.
  - Then repeat the process, continuing to the right and searching for another operator.

- **952+-3***
- Scanning from the left , we first encounter the plus sign.
- Looking to its left we find operands 5 and 2
- Their sum , 7, replaces 52+, and we have the string **97-3***
- Now, the leftmost the result of the subtraction leaves 23*
- Last, the multiplication sign applies to 2 and 3, giving the result 6

Synthesized Attributes

- The idea of associating quantities with programming constructs-for example, values and types with expressions-can be expressed in terms of grammars.
- We associate attributes with nonterminals and terminals.
- Then, we attach rules to the productions of the grammar;
    - ✓ these rules describe how the attributes are computed at those nodes of the parse tree where the production in question is used to relate a node to its children.

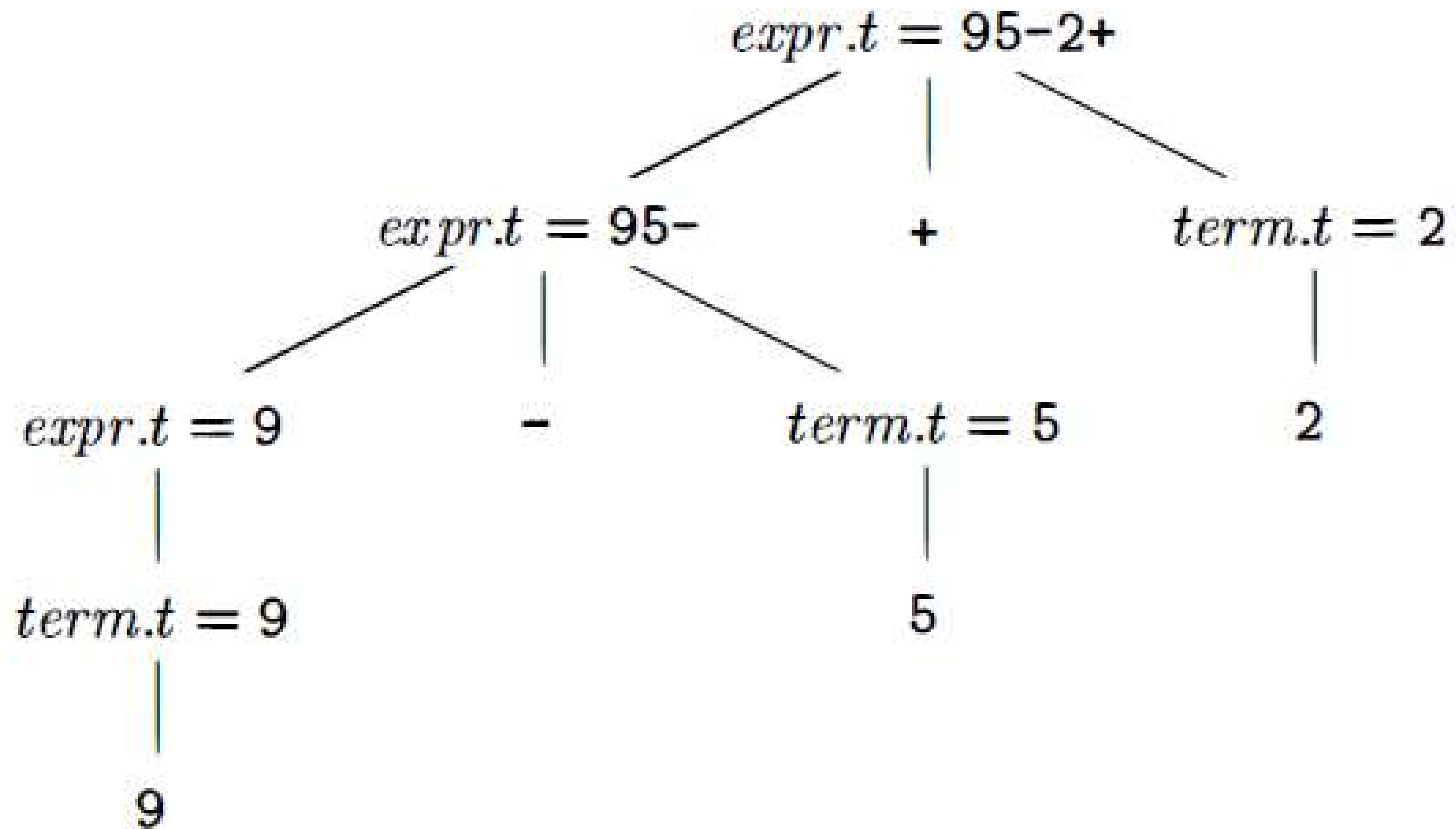
❑ **A syntax-directed definition associates**

- 1. With each grammar symbol, a set of attributes, and
- 2. With each production, a set of semantic rules for computing the values of the attributes associated with the symbols appearing in the production.

Synthesized Attributes

- Attributes can be evaluated as follows.
  - For a given input string x, construct a parse tree for x.
  - Then, apply the semantic rules to evaluate attributes at each node in the parse tree:
- ➢ Suppose a **node N** in a parse tree is labeled by the grammar **symbol X** .
- ➢ We write **X.a** to denote the Value of attribute **a** of **X** at that node.
- A parse tree showing the attribute values at each node is called an **annotated parse tree**.

Annotated parse tree for 9-5+2 with an attribute **t** associated with the nonterminals **expr** & **term**.



Attribute values at nodes in a parse tree

Synthesized Vs. Inherited Attribute

- **Synthesized attribute**: An attribute is said to be ***synthesized*** if its value at a parse-tree node N is determined from attribute values at the children of N & at N itself.
- **Inherited attribute:** have their value at a parse-tree node determined from attribute values at the node itself, its parent , and its siblings in the parse tree.

Example 2.10 : syntax-directed definition for translating expressions consisting of digits separated by plus or minus signs into postfix notation.

Each nonterminal has a string-valued attribute **t** that represents postfix notation for the expression generated by that nonterminal in a parse tree. The symbol || in the semantic rule is the operator for concatenation

PRODUCTION	SEMANTIC RULES
$expr \to expr_1 + term$	$expr.t = expr_1.t \,\|\|\, term.t \,\|\|\, '+'$
$expr \to expr_1 - term$	$expr.t = expr_1.t \,\|\|\, term.t \,\|\|\, '-'$
$expr \to term$	$expr.t = term.t$
$term \to 0$	$term.t = '0'$
$term \to 1$	$term.t = '1'$
...	...
$term \to 9$	$term.t = '9'$

Infix to postfix translation

Parsing

- Parsing is the process of determining how a string of terminals can be generated by a grammar.
- A parser must be capable of constructing the tree in principle, or else the translation cannot be guaranteed correct.
- **Yacc**; Parser tool; can implement the translation scheme without modification

Parsing

- For any CFG there is a parser that takes at most **O (n^3)** time to parse a string of n terminals.
  - But cubic time is generally too expensive.
- For real programming languages, design a grammar that can be parsed quickly (Linear-time algorithms)
- Programming-language parsers almost always make a single left-to-right scan over the input, looking ahead one terminal at a time, and constructing pieces of the parse tree as they go.

Parsing Methodology

- Mostly 02 classes:
  - **Top-down**
  - **Bottom-up**

- **Top-down**: construction starts at the root & proceeds towards the leaves
- **Bottom-up:** construction starts at the leaves & proceeds towards the root
  - top-down parsers is more popular; efficient parsers can be constructed more easily by hand
  - Bottom-up parsing, can handle a larger class of grammars & translation schemes, so software tools for generating parsers directly from grammars

Top-down parsing

- Starting with the **root (stmt)**, & repeatedly performing the following two steps:

- 1. At node **N**, labeled with nonterminal A, select one of the productions for A & construct children at N for the symbols in the production body.
- 2. Find the next node at which a **subtree** is to be constructed, typically the leftmost unexpanded nonterminal of the tree

Example

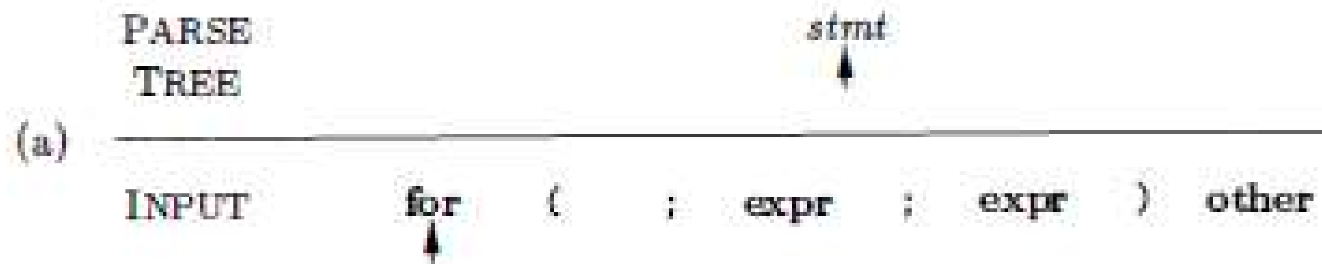
$$
\begin{array}{rcl}
stmt & \rightarrow & \textbf{expr} \ ; \\
 & | & \textbf{if} \ (\ \textbf{expr} \ ) \ stmt \\
 & | & \textbf{for} \ (\ optexpr \ ; \ optexpr \ ; \ optexpr \ ) \ stmt \\
 & | & \textbf{other} \\
optexpr & \rightarrow & \epsilon \\
 & | & \textbf{expr}
\end{array}
$$

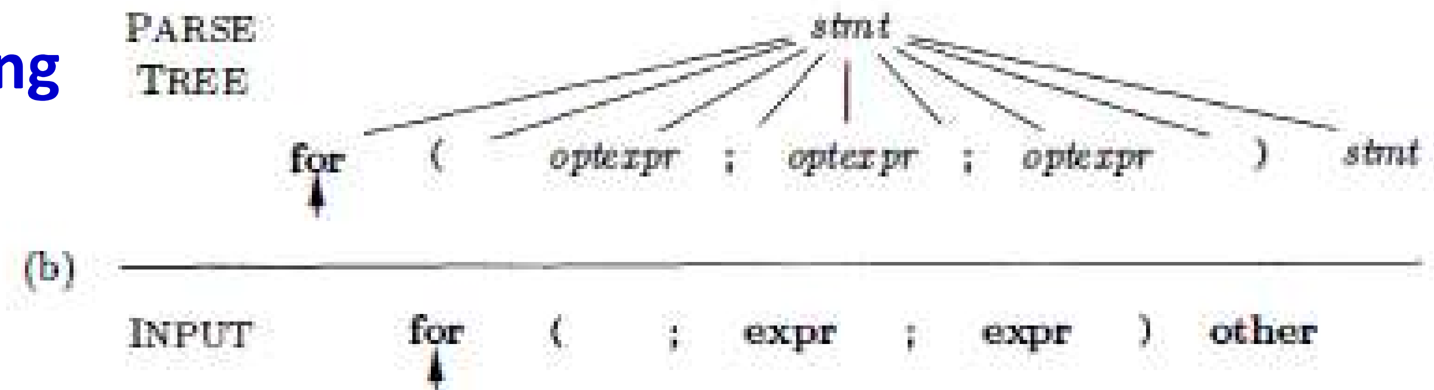
- **Lookahead symbol:** current terminal being scanned in the input

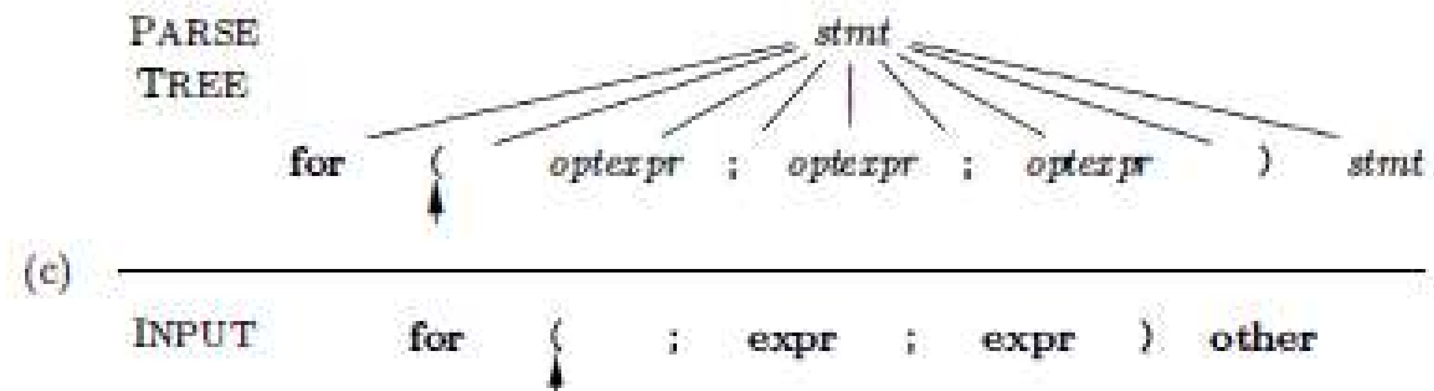
for r (; expr ; expr) other

- Initially, the lookahead symbol is the first
  - leftmost , terminal of the input string.

Top-down parsing while scanning the input from left to right







Left Recursion

- For a recursive-descent parser to loop forever. A problem arises with "left-recursive" productions like

$$expr \rightarrow expr + term$$

- where the leftmost symbol of the body is the same as the nonterminal at the head of the production.
- Suppose the procedure for **expr** decides to apply this production. The body begins with expr so the procedure for expr is called recursively.
- Since the **lookahead symbol** changes only when a terminal in the body is matched, no change to the input took place between recursive calls of **expr**.
- As a result, the second call to **expr** does exactly what the first call did, which means a third call to **expr,** and so on, forever.