

# CS419 Compilers Construction

## A Simple One-Pass Compiler [Chapter 2]

### Lecture 7

*Edited by Dr. Ismail Hababeh*

*Adapted from slides by Dr. Mohammad Daoud*

*German Jordanian University*

**Originally from slides by Dr. Robert A. Engelen**

# Postfix Notation

- The Postfix notation is used to **represent algebraic expressions** without using parenthesis ( ).
- The expressions written in postfix form are **evaluated faster** compared to infix notation as parenthesis are not required in postfix.

# Postfix Notation Rules

- If  $E$  is an expression of the form  $E_1 \text{ op } E_2$ , where  $\text{op}$  is any binary operator, then the postfix notation for  $E$  is  $E_1 E_2 \text{op}$ , where  $E_1$  and  $E_2$  are the operands
- IF  $E$  is variable or constant, then the postfix notation for  $E$  is  $E$  itself.
- The postfix notation for  $(E)$  is the same as the postfix notation for  $E$ , thus no need for the  $( )$
- Examples:
  - The postfix notation for  $(9-5) + 2$  is  $95-2+$
  - The postfix notation for  $9-(5+2)$  is  $952+-$

# Postfix Evaluation

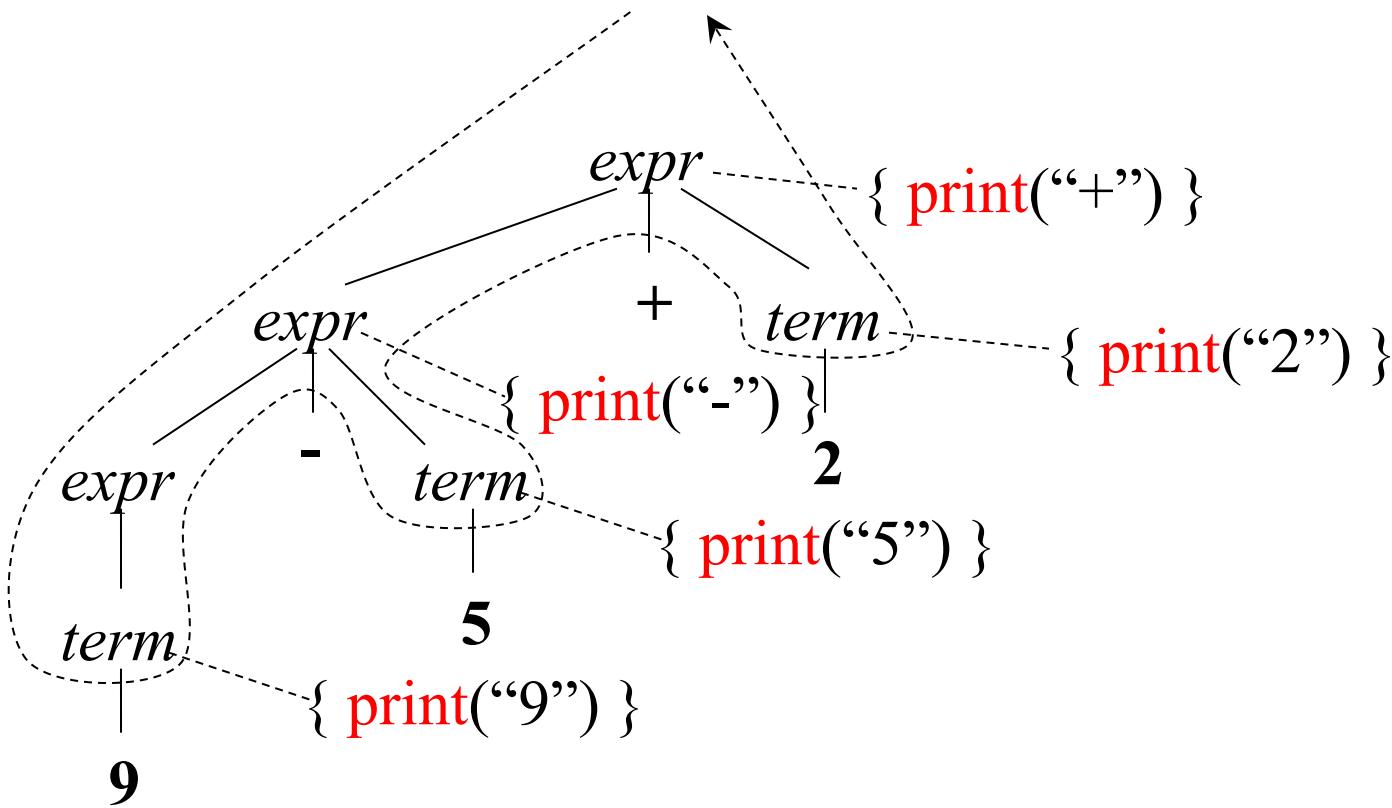
- To **read** the postfix notation, repeatedly scan the postfix string from the left until an operator is found, then evaluate the operator on the operands.
- Example: Evaluate the following postfix notation expression **952+-3\***  
Solution:  $(9-(5+2))*3 = 6$
- **Syntax-Directed Translation** is used to translate infix expressions, for example, **(9-5)+2** to **postfix notation 95-2+**, evaluate expressions and build syntax trees for programming constructs.

# Syntax-Directed Translation - Example

- Translate the infix expression  $9-5+2$  into postfix notation  $95-2+$  using the following productions.

$expr \rightarrow expr + term$	$\{ \text{print}(“+”) \}$	Print is performed without the need to store attributes.
$expr \rightarrow expr - term$	$\{ \text{print}(“-”) \}$	<i>semantic actions</i>
$expr \rightarrow term$		attaches program
$term \rightarrow 0$	$\{ \text{print}(“0”) \}$	fragments to productions
$term \rightarrow 1$	$\{ \text{print}(“1”) \}$	in a grammar
...	...	
$term \rightarrow 9$	$\{ \text{print}(“9”) \}$	

# Translation Scheme Parse Tree - Example



Translates the infix **9-5+2** into postfix **95-2+**

# FIRST of Production

$\text{FIRST}(\alpha)$  is the set of **terminals** that appear as the **first symbols** of one or more strings generated from production ( $\alpha$ ). FIRST used to write a **predictive parser**.

*Example:* Write the  $\text{FIRST}(\alpha)$  of the following Grammar:

$$\begin{aligned} \text{type} &\rightarrow \textcolor{red}{simple} \\ &\mid \textcolor{red}{^ id} \\ &\mid \textcolor{red}{array} \ [ \textcolor{red}{simple} ] \ \textcolor{red}{of} \ \textcolor{red}{type} \end{aligned}$$

$$\begin{aligned} \text{simple} &\rightarrow \text{integer} \\ &\mid \text{char} \\ &\mid \text{num} \text{ dot } \text{num} \end{aligned}$$

**Solution:**

$$\text{FIRST}(\textcolor{red}{simple}) = \{ \text{integer}, \text{char}, \text{num} \}$$

$$\text{FIRST}(\textcolor{red}{^ id}) = \{ \textcolor{red}{^} \}$$

$$\text{FIRST}(\textcolor{red}{type}) = \{ \text{integer}, \text{char}, \text{num}, \textcolor{red}{^}, \text{array} \}$$

# Using FIRST - Example

$\text{expr} \rightarrow \text{term rest}$   
 $\text{rest} \rightarrow + \text{ term rest}$   
| -  $\text{term rest}$   
|  $\epsilon$   
 $\text{term} \rightarrow 0|1|\dots|9$

```
procedure rest();  
begin  
  if lookahead in FIRST(+ term rest) then  
    match('+'); term(); rest()  
  else if lookahead in FIRST(- term rest) then  
    match('-'); term(); rest()  
  else return  
end;
```

Note: When a **nonterminal  $A$**  has two (or more) productions as in

$$\begin{array}{l} A \rightarrow \alpha \\ | \beta \end{array}$$

Then  $\text{FIRST}(\alpha)$  and  $\text{FIRST}(\beta)$  must be disjoint for predictive parsing to work. **Why?**

# Left Factoring

When more than one production for nonterminal  $A$  starts with the **same symbols**, then FIRST sets are **not disjoint** and lead to ambiguity.

Example:  $stmt \rightarrow \text{if } expr \text{ then } stmt$

|  $\text{if } expr \text{ then } stmt \text{ else } stmt$

How can we solve the FIRST ambiguity?

By using *left factoring*:

$stmt \rightarrow \text{if } expr \text{ then } stmt \text{ opt\_else}$

$\text{opt\_else} \rightarrow \text{else } stmt$

|  $\epsilon$

# Left Recursion

When a production for nonterminal  $A$  starts with a self reference (*left recursive productions*) then a predictive parser loops forever

$$\begin{aligned} A \rightarrow & A \alpha \\ | & \beta \\ | & \gamma \end{aligned}$$

Solution: eliminate *left recursive productions* by systematically rewriting the grammar using (*right recursive productions*)

$$\begin{aligned} A \rightarrow & \beta R \\ | & \gamma R \\ R \rightarrow & \alpha R \\ | & \varepsilon \end{aligned}$$

# Left Recursion Problem

Translate from simple arithmetic expressions into postfix form. Start with the syntax-directed translation scheme.

**Problem! left recursion !**

$expr \rightarrow expr + term$	{ print(“+”) }
$expr \rightarrow expr - term$	{ print(“-”) }
$expr \rightarrow term$	
$term \rightarrow 0$	{ print(“0”) }
$term \rightarrow 1$	{ print(“1”) }
...	...
$term \rightarrow 9$	{ print(“9”) }

# Left Recursion Elimination

$$\begin{aligned}
 \textit{expr} &\rightarrow \textit{expr} + \textit{term} \quad \{ \text{print}(“+”) \} \\
 \textit{expr} &\rightarrow \textit{expr} - \textit{term} \quad \{ \text{print}(“-”) \} \\
 \textit{expr} &\rightarrow \textit{term} \\
 \textit{term} &\rightarrow \mathbf{0} \quad \{ \text{print}(“0”) \} \\
 \textit{term} &\rightarrow \mathbf{1} \quad \{ \text{print}(“1”) \} \\
 &\dots \quad \dots \\
 \textit{term} &\rightarrow \mathbf{9} \quad \{ \text{print}(“9”) \}
 \end{aligned}$$

left recursion elimination:

$$\begin{aligned}
 \textit{expr} &\rightarrow \textit{term} \textit{rest} \\
 \textit{rest} &\rightarrow + \textit{term} \{ \text{print}(“+”) \} \textit{rest} \mid - \textit{term} \{ \text{print}(“-”) \} \textit{rest} \mid \varepsilon \\
 \textit{term} &\rightarrow \mathbf{0} \{ \text{print}(“0”) \} \\
 \textit{term} &\rightarrow \mathbf{1} \{ \text{print}(“1”) \} \\
 &\dots \\
 \textit{term} &\rightarrow \mathbf{9} \{ \text{print}(“9”) \}
 \end{aligned}$$

# Translate Infix Expressions into Postfix Form

Program to translate infix  
expressions into postfix form.

$\begin{array}{c} expr \rightarrow term\ rest \\ \hline rest \rightarrow +\ term\ \{ print(“+”) \}\ rest \\ \quad | -\ term\ \{ print(“-”) \}\ rest \\ \quad | \epsilon \end{array}$

$\begin{array}{c} term \rightarrow 0\ \{ print(“0”) \} \\ term \rightarrow 1\ \{ print(“1”) \} \\ \dots \\ term \rightarrow 9\ \{ print(“9”) \} \end{array}$

```
main()
{
    lookahead = getchar();
    expr();
}
expr()
{
    term();
    while (1) /* optimized by inlining rest()
                and removing recursive calls */
    {
        if (lookahead == '+')
        {
            match('+'); term(); putchar('+');
        }
        else if (lookahead == '-')
        {
            match('-'); term(); putchar('-');
        }
        else break;
    }
    term();
    if (isdigit(lookahead))
    {
        putchar(lookahead); match(lookahead);
    }
    else error();
}
match(int t)
{
    if (lookahead == t)
        lookahead = getchar();
    else error();
}
error()
{
    printf("Syntax error\n");
    exit(1);
}
```

# Reading One Character Ahead

- Reading ahead is required to differentiate numbers as 1 from 10 or operators as < from <=
- One-character read-ahead is usually used a variable called *peek* to hold the next input character
- Reading ahead is perform only when it is required
  - Example: the operator \* is identified without reading ahead, therefore *peek* is set to blank

# Reading Constants

- Constants are allowed by creating a terminal symbol, **num**
- When the lexical analyzer encounters a sequence of digits in the input stream, it creates a token consisting of the **terminal num** and the **integer-valued attribute** computed for the digits:

token : <**num**, attribute>

- **Example:**

The input **123** is transformed into a token <**num**, 123>

# Reading Constants Algorithm

To compute a constant value consists of a sequence of digits:

```
if(peek holds a digit)
{
    v = 0;

    do{
        v = v*10 + integer value of digit peek;
        peek = next input character;
    }while (peek holds a digit);

    return token <num, v>;
}
```

# Reading Keywords and Identifiers

- Language keywords, such as **for**, **do**, **if**, ... etc. are fixed character strings.
- Language keywords are **reserved**, and hence they cannot be used as identifiers
- Character strings are also used as identifiers to name variables, arrays, functions, ....
- The parser treats identifiers as terminals.
- When the **lexical analyzer encounters** an **identifier** in the input stream, it **creates a token** consisting of the **terminal id** and its **lexeme**.
- Example: the input **count = count + increment** is transformed into the following tokens  
**<id, “count”> <“=”> <id, “count”> <“+”> <id, “increment”>**

# Reading Identifiers

When the **lexical analyzer** reads a string that could form an identifier, it checks the **symbol table**\*:

- If the symbol table includes the lexeme, it returns the identifier token from the table.
- Otherwise, it inserts a token with terminal **id** in the symbol table and returns the token with terminal **id**.

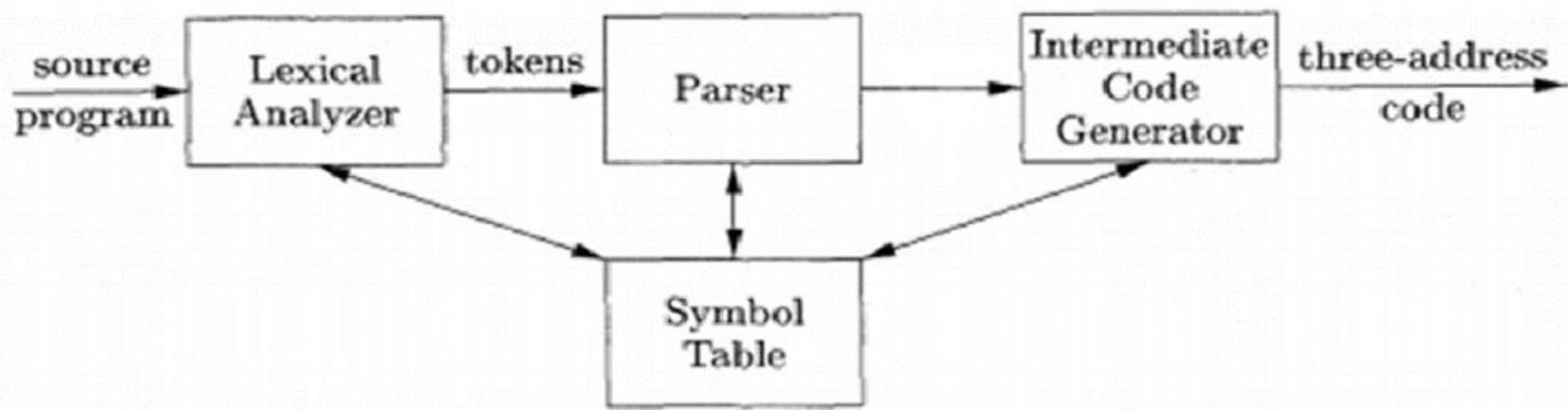
\***Symbol Table** is a data structure used by the compiler to keep track of identifiers used in the source program. Symbol table is used at compile-time but not used at run time.

# Reading Identifiers Algorithm

```
if(peek holds a letter)
{ // collect letters and/or digits into a buffer b
    s = string formed from the characters in buffer b;
    // The Lexical Analyzer lookup s in the symbol table
    // If the symbol table has an entry for s, then the
    // token retrieved by lookup is returned
    w = token returned by lookup(s);
    if( w != null) return w;
    // If the table does not contain an entry for s, then
    // the token <id, s> is inserted in the table
    else{
        insert(s, t); // t is the token that holds s
        return token <id, s>;
    }
}
```

# Symbol Table Entry

The **symbol table** is accessible to all phases of the compiler



A model of a compiler front-end Pass

# Symbol Table Entry

Each **entry** in the symbol table contains a **string** and a **token value**:

```
struct entry
{    char *lexptr; // lexeme (string)
    int tokenvalue;
};

struct entry symtable[];
```

**insert(s, t)**: inserts new entry for string **s** token **t**

**lookup(s)**: returns array index to entry for string **s** or returns 0 if s is not found

The symbol table is initialized with the reserved keywords and their tokens

Possible Symbol table implementations:  
- simple C code  
- hashtables