Associativity of Operators

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 takes that operand. We say that the operator + *associates to the left* because an operand with plus signs on both sides of it is taken by the operator to its left. In most programming languages the four arithmetic operators, addition, subtraction, multiplication, and division are left associative.

<<Associativity.ptt>>

Some common operators such as exponentiation are right associative. As another example, the assignment operator = in C is right associative; in C, 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:

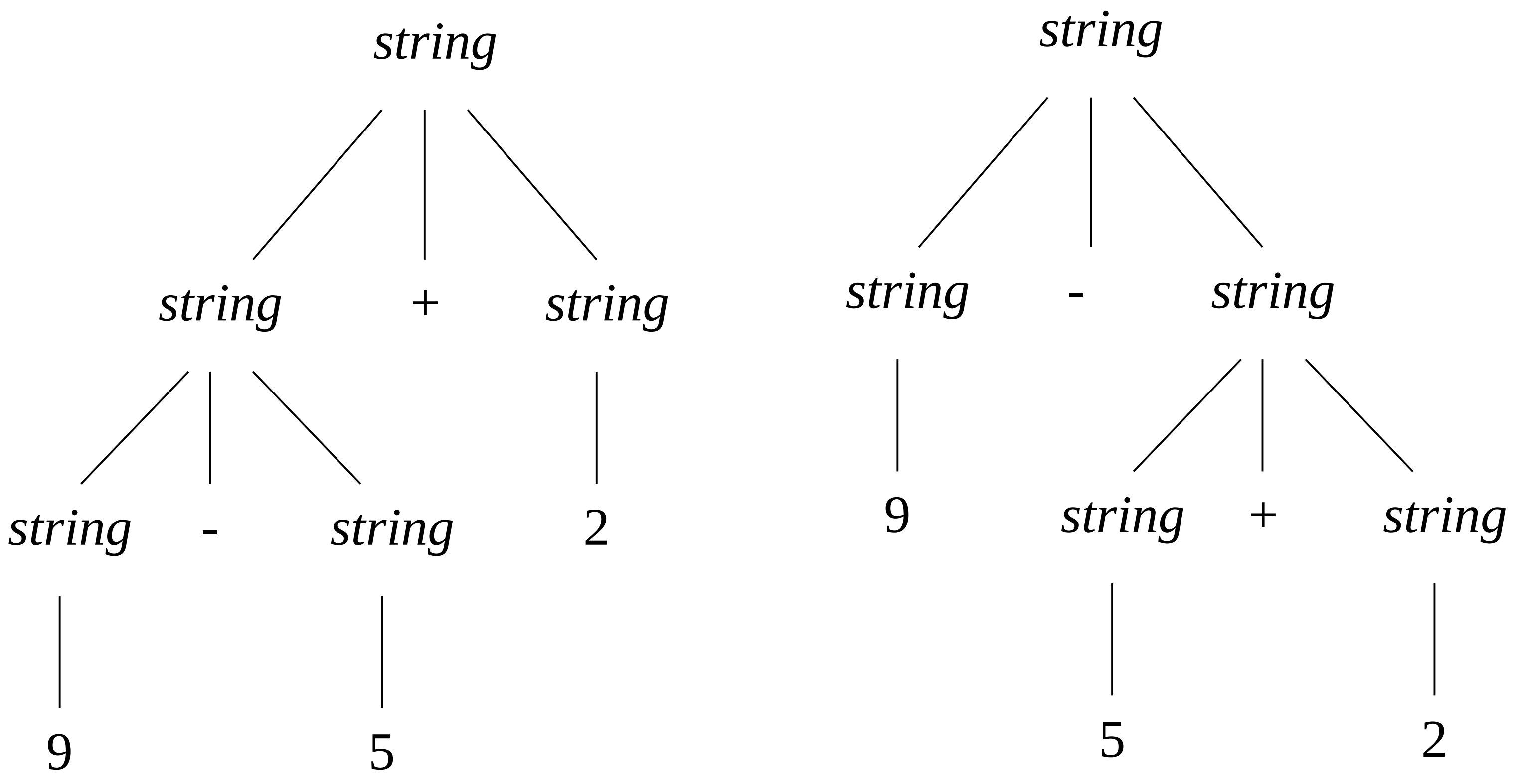

Fig 2.3. Two parse trees for 9-5+2.

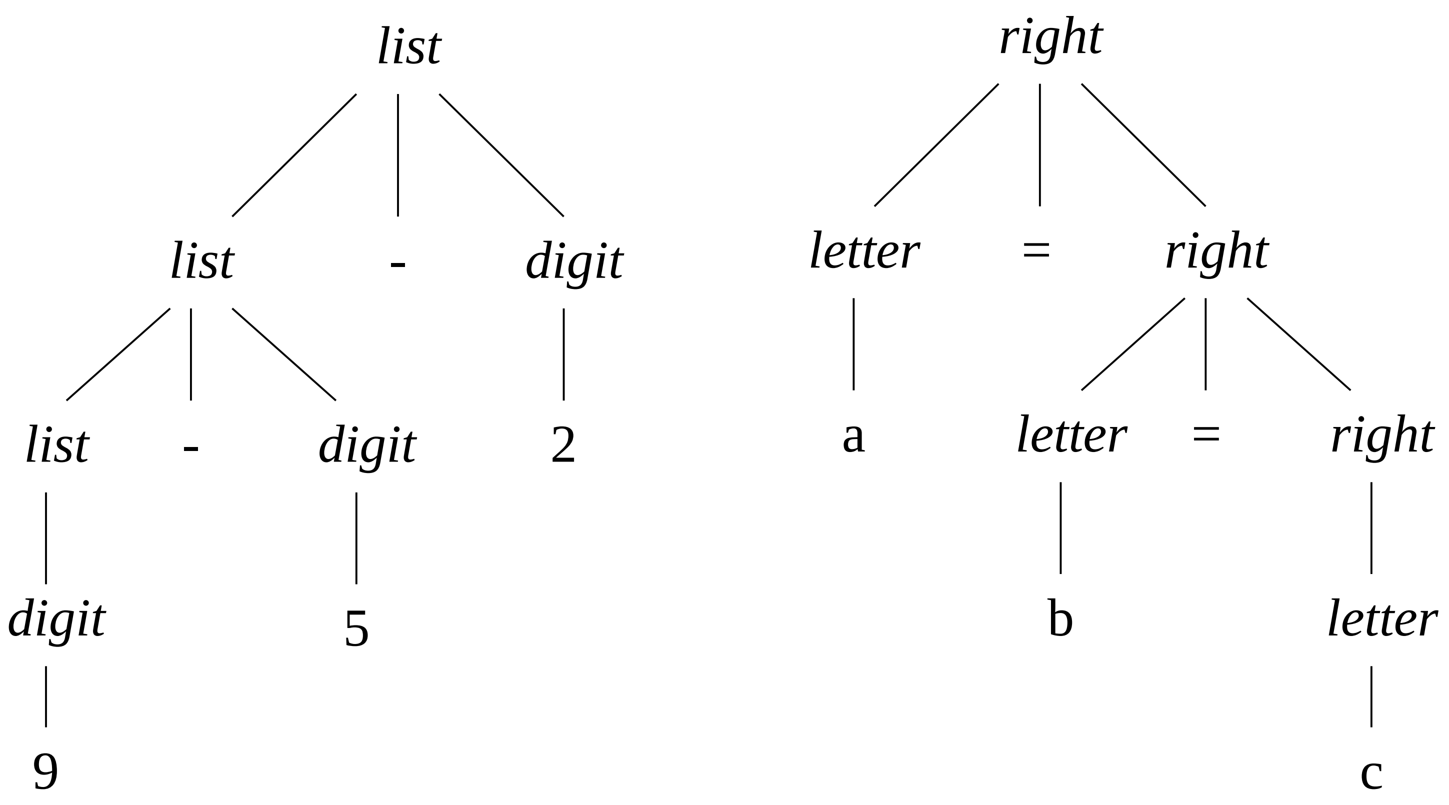

Fig 2.4. Parse trees for left- and right- associative operators.

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 of + and * do not resolve this ambiguity. For this reason, we need to know the relative precedence of operators when more than one kind of operator is present.

$expr \longrightarrow expr + term$

$\quad |\ expr - term$

$\quad |$ term

The resulting grammar is therefore

expr → *expr* + *term* | *expr* – *term* | *term*

term → *term* * *factor* | *term* / *factor* |

factor

factor → **digit** | (*expr*)

Syntax of statements. Keywords allow us to recognize statements in most languages. All Pascal statements begin with a keyword except assignments and procedure calls. Some Pascal statements are defined by the following (ambiguous) grammar in which the token id represents an identifier.

$$
\begin{aligned}
\text{stmt} \rightarrow\ & \text{id} \ := \ \text{expr} \\
& |\ \text{if} \ \mathit{expr} \ \text{then} \ \mathit{stmt} \\
& |\ \text{if} \ \mathit{expr} \ \text{then} \ \mathit{stmt} \ \text{else} \ \mathit{stmt} \\
& |\ \text{while} \ \mathit{expr} \ \text{do} \ \mathit{stmt} \\
& |\ \text{begin} \ \mathit{opt\_stmts} \ \text{end}
\end{aligned}
$$

2.3 SYNTAX-DIRECTED TRANSLATION

To translate a programming language construct, a compiler may need to keep track of many quantities besides the code generated for the construct. For example, the compiler may need to know the type of the construct, or the location of the first instruction in the target code, or the number of instructions generated. We therefore talk abstractly about attributes associated with memory location. An attribute may represent any quantity, e.g., a type, a string, a memory location, or whatever.

In this section, we present a formalism called a syntax-directed definition for specifying translations for programming language constructs. A syntax-directed definition specifies the translation of a construct in terms of attributes associated with its syntactic components. In later chapters, syntax-directed definitions are used to specify many of the translations that take place in the front of a compiler.

Postfix Notation

The *postfix notation* for an expression *E* can be defined inductively as follows:

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' and E_2' the postfix notations for E_1 and E_2, respectively.
3. If E is an expression of the form (E_1), then the postfix notation for E_1 is also the postfix notation for *E*.

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. For example, the postfix notation for (9-5)+2 is 95-2+ and the postfix notation for 9-(5+2) is 952+ -.

Adapting the Translation Scheme

The left-recursion elimination technique sketched in fig. 2.18 can also be applied to productions containing semantic actions. We extend the transformation in Section 5.5 to take synthesized attributes into account. The technique transforms the productions

$$A \rightarrow A\alpha \mid A\beta \mid \gamma \text{ into}$$

$$A \rightarrow \gamma R$$

$$R \rightarrow \alpha R \mid \beta R \mid \varepsilon$$

When semantic actions are embedded in the productions, we carry them along in the transformation. Here, if we let *A* = *expr*, α = + *term* { *print*('+') }, β = - *term* { *print*('-') },and γ = *term*, the transformation above produces the translation scheme(2.14). The *expr* productions in Fig. 2.19 have been transformed into the productions for *expr* and the new nonterminal rest in (2.14). The productions for *term* are repeated from Fig. 2.19. Notice that the underlying grammar is different from the one in Example 2.9 and difference makes the desired translation possible.

expr → *term rest*

rest → + *term* { *print*(‘+’) } *rest* | - *term* { *print*(‘-‘) } *rest* |ε

term → *0* { *print*(‘*0*’) }

term → *1* { *print*(‘*1*’) }

… …

term → *9* { *print*(‘*9*’) }

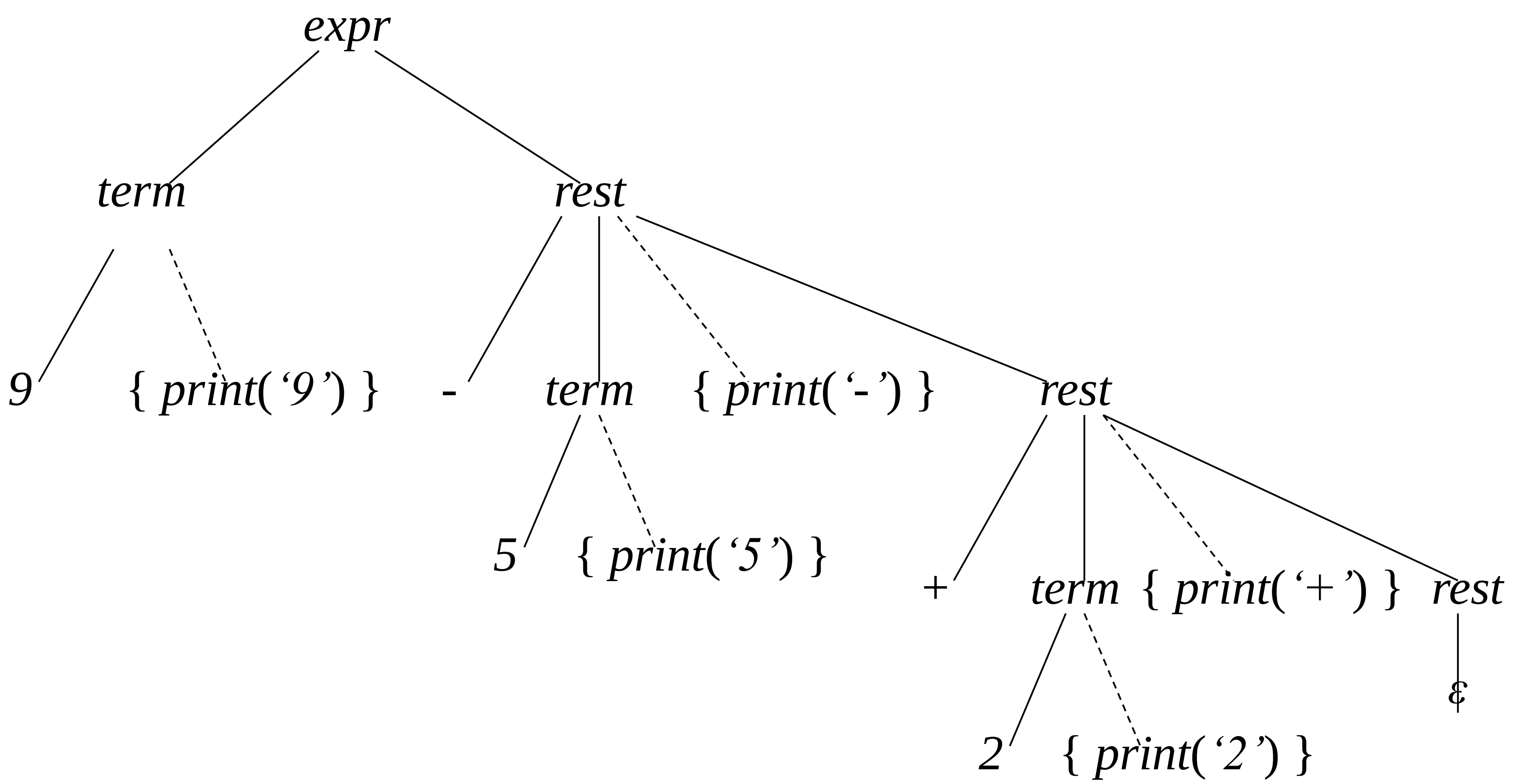

Fig. 2.21 Translation of 9-5+2 into 95-2+.

As another example, function rest( ) uses the first production for *rest* in (2.14) if the lookahead symbol is a plus sign, the second production if the lookahead symbol is a minus sign, and the production *rest* $\rightarrow \varepsilon$ by default. The first production for *rest* is implemented by the first if-statement in Fig. 2.22. If the lookahead symbol is +, the plus sign is matched by the call match(‘+’). After the call term( ), the C standard library routine putchar(‘+’) implements the semantic action by printing a plus character. Since the third production for *rest* has ε as its right side, the last else in rest( ) does nothing.

Arithmetic Instructions

The abstract machine must implement each operator in the intermediate language. A basic operation, such as addition or subtraction, is supported directly by the abstract machine. A more complex operation, however, may need to be implemented as a sequence of abstract machine instructions. We simplify the description of the machine by assuming that there is an instruction for each arithmetic operator.

The abstrct machine code for an arithmetic expression simulates the evaluation of postfix representation for that expression using a stack. The evaluation proceeds br processing the postfix representations from left to right, pushing each operand onto the stack as it is encountered. When a k-ary operator is encountered, its leftmost argument is k-1 positions below the top of the stack and its rightmost srgument is at the top. The evaluation applies the operator to the top k values on the stack, pops the operands, and pushes the result onto the atack. For example, in the evaluation of the postfix expression 1 3 + 5 $\ast$, the following actions are performed:

1. stack 1
2. stack 3
3. Add the two topmost elements, pop them, and stack the result 4.
4. stack 5.
5. Multiply the two topmost elements, pop them, and stack the result 20.

The value on top of the stack at the end (here 20) is the value of the entire expression.

In the intermediate language, all values will be integers, with 0 corresponding to false and nonzero integers corresponding to true. The boolean operators and and or require both their arguments to be evaluated.

	INSTRUCTIONS			STACK		DATA	
1	push 5			16		0	1
2	rvalue 2		top→	7		11	2
3	+					7	3
4	rvalue 3					…	4
5	*	←pc					
6	…						

Fig. 2.31 Snapshot of the stack machine after the first four instructions are executed.

L-values and R-values

There is a distinction between the meaning of identifiers on the left and right sides of an assignment. In each of the assignments

```
i := 5;

i := i + 1;
```

the right side specifies an integer value, while the left side specifies where the value is to be stored. Similarly, if p and q are pointers to characters, and p ↑ := q ↑ ;

the right side q ↑ specifies a character, while p ↑ specifies where the character is to be stored. The terms *l-value* and *r-value* refer to values that are appropriate on the left and right sides of an assignment, repectively. That is, *r-values* are what we usually think of as “values,” while *l-values* are locations.

Stack Manipulation

Besides the obvious instruction for pushing an integer constant onto the stack and popping a value from the top of the stack, there are instructions to access data memory:

push *v*	push *v* onto the stack
rvalue *l*	push contents of data location *l*
lvalue *l*	push address of data location *l*
pop	throw away value on top of the stack
:=	the *r-value* on top is placed in the *l-value* below it and both are popped
copy	tush a copy of the top value on the stack

lvalue day	push 2
push 1461	+
rvalue y	push 5
*	div
push 4	+
div	rvalue d
push 153	+
rvalue m	:=
*	

Fig 2.32 Translation of day := (1461 * y) div 4 + (153 * m + 2) div 5 + d.

These remarks can be expressed formally as follows. Each nonterminal has an attribute t giving its translation. Attribute lexeme of id gives the string representation of the identifier.

stmt → id := *expr*

{ *stmt.t* := ‘lvalue’ || id.*lexeme* || *expr.t* || ‘:=’ }

Control Flow

The stack machine executes instructions in numerical sequence unless told to do otherwise by a conditional or unconditional jump statement. Several options exist for specifying the targets of jumps:

1. The instruction operand gives the target location.
2. The instruction operand specifies the relative distance, positive or negative, to be jumped.
3. The target is specified symbolically; i.e., the machine supports lablels.

With the first two options there is the additional possibility of taking the operand from the top of the stack.

We choose the third option for the abstract machine because it is easier to generate such jumps. Morever, symbolic addresses need not be changed if, after we generate code for the abstract machine, we make certain improvements in the code that result in the insertion or deletion of instructions.

The control-flow instructions for the stack machine are:

label /	target of jumps to / ; has no other effect
goto /	next instruction is taken from statement with label /
gofalse /	pop the top value; jump if it is zero
gotrue /	pop the top value; jump if it is nonzero
halt	stop execution

Emitting a Translation

The expression translators in Section 2.5 used print statements to incrementally generate the translation of an expression. Similar print statements can be used to emit the translation of statements. Instead of print statements, we use a procedure *emit* to hide printing details. For example, *emit* can worry about whether each abstract-machine instruction needs to be on a separate line. Using the procedure *emit*, we can write the following instead of (2.18):

stmt →	if	
	expr	{ *out* := *newlabel*; *emit*('gofalse' , *out*); }
	then	
	$stmt_1$	{ *emit*('label' , *out*);}

When semantic actions appear within a production, we consider the elements on the right side of the production in a left-to-right order. For the above production, the order of actions is as follows: actions during the parsing of *expr* are done, *out* is set to the label returned by *newlabel* and the gofalse instruction is emitted, actions during the parsing of $stmt_1$ are done, and, finally, the label instruction is emitted. Assuming the actions during the parsing of *expr* and $stmt_1$ emit the code for these nonterminals, the above production implements the code layout of Fig. 2.33.

stmt $\rightarrow$ if *expr* then $stmt_1$	{ *out* :=	*newlabel*	
	stmt.t :=	*expr.t* ||	
		'gofalse' *out* ||	(2.18)
		$stmt_1.t$	
		'label' *out* }	

IF		WHILE
code for *expr*		label test
gofalse out		code for *expr*
code for $stmt_1$		gofalse out
label out		code for $stmt_1$
		goto test
		label out

Fig. 2.33. Code layout for conditional and while statements.

```
procedure stmt
var test,out: integer;   /* for labels */
begin
        if lookahead = id then begin
                emit('lvalue',tokenval); match(id); match(':='); 
expr
        end
        else if lookahead = 'if' then begin
                match('if');
                expr;
                out := newlabel;
                emit('gofalse',out);
                match('then');
                stmt;
                emit('label',out)
        end
                /* code for remaining statements goes here */
        else error;
end
```

Fig. 2.34 Pseudo-code for translating statements.

Example 2.10 The lexical analyzer in Section 2.7 contains a conditional of the form:

If t = blank or t = tab then …

If t is a blank, then clearly it is not necessary to test if t is a tab, because the first equality implies that the condition is true. The expression

$expr_1$ **or** $expr_2$

can therefore be implemented as

if $expr_1$ then true else $expr_2$

The reader can verify that the following code implements the **or** operator:

```
    code for $expr_1$
    copy                    /* copy value of $expr_1$ */
    gotrue out
    pop                          /* pop value of $expr_1$ */
code for $expr_2$
    label out
```

Recall that the gotrue and gofalse instructions pop the value on top of the stack to simplify code generation for conditional and while statements. By copying the value of $expr_1$ we ensure that the value on top of the stack is true if the gotrue instruction leads to a jump.