



## Chapter 8

# Expressions and Evaluation

---

---

### Overview

This chapter introduces the concept of the programming environment and the role of expressions in a program. Programs are executed in an environment which is provided by the operating system or the translator. An editor, linker, file system, and compiler form the environment in which the programmer can enter and run programs. Interactive language systems, such as APL, FORTH, Prolog, and Smalltalk among others, are embedded in subsystems which replace the operating system in forming the program-development environment. The top-level control structure for these subsystems is the Read-Evaluate-Write cycle.

The order of computation within a program is controlled in one of three basic ways: nesting, sequencing, or signaling other processes through the shared environment or streams which are managed by the operating system.

Within a program, an expression is a nest of function calls or operators that returns a value. Binary operators written between their arguments are used in infix syntax. Unary and binary operators written before a single argument or a pair of arguments are used in prefix syntax. In postfix syntax, operators (unary or binary) follow their arguments. Parse trees can be developed from expressions that include infix, prefix and postfix operators. Rules for precedence, associativity, and parenthesization determine which operands belong to which operators.

The rules that define order of evaluation for elements of a function call are as follows:

- Inside-out: Evaluate every argument before beginning to evaluate the function.

- Outside-in: Start evaluating the function body. When (and if) a parameter is used in the body, evaluate the corresponding argument. There are two variants, as follows:
  1. Call-by-name: An argument expression is evaluated each time the corresponding parameter is used. This is inefficient and may result in different values for the argument at different times.
  2. Call-by-need: An argument expression is evaluated the first time its parameter is used, and the result is stored for future use.

Inside-out evaluation has been used for most block-structured languages; outside-in has recently been efficiently implemented, in the form of call-by-need, for functional languages. Some languages, however, use a mixture of strategies. Call-by-value-and-return is used in order to facilitate concurrent, multitasked operations.

---

œ

## 8.1 The Programming Environment

Each program module is executed in some environment. In these days that environment is rarely the bare machine; almost all machines run under an operating system (OS). The OS forms an interface between the hardware, the user, and the user's program, and it creates the environment in which the user's program runs.

### Compiled Language Systems

When working with a compiler, a programmer either works in the environment provided by the OS or uses a *development shell*, often provided with the translator, to tailor the OS environment to current needs. An editor, linker, file system, and the compiler itself are included in this environment. Together they enable the programmer to enter, compile, and link/load programs. The programmer can give OS commands to execute his or her own program or others, and when execution is done, control returns to the OS or to the shell.

If the OS supports multitasking (making it an M-OS) there may be other programs, or tasks, in this context, and the M-OS will mediate between them. It supervises messages, controls signals, and manages shared memory. A task might run indefinitely, putting itself to sleep when it needs information from another task and waking up in response to an interrupt generated when that other task supplies the information.

### Interactive Language Systems

The category of interactive languages includes APL, FORTH, the functional languages, Prolog, Smalltalk, and many others. These languages are embedded in subsystems that take the place of the OS in forming the user's program development environment. Many of these contain independent file systems and editors, and a separate interactive control cycle.

Access to the enclosing OS is almost always restricted. For example, just one language is generally supported, with no possibility of interaction with subprograms written in a different language. Also, even if the OS supports multitasking, the language subsystem may not give access to it. It also may not support use of an outside text editor and may restrict file system access, allowing fewer kinds of operations than the OS.

These language subsystems implement a top-level control structure known as the *Read-Evaluate-Write (REW) cycle*. When the programmer enters this environment she or he sees a prompt on the terminal and the interpreter is in control, waiting for programmer input. The programmer enters expressions which the interpreter reads and evaluates. It then writes the value of the expression, if any, and a prompt on the user's terminal. The user initiates all actual execution.

Three kinds of items are entered to form a complete program: (1) definitions of objects (name-value pairs), (2) definitions of functions, with parameter names, and (3) expressions or function calls with actual arguments. Items of type (1) are handled by the interpreter. Names are entered, with their value bindings, into the program environment.

For items of type (2), the interpreter will call a *half-compiler*, which reads the expression, lexes it, parses it, and converts it to an internal form which represents the semantics of the expression in a form that is convenient for later evaluation. This form might be a computation tree or a linear list of actions in postfix order. (A full compiler would also generate native machine code.)

For items of type (3), the interpreter evaluates the expression and leaves the result on the stack. During execution, this expression may call functions that are defined in its program environment. After returning from the top-level call, the value left on the stack is generally printed out by the REW cycle for inspection by the programmer.

## 8.2 Sequence Control and Communication

A program specifies a set of actions that must be done with a set of representations of objects. In some cases the order of these actions is not important, but generally order matters, particularly when interaction with the outside environment (user or file system) is involved. The words the programmer writes in the program to invoke this ordered series of actions must be arranged in a way that specifies this order, or the programmer might specify that the order is not important. In any case, the connection between the way the programmer writes things and the order of resulting outputs must generally be predictable, easy to control, and stable.

Three basic ways have been developed for specifying the order in which computation will happen:

- Nesting of expressions, lexically or symbolically.

- Sequencing of statements and procedure calls.
- Signaling to other tasks.

### 8.2.1 Nesting

Individual symbols are the lowest-level construct in any programming language. These are formed into expressions (see Section 8.3) which are contained in larger expressions and, in the end, organized into some larger unit. In some languages these units are statements, written in sequence. In others they are function definitions, written sequentially or in nested form.

#### Lexical Nesting by Parenthesization

Each function call is a function name with expressions as arguments, which might themselves be function calls. When a call does contain another function call, we say that it is a *nested expression*.

A nested expression can be evaluated using only a stack for intermediate results, and without use of assignment or nonstack variables.<sup>1</sup> Here we consider in detail how the stack works. Before a function call can be evaluated, its argument expressions must be evaluated. Evaluation of arguments thus proceeds inward through the nest of function calls, until there are no more enclosed expressions. As each argument expression is evaluated, its value is left at the top of the stack. When all arguments for a function call have been evaluated, the function can be called. The values on the stack will be bound to parameter names within the function. The function will run, compute an answer, and leave it on the stack for later use by an enclosing expression.

Parameter binding enables communication of information in and out through the hierarchical nest of function calls. During execution, the position on the stack where an expression's results will be stored corresponds directly to the position in which the programmer wrote the subexpression in the program, and to the position in the program of the expression that will use the results.

#### Symbolic Nesting by Declaration

A function call with function calls in its parameter list, which, in turn, have embedded function calls can become quite incomprehensible. This is so even if indentation and spacing are used to clarify the structure of the whole unit.

Declarations are used (among other reasons) to reduce the apparent nesting depth of expressions. A local variable declaration permits the programmer to define a name for the result of a subexpression. A subroutine declaration permits one to isolate and name some part of the program code. These defined names can then be used to build routines with a lower apparent degree of nesting, which are easier for a human to cope with than a deeply nested program. Modularizing a program by defining local names and short functions helps a programmer create a correct program more easily.

---

<sup>1</sup>All the familiar older languages use a stack. Some of the modern functional languages use a more complex data structure that avoids evaluating arguments that are not actually used in the function code.

Programs in functional languages tend to be highly modular because of this. Since parameter binding, not assignment, is used to give names to computed values, a new function must be defined whenever the programmer wishes to name an intermediate result. Further, when a nested expression has many nesting levels, humans tend to become confused about the meaning of the code. Subroutines are used to minimize the depth of nesting in a single program unit.

### 8.2.2 Sequences of Statements

A *statement* is an expression that leaves no return value on the stack and thus cannot be used as the argument to a function. Statements are executed for their *side effects*, not for their return values. Side effects include changing the value in a storage object and changing the status of anything in the program environment, the shared environment, or the streams. To be useful, a statement must use one of these channels to affect the world, since it does not change the program stack. Either it must alter program memory or produce input/output activity.

A *procedural language* is one in which a program is a sequence of statements. Procedural languages commonly contain statement forms to do the following tasks:

- **Assignment statement:** Changes the value of a variable.
- **Binding statement:** Enters a named value into the symbol table.
- **Procedure call:** A call on a user-defined function that returns no result.
- **Input statement:** Changes the values of some variables.
- **Output statement:** Changes the state of the external world.
- **Control statement:** A compound of other statements which implements a loop or conditional structure.

Procedural and functional languages both contain expressions. Within an expression, one function communicates its result directly to the enclosing function without using the program environment. This is done by taking arguments from the program stack and leaving results on it.

In *functional languages*, the expression is the outermost control structure, and the stack is the only form of communication between the parts of a program. Modern functional languages do not support data objects in the program environment. In contrast, in *procedural languages* (ALGOL, FORTRAN, Pascal, APL, etc.) all expressions eventually become part of some statement, and a program body is a series of statements, written top to bottom in the order they are to be executed. Communication between one statement and another can go through variables in the program environment; each statement finds its inputs in a set of variables and leaves its results in variables for possible later use [Exhibit 4.9].

### 8.2.3 Interprocess Sequence Control

When all computers had small memories and single, slow processors, it was adequate to design languages for writing independent programs. Such a program would follow one control path during execution, which was independent of all other programs. Its memory was its own, and it could interact with the world only through input and output. But as multiprocessor systems and networks have become common, we have had to expand our area of consideration from a single, isolated program (or *process* or *task*) to a group of asynchronous processes that are actively interacting and communicating.

A single process, no matter how large, can communicate through global variables and parameters. If that process were to be compiled in several modules, each module could have external symbols (variables and functions) that would be defined in other modules. In this case, the linking loader would connect each external reference from one module with the definition of that object or function in another module, creating a single, connected unit that could communicate through variables and subroutine calls.

Communication for two separate interacting processes is quite different. No compiler or linking loader links up external references in one process to symbols defined in the other. In a modern system, one process may not know the location of another process with which it is interacting. All communication must take place through the operating system or systems which host the two processes, or through the file system. In terms of our formal model, communication must be through the shared environment or through streams, and both of these areas of storage are managed by the OS. The specific kind of communication possible depends on the operating system host.

One common mechanism is message passing. A message is a string of bytes that is written into an OS buffer and either broadcast to all processes or addressed to one particular process whose process-ID is known to the sender. To send a message, a program calls the operating system's message sending routine. To receive a message, a process makes a corresponding call to the operating system, signaling that it is ready to process a message either from anyone or from a particular sender. The message is then copied into a buffer belonging to the receiver.

Another common communication mechanism is the semaphore. This is a storage location in protected OS memory, accessible to a program only through system calls. Semaphores come in many varieties, but they basically function as pigeonholes that contain one of two signals meaning either "wait until I'm ready" or "go ahead". These are commonly used to control and synchronize access to files and buffers that are shared among interacting processes.

## 8.3 Expression Syntax

An expression is a nest of function calls (or operators, in some languages) that returns a value. The syntax for writing an expression varies among languages. Exhibit 8.1 shows examples of function calls drawn from several languages which illustrate the various kinds of syntax in common use. In all cases, the action represented is adding one to an integer, *A*, and returning the result.

---

**Exhibit 8.1. Syntax for expressions.**

Lambda calculus		
An application	( S A )	– Compute the successor of A.
LISP		
An expression	( + A 1 )	– Add the value of A and 1.
An expression	( 1+ A )	– Add 1 to the value of A.
Pascal		
A function call	succ( A )	– Compute the successor of A.
Use of an operator	A + 1	– Add the value of A and 1.
APL		
An expression	A + 1	– Add the value of A and 1.
(APL has no other syntax for function calls.)		

---

**8.3.1 Functional Expression Syntax**

In the most basic syntax, one writes an expression by writing a function name, with its actual arguments, enclosed in parentheses. In lambda calculus and in LISP the function name is *inside* the left paren [Exhibit 8.1]; in most languages it is *outside* the left paren. We will call the first variant *lambda calculus function-call syntax* and the latter variant *normal function-call syntax*.

In modern languages, an actual argument is an expression. It is evaluated, in the context of the calling program, either before control is transferred to the function or during evaluation of the function. The value of each argument expression is used to initialize the corresponding formal parameter.

There are two ways that the correspondence between arguments and parameters can be specified: positionally or by keyword. With positional notation, which is supported in virtually every language, the first argument value initializes the first parameter name, the second argument corresponds to the second name, and so on. To call a function positionally one must know the number and order of the parameters, and know the type and semantics of each, but not its declared name.

A few languages support a second parameter correspondence mechanism, sometimes called *correspondence-by-keyword*, or *named parameter association*. To use named association, the argument expression in the function call is preceded by the dummy parameter name as defined in the procedure definition. This mechanism has the advantage that the arguments in a call may be written in any order, and it is useful for implementing functions with optional arguments (Section 9.1.1). It is syntactically more complex, though, than positional correspondence, and more complex to implement. Further, the parameter names, which are otherwise arbitrary and local to the function, must be made known to all users.

Some confusion of terminology has developed among different programming languages. In LISP “+” is called a “function” and is written with lambda calculus function-call syntax. In Pascal it

is called an “operator” and written with *infix operator syntax*; that is, each operator is written between its operands. In APL “+” is called a “function” but written in infix operator syntax.

Operators are just a syntactic variant of functions; the underlying semantics are the same. Both denote an action which, when applied to some objects (the arguments), will produce another object (the function result). Some languages include only the function call syntax; some include only the operator syntax; some include both. There are two minor differences between operators and functions. First, functions may have any number of arguments, while operators are almost universally limited to one or two. Second, in languages that include both functions and operators, it is generally not possible to define new operators; all newly defined verbs are functions.

### 8.3.2 Operator Expressions

An operator with two arguments is called *binary* or *dyadic*. It is generally called using infix operator syntax; that is, the operator is written between its arguments. A single-argument operator, known as *unary* or *monadic*, is generally called using *prefix syntax*; that is, the operator is written before its argument. Some languages also have monadic postfix operators, which are written after the operand. C has a diverse collection and even has one two-part operator with three operands!

Any expression specifies a *computation tree*, which is the *meaning* of the source expression. The task of a compiler (specifically, the parsing phase of a compiler) is to take the source syntax of an expression and produce the computation tree, or *parse tree*, that it denotes.

For normal function-call syntax, producing a parse tree is an easy task. Each function call corresponds to a node in the tree, and each argument corresponds to a child of that node. If some argument position contains a nested function call, then that child is itself a node with children [Exhibit 8.2].

Developing a parse tree from an expression that includes a combination of infix, prefix, and postfix operators is much more complex. Two sets of rules (the rules for precedence and the rules for associativity), plus explicit parenthesization, determine which operand belongs to which operator.

#### Parenthesization

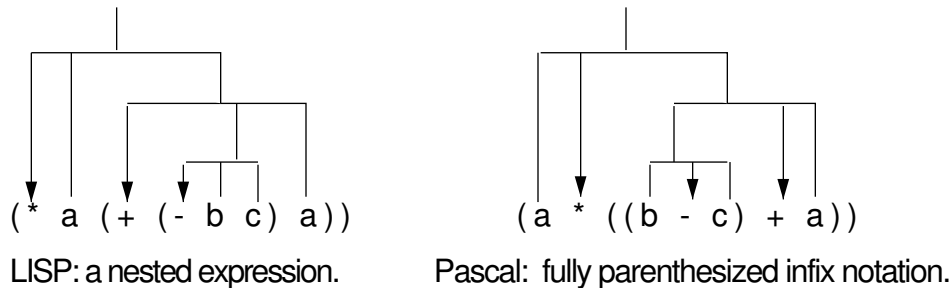
Languages that have infix operators permit the programmer to use explicit parentheses to surround an operator and its one or two operands. Like an expression written using nested function calls, a fully parenthesized infix expression explicitly and unambiguously specifies which operands belong to each operator. Exhibit 8.2 shows the parse trees derived from a LISP expression that is a nest of function calls, and the analogous Pascal expression, written with operators. The trees are identical except for the order of the operator and the first operand.

#### Postfix and Prefix: Unambiguous Unparenthesized Expression Syntax

The parentheses in LISP, or in a fully parenthesized infix expression, group each function or operator with its operands. This permits one to draw a parse tree with no further information and no

**Exhibit 8.2. Nested function calls and parenthesized infix expressions.**

An expression written in LISP, using functional notation, and in Pascal, using fully parenthesized infix notation. Both denote the same computation tree.



knowledge of the meaning of the operator. In a language where each operator has a fixed number of operands, these parentheses are nice but not necessary. They guide the eye and the parser, and permit us to check whether each operator has been written with the correct number of operands. There are other ways, however, to indicate which operands belong to which operator.

We can actually eliminate the parentheses if we are willing to restrict each operator to exactly one meaning and require that its number of operands be fixed. (This eliminates the possibility of an ambiguous operator, such as “-” in FORTRAN, which exists in both unary and binary forms.)

With this restriction, there are two forms for expressions, called *prefix order* and *postfix order*, that unambiguously specify the computation tree without the use of parentheses or precedence. In a prefix expression, each operator is written before its operands, like a LISP expression without the parentheses [Exhibit 8.3].

In *postfix order*,<sup>2</sup> an operator is written after its operands. The FORTH language and Postscript<sup>3</sup> use postfix notation exclusively. This order corresponds to the actual order of machine instructions needed to compute the value of an expression using a stack machine. The evaluation process is illustrated in Exhibit 8.4, and works like this:

<sup>2</sup>Also known as reverse Polish notation.

<sup>3</sup>A language for controlling graphic output devices. Adobe Systems [1985].

**Exhibit 8.3. Unambiguous ways to write an arithmetic expression.**

This is how the expression  $(a * ((b - c) + a))$  would be written in prefix and postfix notations.

*Prefix:*    \* a + - b c a    This is like LISP, but without parentheses. The scope of each operator is the two operands or subexpressions following it.

*Postfix:*    a b c - a + \*    This is used in FORTH. The scope of each operator is the two operands or subexpressions preceding it.

**Exhibit 8.4. Evaluation of a postfix expression using a stack.**

We evaluate the expression `a b c - a + *` from Exhibit 8.3. Assume the stack is initially empty and the names `a`, `b`, and `c` are bound to constant values: `a = 5`, `b = 4`, `c = 7`.

Evaluation Step	Stack Contents (top is at right)
<code>a</code>	5
<code>b</code>	5, 4
<code>c</code>	5, 4, 7
<code>-</code>	5, -3
<code>a</code>	5, -3, 5
<code>+</code>	5, 2
<code>*</code>	10

- Evaluate the expression left-to-right.
- As each symbol is encountered, put its current value on the stack.
- For each operator, take two values off the stack and execute the operation. Put the result back on the stack.

One might ask why LISP uses both prefix order and parentheses if prefix order is unambiguous without the parentheses. The parentheses in LISP permit its functions and operators to take a varying number of operands [Exhibit 8.5, line 6]. Several primitive LISP functions can have variable-length argument lists, including one at the core of the language. The LISP conditional expression, `COND`, is a variable-argument function.

Note the similarity between form 4, prefix notation, and form 5, written in LISP. If functions in LISP were restricted to a fixed number of parameters, the parentheses would not be necessary. But LISP has variable-argument functions, including `+`. In form 6, the expression is rewritten using the variable-argument form of `+`, and parentheses are used to delimit the scope of the `+`. It can be cogently argued that this is the clearest way to write this expression.

Form 6, in LISP, strongly resembles form 7, in COBOL, which adds a variable number of items to the last item. Reserved words are used in COBOL instead of parentheses to bracket the variable-length parameter list, but the principle is the same; it is possible to parse a language that has variable-length parameter lists as long as the lists are bracketed somehow.

**Precedence**

The early computer languages FORTRAN and ALGOL-60 were designed for use by scientists and engineers who were accustomed to traditional mathematical notation for expressions.<sup>4</sup> Engineers wanted a “natural” way to write formulas in their programs; that is, they wanted the formulas

<sup>4</sup>This is evident from their names. FORTRAN was derived from FORMula TRANslator, and ALGOL from ALGORithmic Language.

**Exhibit 8.5. Bracketing used to permit variable-argument expressions.**

Seven forms are given of an expression which sums four items. In the first four forms the + operator must be written three times, because it is “built into” these languages that the + operator takes two operands.

Forms 5 and 6 show an analogous expression written in LISP. Form 5 has two operands for each +, but version 6 takes advantage of the fact that + can accept a variable number of arguments.

Form 7 is a statement, not a function: it modifies the value of C rather than returning a value. Note that it uses the reserved words ADD and TO, rather than parentheses, to bracket the variable-length list.

1. Fully parenthesized infix:	(( (a + b) + 3) + c)	Pascal or C
2. Infix without parentheses:	a + b + 3 + c	Pascal or C
3. Postfix notation:	a b + 3 + c +	FORTH
4. Prefix notation:	+ + + a b 3 c	
5. Use of + in binary form:	(+ (+ (+ a b) 3) c )	LISP
6. Multiple arguments for +:	(+ a b 3 c)	LISP
7. A statement that modifies C:	ADD A B 3 TO C.	COBOL

in their computer programs to look as much as possible like formulas on paper. Engineers and mathematicians have never considered full parenthesization, prefix order, or postfix order to be “natural” ways to express a formula. The familiar way of writing expressions using operator precedence was used in FORTRAN and ALGOL-60 because it had been in use in mathematics for many years. Since then precedence expressions have been implemented in all their descendents.

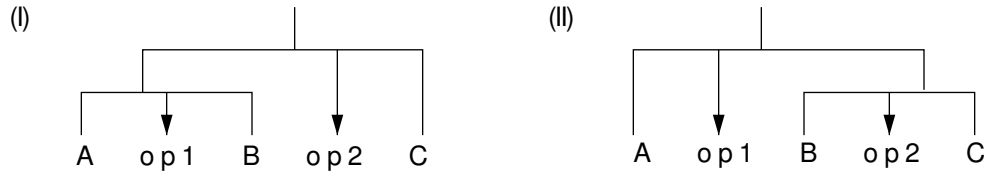
Without parentheses or additional parsing rules, infix notation is ambiguous. The symbols written on the page do not indicate which operands belong with each operator. Consider the expression in Exhibit 8.6, where *op1* and *op2* are two unspecified binary operators. Two parses of this expression are possible and correspond to the two trees shown. The rule of operator precedence was adopted from mathematics to disambiguate the nonparenthesized parts of infix expressions. The precedence rule determines which parse is correct, and thus whether the scope of *op2* includes the result of *op1* or vice versa.

In a precedence language, each operator is assigned a precedence level. These levels can be arbitrary integers; only their order matters. The standard precedence rule can now be stated:

- If  $prec_{op1} > prec_{op2}$  then the meaning is parse tree I.
- Else if  $prec_{op1} < prec_{op2}$  then the meaning is parse tree II.
- Else  $prec_{op1} = prec_{op2}$ . Use the rule of associativity to determine which parse is correct.

**Exhibit 8.6. Infix notation is ambiguous without precedence.**Source expression:  $A \text{ op1 } B \text{ op2 } C$ 

Possible parse trees:

**Associativity**

Associativity is used as a tie-breaking rule in languages with precedence. It is also used in APL as the only added parsing rule to disambiguate infix expressions. Associativity governs the choice of parse trees when there are consecutive binary operators with equal precedence. Associated with each precedence class is an associativity direction, either left-to-right or right-to-left. All operators with the same precedence must have the same associativity. This direction is used as follows:

- If the associativity of  $op1$  and  $op2$  is left-to-right then the meaning of the expression is parse tree I.
- Else (the associativity of  $op1$  and  $op2$  is right-to-left) the meaning of the expression is parse tree II.

The associativity of each operator is determined by the language designer and is usually chosen to seem natural to mathematicians. For example, in C, the associativity of  $-$  (subtraction) is left-to-right, but the associativity of  $=$  (assignment) is right-to-left. The programmer can write:  $X=Y=Z-1-X$  to compute the value  $((Z-1)-X)$  and store the answer in both  $Y$  and  $X$ .

In APL there were so many operators that the language designer evidently felt that establishing precedence classes for them would cause more confusion than help. Therefore, although APL functions are written with infix notation, only an associativity rule, right-to-left, is used for disambiguation.

**8.3.3 Combinations of Parsing Rules**

Existing languages use varying combinations of rules to define the meaning of an expression. Very different effects can be achieved by combining these few simple rules, as illustrated by Exhibits 8.7, 8.8, and 8.9. Exhibit 8.7 gives brief summaries of the parsing rules in some of the languages previously mentioned. Exhibit 8.8 shows how the arithmetic expression  $((b * c) / ((a + 1) * (b - 2)))$  would be written in each language using as few parentheses as possible. Exhibit 8.9 shows the

---

**Exhibit 8.7. Varying rules for parsing expressions.**

Summaries of the parsing rules are given below for several languages.

- FORTH: postfix notation, no precedence, no associativity, no parentheses. Operators are executed left-to-right. Parameter lists are not delimited at all.
  - APL: infix notation for all functions, no precedence, right-to-left associativity. Parentheses may be used to override associativity.
  - Pascal: infix notation for operators, with precedence and left-to-right associativity. Parentheses may be used to override precedence or associativity.
  - C: infix, prefix, and postfix operators, with precedence. Left-to-right or right-to-left associativity, depending on the operator. Parentheses may be used to override precedence or associativity.
- 

---

**Exhibit 8.8. Varying syntax for the use of operators.**

Each expression below illustrates the rules summarized in Exhibit 8.7 and will translate into the computation tree in Exhibit 8.9. A, B, and C are integer variables.

- FORTH:  $B @ C @ * A @ 1 + B @ 2 - * /$

Note: A variable name in FORTH is interpreted to mean the address of the variable. The symbol @ must be used explicitly to fetch the value from that address.

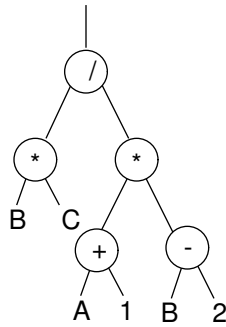
- APL:  $(B \times C) \div (A + 1) \times B - 2$
- Pascal:  $B * C / ((A + 1) * (B - 2))$

Relevant precedence values, with highest precedence listed first:  $( *, / ) > ( +, - )$ .

---

**Exhibit 8.9. A computation tree.**

This tree represents the computation  $((b * c) / ((a + 1) * (b - 2)))$ . Each leaf represents the value of an integer variable, and each interior node represents an operation to be applied to the values returned by its children. The expressions in Exhibit 8.8 will all translate into this computation tree.



computation tree into which all of these expressions will be translated. Note the similarities and contrasts in these languages, especially the contrast in placement of parentheses.

The language syntax, parentheses, precedence, and associativity all help the translator to parse a program and produce a parse tree. After the parse tree is produced, it is interpreted, or code is generated that will later be executed to evaluate the parse tree. It is perhaps surprising, but true, that the same parse tree can have very different meanings in different languages. There are still semantic issues to be resolved *after parsing* that determine the meaning of the expression.

## 8.4 Function Evaluation

A function call (like a lambda calculus application) consists of a literal function or a function name, with formulas representing actual arguments, and an indication that the function is to be applied to those arguments. In many languages, the actual arguments may contain function calls, producing a hierarchy, or nest, of calls. The order and method of evaluation of such a nest depends on the programming language and sometimes also on the compiler writer.

### 8.4.1 Order of Evaluation

Two very different evaluation rules are in use which govern the order of evaluation of the elements of a function call:

1. *Inside-out evaluation:* Evaluate every actual argument before beginning to evaluate the function itself. In a given language, arguments might be evaluated left-to-right, right-to-left, or in an order determined by the compiler writer.

---

**Exhibit 8.10. The substitution rule.**

The “substitution rule” of lambda calculus defines the semantics of arguments. It is stated here using the familiar terms “function”, “argument”, and “parameter” instead of the usual lambda calculus terminology.

- Let  $P$  be a parameter for function  $F$ .
- Let  $Exp$ , an expression, be the corresponding actual argument.
- Declarations in  $F$  must not capture unbound variables in  $Exp$ . If some variable,  $V$ , defined in  $F$  has the same name as an unbound symbol in  $P$ , rename it with a unique name, say  $V2$ , and replace all  $V$ s in  $F$  by  $V2$ .
- Now the meaning of  $F(Exp)$  is found by replacing every occurrence of  $P$  in  $F$  by  $Exp$ .

- 
2. *Outside-in evaluation*: Start evaluating the function call first. The first time the value of an argument is needed, evaluate that argument and remember the answer in case that argument is used again.

Rule (1) has been used for LISP, Pascal, ALGOL-60, C, and most other languages designed since block structure was developed for ALGOL-60. Implementations of these languages are based on the use of a stack to store arguments and local variables.

Recently, efficient implementation methods for rule (2) have been developed, leading to a new class of languages called “functional languages” (Miranda, Haskell, etc.). Let us examine the relationship between evaluation order, stacks, and block structure.

The *substitution rule* of lambda calculus says that if an expression,  $E$ , contains a nonprimitive symbol,  $S$ , the meaning of the  $E$  is the same as if each occurrence of the symbol  $S$  were replaced by its definition. This rule applies in the following two situations:

- $S$  is a symbol whose meaning is defined by the user in a declaration.
- $S$  is a parameter of  $E$ , and  $S$  is given a meaning by applying  $E$  to a list of arguments. (The meaning of  $S$  is the expression that forms the argument corresponding to  $S$ .)

This seems to be an obvious definition and clearly something that should be part of the implementation of any programming language. But implementing it directly causes some trouble for two reasons. Let  $Exp$  be the meaning of  $S$ .  $Exp$  may contain *unbound* symbols, that is, symbols whose meaning is not defined within  $Exp$ . We say there is a *name conflict* if one of these unbound symbols also occurs in  $E$ . If we simply substitute  $Exp$  for  $S$  in  $E$ , these unbound symbols will be *captured* and given spurious meanings by bindings declared in  $E$ . Thus symbols involved in name conflicts must be renamed. The substitution rule is stated formally in Exhibit 8.10.

The need for renaming complicates the substitution rule. A second problem is presented by recursion. A recursive definition would lead to an infinite number of substitutions if you actually tried to physically substitute the definition of  $S$  into the expression  $E$ .

At this point we can take a shortcut. To the extent that every copy of  $S$  is identical, a single copy is able to emulate multiple copies. Thus we only need to copy the *variable* parts of  $S$ . We are thereby led directly to making a distinction between “pure code”, which never varies, and “data”, which does. Multiple copies of “pure code” may be implemented efficiently by making one copy and accessing it with a “jump to subroutine” instruction. On the other hand, a correct implementation of the substitution rule implies that multiple copies of data and argument values must exist simultaneously.

The oldest computer languages, FORTRAN and COBOL, use only static storage, with one storage location for each variable or parameter. For this reason neither language can support recursion. When ALGOL-60 was first designed no one knew how to implement it efficiently, because an efficient implementation for multiple copies of local variables had not been invented.

Within a few years, however, a significant insight was gained. No matter how many copies of the variables for a function  $S$  exist, the real computer has only one processor and, therefore, can be actively using only one copy at a time. All that is necessary to implement the substitution rule efficiently is to make *one* copy (the one needed at the moment) of  $S$ 's variables conveniently available and bind them to the local names defined in  $S$ . When a copy of  $S$  is fully evaluated we no longer need its corresponding variables. They can be forgotten, and the names in  $S$  can be bound to some other copy. All is well so long as the *correct* copy of the variables always gets bound to the names.

When function evaluation rule (1) is used, the groups of variables are allocated, used, and deallocated in a last-in-first-out manner. The only set of variables needed at the moment are those for the current function. They can be created at the beginning of function execution and forgotten at the end. The stack of our abstract machine is able to perform all the required allocation, binding, and deallocation operations efficiently. But to use a stack like this we must *always evaluate the arguments to a function before evaluating the function*. At first sight this restriction seems to prohibit nothing of importance, and the evaluation order it requires is intuitively appealing. Actually, it is an important and fundamental restriction, as we shall see.

The *evaluation rule of lambda calculus* specifies that the order of evaluation of parts of an expression is undefined, but that parameters must be evaluated if and when they are used (if not earlier). It can be shown that anything that can be computed using this unconstrained evaluation order can also be computed using outside-in order, but some computable things cannot be computed using inside-out order.

The outside-in evaluation rule is: evaluate as much of an expression as possible before using the substitution rule, then replace a symbolic parameter by its definition. This strategy has one very nice property: a parameter that is never used does not ever have to be evaluated. This can happen when the only references to that parameter are in a part of the program that is skipped over by a conditional.

Outside-in evaluation also has one awkward property: the simple stack implementation for parameters won't work. A different parameter evaluation method named *call-by-need* has been developed which produces an efficient implementation of outside-in evaluation. It avoids both the unnecessary evaluation of unused parameters that is inherent in call-by-value, and the repeated evaluation of the same parameter inherent in call-by-name. It is often referred to as *lazy evaluation*, because all unnecessary parameter evaluation is avoided.

In call-by-need evaluation, every function is evaluated until the point that it refers to one of its parameters,  $P$ . That parameter is then evaluated, and its value is saved. Evaluation then proceeds until the next parameter reference. All future references to  $P$  in the same function will be interpreted to have this saved value.

A programmer can exploit the lazy nature of this evaluation order by using, as arguments, expressions that are erroneous or nonterminating under some conditions. The programmer must then use these parameters in guarded ways, checking for the dangerous conditions and being sure not to evaluate the argument if they occur. Lazy evaluation can also be used to build a useful kind of data structure called an “infinite” list. These lists have a head section that is like an ordinary list, and a tail that is a function that can be evaluated to produce the next item on the list. Evaluating it  $N$  times extends the list by  $N$  elements.

An `if..then..else` conditional is actually a function of three arguments, but a very special kind of function. The intent and meaning of a conditional is that either the second argument or the third, but never both, is evaluated. Thus, by its very nature, a conditional, *must* be evaluated outside-in. All languages use a variant of lazy evaluation for conditional statements and expressions. Thus a programmer can emulate lazy evaluation in any language by using conditionals liberally. Exhibit 12.14 shows the use of `if` in Pascal to emulate the lazy evaluation built into Miranda.

### 8.4.2 Lazy or Strict Evaluation

The most important issue concerning order of evaluation is whether an expression will be evaluated outside-in or inside-out. The modern functional languages are the only ones that apply either rule consistently; they use outside-in evaluation order. Other languages use a mixture of strategies, inside-out for most things, but outside-in for conditionals and sometimes also for Boolean expressions.

Ordinary inside-out evaluation is called *strict evaluation*. An evaluator is strict if it evaluates all the arguments to a function before beginning to evaluate a function. Its opposite is *lazy evaluation*. Evaluation is lazy if subexpressions are evaluated only when necessary; if the outcome of an expression does not depend on evaluation of a subexpression, the subexpression is skipped. In modern functional languages, all function calls are interpreted using call-by-need, a kind of lazy evaluation. Each argument expression is evaluated the first time the function body refers to the corresponding parameter. The resulting value is bound to the parameter name and is available for future use. If an argument is not used, it will not be evaluated.

In general, to evaluate an arithmetic expression one must first evaluate all its subexpressions. (For example, to calculate `a + b` one must first know the values of `a` and `b`.) But Boolean expres-

---

**Exhibit 8.11. Lazy evaluation of AND.**

Value of L	Value of R	Value of L AND R
T	T	T
T	F	F
F	T	F
F	F	F

We see that if L is FALSE, the value of the expression is always FALSE, and R does not need to be evaluated.

---

sions are different. Exhibits 8.11 and 8.12 show that it is sometimes possible to know the outcome of a logical AND or OR operation after evaluating only one of its subexpressions. Thus evaluation of the second subexpression can be skipped and execution time can be saved. Some languages, for example, MAD and C, use lazy evaluation for Boolean expressions (also called *short circuit evaluation*) because it is more efficient.

Boolean expressions in many other languages, for example, Pascal, are evaluated inside-out. Thus parts of a program that could, under some circumstances, cause run-time errors must be enclosed inside a control statement (an IF or a WHILE) which checks the error condition and determines whether or not to execute the code.

For example, in Exhibit 8.13, an extra control statement must be used to check for a subscript-out-of-bounds error before executing the subscripted expression. There are two possible reasons for leaving any search loop: either the key item was found, or the array to be searched was exhausted. We would like to test both conditions in the following WHILE statement:

```
WHILE (scan < 101) and (a[scan] <> key) ...
```

But this would cause the program to “bomb” when `scan` exceeded 100, and `a[101]` was tested. This happens because Pascal uses inside-out evaluation order, and *both comparisons are made before*

---

**Exhibit 8.12. Lazy evaluation of OR.**

Value of L	Value of R	Value of L OR R
T	T	T
T	F	T
F	T	T
F	F	F

We see that if L is TRUE, the value of the expression is always TRUE, and R does not need to be evaluated.

---

---

**Exhibit 8.13. Extra code needed with inside-out evaluation.**

```

TYPE array_type = array[0..100] of char;
FUNCTION search (a:array_type, key:char):integer;
VAR done_flag: boolean;
    scan: -1 .. 101;          (* An integer in the range -1 to 101. *)
BEGIN
    done := FALSE;          (* Flag to control loop exit. *)
    scan := 0;
    WHILE ( scan < 101) and not done DO
        IF a[scan] = key
            THEN done := TRUE          (* Leave loop next time. *)
            ELSE scan := scan + 1;
        IF scan = 101 THEN search := -1 ELSE search := scan;
    END

```

---

*performing the WHILE test.*

To code this function in Pascal, the two tests must be written separately. An extra boolean variable is then introduced to effect exit from the loop when the second condition becomes true. We see that the Pascal version requires an extra control statement and makes exit from a simple loop awkward.

Compare this to the parallel form in Exhibit 8.14 written in C. C uses lazy evaluation for Boolean expressions, with arguments evaluated in a left-to-right order. Further evaluation is aborted as soon as the outcome of the expression is determined. The expression that controls the `while` statement can, therefore, encompass both the subscript-out-of-range test and the search test. The error condition is tested first. If the subscript is too large, the rest of the expression (including the part that would malfunction) is not evaluated. This greatly simplifies construction of the loop.

---

**Exhibit 8.14. Simpler control structures with outside-in evaluation.**

```

int search (char a[], char key);
{
    int scan;
    scan = 0;
    while ( scan < 101 && a[scan] != key ) ++scan;
    if (scan == 101) return -1; else return scan;
}

```

---

**Exhibit 8.15. Expressions whose values are indeterminate in C.**


---

```

int x, z, a[10];
int throwaway (int b, int c) { return c };

x = 3;    z = x * ++x;           /* z could be 9 or 12. */
x = 3;    z = throwaway(++x, x); /* z could be 3 or 4. */
x = 3;    a[x] = x++;           /* 3 is stored in either a[3] or a[4]. */

```

---

**8.4.3 Order of Evaluation of Arguments**

One final design decision is whether to specify that arguments are evaluated left-to-right or right-to-left, or leave the order unspecified. This makes no difference when inside-out evaluation is used and the subexpressions have no side effects. But when expressions can produce output or modify memory, the order of evaluation can determine the outcome of the program.

In the modern functional languages, this order is unspecified. These languages do not implement destructive assignment, and, therefore, there is no problem with side effects *except* where output is concerned. For these situations, the languages provide a way, called *strict evaluation*, to specify an ordered evaluation of all subexpressions.

In languages such as Pascal where functions and operators cannot have side effects, there is also no problem. All side effects (assignment, input, output) are restricted to statements, whose order is clearly specified. It is impossible to tell whether a Pascal compiler evaluates expressions left-to-right or right-to-left.

Finally, there are languages such as C and APL where expressions can have side effects. Assignment, input, and output are all expressions in these languages. In addition, C has an increment operator. The language definition in these cases must clearly specify what the evaluation order is. APL does specify right-to-left.

In C, though, the decision was left to the language implementor, and both right-to-left and left-to-right evaluation are permitted and considered to conform to the standard. It is, therefore, incumbent upon a C programmer to avoid using any variable in an expression whose value is changed by a preceding or following subexpression with a side effect [Exhibit 8.15]. The result of such an expression is called *indeterminate*; that is, it may vary from compiler to compiler, even on the same hardware.

**Exercises**

1. What is the programming environment? How is it provided?
2. What is the Read-Evaluate-Write cycle? How is it implemented?
3. What are the three basic methods that specify order of computation?

4. Explain how an expression can be nested.
5. How is a stack used in evaluating a nested expression?
6. What is an expression? How does the role of expressions differ in procedural and functional languages?
7. What is “message passing”, and how is it accomplished between two separate interacting processes?
8. Parameters allow program modules to communicate with each other. Name at least three mechanisms or data structures that must be a part of the semantic basis of a language to implement parameter passing for recursive subroutines.
9. Is there a difference between operators and functions? Explain.
10. Define infix, prefix, and postfix operator syntax.
11. What is the role of parentheses in an expression?
12. How do prefix and postfix order for expressions eliminate the need for parentheses?
13. Using a stack, evaluate the following postfix expression. Assume the stack is initially empty and that  $x=2$ ,  $y=3$ , and  $z=4$ . Show the contents of the stack after each push or pop.
 
$$x \ y \ + \ z \ - \ x \ z \ * \ -$$
14. Draw a computation tree for the following infix expression:
 
$$((x/y) * (((3*z) + (2 + y)) + 7))$$
15. Rewrite the expression of question 14 with the minimum number of parentheses, using the precedence rules in C or Pascal.
16. Draw a computation tree for the following prefix expression:
 
$$* / x \ y \ + \ 3 \ * \ * \ z \ 2 \ + \ y \ 7$$
17. Explain the two evaluation rules which govern order of evaluation of elements of a function call.
18. What is the difference between call-by-need and call-by-name?
19. What is automatic dereferencing? Give a specific example.
20. In FORTRAN, call-by-reference is the only mechanism for passing parameters. Why is this now considered poor design?

21. When should call-by-reference be used instead of call-by-value? Give two distinct situations.
22. In APL, parameter names are declared but their types are not specified. What is the purpose of the parameter declarations? How can communication work without parameter type specifications?