

2. *procedural decomposition*

```

for each  $x \in list$  do
  if  $x$  appears earlier in  $list$  then
    output ( $x$ )

```

or (using an auxiliary table)

```

for each  $x \in list$  do
  if  $find(table, x)$  then
    output ( $x$ )
  else
     $insert(table, x)$ 

```

Advantages

- the package ($table$, $find$, $insert$) can be designed separately from calling routine
- may produce first output sooner than (1)

Common terms for the solutions' structures:

1. producer-consumer:
 $sort$ produces list elements
 $findAdjacentDuplicates$ consumes them
2. client-server:
 $(table, find, insert)$ provides a service that is used by the caller (client)

Syntactic structure

Perhaps FORTRAN's most prominent feature (included in nearly all subsequent languages):

the arithmetic expression

— a major advance over assembler notation:

1. mathematical notation's familiarity
2. support for **modularization**

Consider an arithmetic expression of the form

$$E + F$$

where E and F may themselves be simple or complex arithmetic expressions.

- (1) The meaning of this whole expression can be understood wholly in terms of an understanding of the meanings of E and F [and (+)];
- (2) the purpose of each part consists solely in its contribution to the purpose of the whole;
- (3) the meaning of the two parts can be understood wholly independently of each other;
- (4) If E or F is an arithmetic expression, the same structuring principle can be applied to the analysis of the parts as is applied to the understanding of the whole.
- (5) the interface between the parts is clear, narrow, and well controlled: in this case just a single number;
- (6) the separation of the parts, and their relation to the whole, are apparent from their written form.

— C. A. R. Hoare, "Hints on programming-language design" in *Essays in Computing Science* (Prentice-Hall, 1989)

Expression notations come in several flavors; we'll examine four.

The purpose of every expression notation is to specify the relationships between expressions' parts.

Since expressions consist of *operators* and *operands*, an expression notation must specify

which operand(s) belong to each operator.

Conventional arithmetic expressions employ a mixture of *infix*, *prefix*, and *postfix* notation:

infix (mostly): $-b + \sqrt{b^2 - 4 \cdot a \cdot c}$

- each (dyadic) operator lies **between** its operands

Two less conventional notations: *prefix* and *postfix*

prefix: $+ _1 b \sqrt{_2 \wedge b^2} \cdot _4 a c$

- each operator **precedes** its operand(s)

postfix: $b _1 b^2 \wedge 4 a \cdot c \cdot _2 \sqrt{_4}$

- each operator **follows** its operand(s)

Infix notation needs **associativity**, **precedence**, and **parentheses** to resolve ambiguities.

In infix notation, *precedence* and *associativity* come into play when an **operand** occurs between two **operators**.

Example:

$$a + b \cdot c$$

To which operator does b belong?

- If the operators precedences are different, the higher one wins.
- When the operators' precedences are equal, the question is settled by their associativity (*left* or *right*); examples:

$$a + b + c, \quad a - b - c, \quad a / b \cdot c$$

Prefix and postfix are simpler— precedence, associativity, and parentheses are all unnecessary:

To interpret a postfix expression such as

$$16 \ 24 + \ 97 \ 53 - \cdot$$

all we need to know is each operator's **arity**.

Possible confusion: In mathematics, some operators are known as *associative*. An example is (+) on real numbers: For all reals a , b , and c ,

$$(a+b) + c = a + (b+c)$$

This mathematical associativity is a *semantic* property of (+)—i.e., it's a consequence of what (+) is defined to *mean*. In contrast, left-associativity and right-associativity are purely *syntactic* properties, having nothing to do with operators' meaning.

* an operator's *arity* is the number of operands it takes. If the arity is not fixed (as in Lisp) then parentheses are required.

Abstract Syntax Trees

Consider the expressions

infix	prefix	postfix
$(\sqrt{a})+(b-c)$	$+\sqrt{a}-bc$	$a\sqrt{bc}-+$

The *concrete* expressions—that is, the sequences of symbols on the page—are all different, but their *abstract* structure is the same.

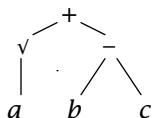
Abstractly speaking, an expression is one of the following:

- a name
- a numeral
- an expression with a unary operation
- a pair of expressions with a binary operation.

This is an *abstract syntax*: for each kind of expression, it specifies its constituent parts, but it does not specify how the parts are arranged.

How can an expression's abstract structure be expressed independent of any concrete representation?

It can be depicted as an **abstract syntax tree**:



Each subtree \Leftrightarrow a subexpression.
 each subtree's *root* \Leftrightarrow the subexpr's *operator*
 each root's *children* \Leftrightarrow the *operands*

Aside: Traversing an expression's abstract syntax tree ...

in $\left\{ \begin{matrix} \text{preorder} \\ \text{postorder} \end{matrix} \right\}$ yields a $\left\{ \begin{matrix} \text{prefix} \\ \text{postfix} \end{matrix} \right\}$ expression

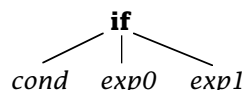
An *inorder* traversal gives an *infix* expression, but every subexpression must be parenthesized.

Each node's *degree* depends on its operator's *arity*.

Example: two versions of an arity-3 operator:

if cond then exp0 else exp1 (ML)
 cond ? exp0 : exp1 (C/C++)

and their abstract syntax tree:



Observation:

When we draw an abstract syntax tree, we're really creating another concrete expression notation—albeit a two-dimensional one.

Grammars

How can the syntax of a given notation be specified precisely, so that people can understand it and compilers can implement it?

The usual means of specifying a programming language's *concrete* syntax is a *set of rules* called its **grammar**.

A grammar is given in a *meta*-notation (which is a notation that is used in defining another notation).

A **context-free** grammar (the most common and useful kind for programming languages) consists of

1. atomic symbols (*terminals*)
2. variables (*nonterminals*)
3. *productions* define each nonterminal as a sequence of terminals and nonterminals
4. a *starting nonterminal*

The most popular notations for context-free grammars are

- BNF ("Backus-Naur Form")
- Extended BNF
- Syntax charts ("railroad diagrams")

BNF grammar notation

- each nonterminal is enclosed in $\langle \rangle$
- production: $\langle \text{nonterminal} \rangle ::= \text{sequence} \dots$
- empty sequence: $\langle \text{empty} \rangle$
- alternatives are separated by '|'
- all other symbols are terminals
- repetition is expressed only by recursion

Example: a language of integer expressions

$\langle \text{expression} \rangle ::= \langle \text{expression} \rangle + \langle \text{term} \rangle$
 $\quad \quad \quad | \langle \text{expression} \rangle - \langle \text{term} \rangle$
 $\quad \quad \quad | \langle \text{term} \rangle$
 $\langle \text{term} \rangle ::= \langle \text{term} \rangle * \langle \text{factor} \rangle$
 $\quad \quad \quad | \langle \text{term} \rangle / \langle \text{factor} \rangle$
 $\quad \quad \quad | \langle \text{factor} \rangle$
 $\langle \text{factor} \rangle ::= (\langle \text{expression} \rangle)$
 $\quad \quad \quad | \langle \text{name} \rangle | \langle \text{numeral} \rangle$
 $\langle \text{name} \rangle ::= \langle \text{letter} \rangle$
 $\quad \quad \quad | \langle \text{name} \rangle \langle \text{letter} \rangle$
 $\quad \quad \quad | \langle \text{name} \rangle \langle \text{digit} \rangle$
 $\langle \text{numeral} \rangle ::= \langle \text{digit} \rangle | \langle \text{numeral} \rangle \langle \text{digit} \rangle$
 $\langle \text{letter} \rangle ::= a | b | \dots | z | A | \dots | Z$
 $\langle \text{digit} \rangle ::= 0 | 1 | 2 | \dots | 9$