Expressing Algorithms

An algorithm may be expressed in a number of ways:

natural language:	usually verbose and ambiguous
flow charts:	avoid most (if not all) issues of ambiguity; difficult to modify w/o specialized tools; largely standardized
pseudo-code:	also avoids most issues of ambiguity; vaguely resembles common elements of programming languages; no particular agreement on syntax
programming language:	tend to require expressing low-level details that are not necessary for a high-level understanding

Common Elements of Algorithms

acquire data (input)

some means of reading values from an external source; most algorithms require data values to define the specific problem (e.g., coefficients of a polynomial)

computation

some means of performing arithmetic computations, comparisons, testing logical conditions, and so forth...

selection

some means of choosing among two or more possible courses of action, based upon initial data, user input and/or computed results

iteration

some means of repeatedly executing a collection of instructions, for a fixed number of times or until some logical condition holds

report results (output)

some means of reporting computed results to the user, or requesting additional data from the user

Simple Variables

simple variables

Some means hold a simple value, like the number of elements in a list.

Generally, we do not need to be too careful about the notion of types, but it is useful to distinguish between variables that hold numbers, character strings, and logical values.

Variables must be given names and those names should be descriptive. For readability, we will not allow whitespace in names.

Variables must be declared and given a type before they are used:

```
number WidthInFeet
string NameOfBook
logical valueWasEven
```

Logical variables can have the values `true` and `false`, which are considered part of the pseudo-language vocabulary and should not be used as variable names.

List Variables

list variables

Some means to refer to a collection of simple values as a unit, and to also refer to individual values within the collection.

Mathematically we can think of this as a subscripted list.

For convenience we often adopt a slightly different notation;

for a list named Scores, `Scores[i]` refers to the i-th element in the list;

we number elements starting at 1.

List declarations look like:

```
list number Scores
list string bookTitles
```

(We don't specify a size for the list, as we would do for an array in most programming languages… that's not necessary for stating an algorithm.)

pseudo-Language: the Basics

acquiring data

`get` `<variable>`

Obtain a value for the specified variable; no source is specified; this could stand for reading from a file or other device, or even for prompting a user for interactive input.

reporting results

`display` `<variable>`

Report the value of the variable in some manner; no destination is specified; this could stand for returning a value from the algorithm, for writing to a file or other device, or for displaying to a console window.

pseudo-Language: the Basics

computation

Use common mathematical notation, slightly adjusted to reflect keyboard limitations.

Any common mathematical and logical expressions can be formed using:

`+, -`	addition, subtraction
`*, /`	multiplication, division
`^`	exponentiation, e.g, `x^y` means `x` to the power `y`
`NOT, !`	logical negation
`AND, &&`	logical and
`OR, ||`	logical or
`(, )`	grouping

The default precedence is defined by the ordering above (low to high precedence).

When in doubt, add parenthesis for clarity.

pseudo-Language: the Basics

computation (continued)

In addition, any standard mathematical functions can be used, with suitable notation:

```
| x - y |
sin( theta )
```

Since the basic precedence rules cannot cover all such cases, if there's any doubt about clarity, use parentheses to disambiguate.

For setting the value of a variable from an expression, we will use

```
<variable> := <expression>
```

pseudo-Language: Precedence Rules

We must have some definition of the order of operations, unless we want to write lots of parentheses. The ordering used here is slightly adapted from the precedence rules for the C language:

highest

`(, )`	grouping
`NOT, !`	logical negation
`^`	exponentiation
`*, /`	multiplication, division (parenthesize if both are chained)
`+, -`	addition, subtraction (parenthesize if both are chained)
`AND, &&`	logical and
`OR, ||`	logical or

lowest

Remember, when in doubt, add parenthesis for clarity.

pseudo-Language: Other Issues

halting the algorithm

`halt`

Means "stop now".

May be used at any point within the algorithm.

comments

In many cases, it is useful to add comments to the (human) reader;

there must be some way to distinguish comments from the algorithm itself;

we'll use the convention that anything following the symbol `'#'` on a line is a comment

Algorithm Interface

To refer to algorithms, and especially to refer to one algorithm from another, we need to incorporate the notion of an interface to an algorithm:

algorithm <name> **takes** <list of inputs>

For example:

```
algorithm XtoN takes number X, number N

# Computes the value of X^N.
# Pre:  X is a number, N is an integer, N >= 0.
#
    number XtoN     # result

    . . .
    display XtoN    # report result
    halt
```

Algorithm Invocation

We must also be able to express the invocation of an algorithm:

<name> **(** <list of input values to algorithm> **)**

For example:

```
algorithm FutureValue takes number PV,      # present value
                            number Rate,    # annual pct rate
                            number Years    # term

# Computes the future value of an asset, assuming
# annual compounding of a given rate of return.
# Pre:  PV and Rate are non-negative numbers.
#       Years is a non-negative integer.
#
    number  FV                          # future value
    number  ScalingFactor               # growth factor

    ScalingFactor := XtoN(1 + Rate, Years)  # invoke algorithm XtoN
    FV := PV * ScalingFactor

    display FV
    halt
```

pseudo-Language: Selection

selection

```
if <condition>
    # one or more statements
endif

if <condition>
    # one or more statements
else
    # one or more statements
endif
```

A *condition* may be an algebraic comparison of two variables, such as

```
numberOfZeros < 100
```

or a logical concept, such as

```
haveNotSeenAZero
```

pseudo-Language: Selection Examples

```
if x <= y
    diff = y - x
endif

if x <= y
    diff = y - x
else
    diff = x - y
endif
```

pseudo-Language: Iteration

iteration

```
    while <condition>
        # one or more statements
    endwhile
```

pseudo-Language: Iteration Examples

```
while x <= y
    x := 2 * x
endwhile


while y >= 0
    if ( x > y )
        display iter
        halt
    endif
    iter := iter + 1
    y := y - x
endwhile          # y >= 0
```