–contents of Reference Manual for X–

Informal Reference Manual for X

People, even programmers, can use language with very little *a priori* knowledge. We all generalize from examples, guess intentions, and succeed in communicating long before the schools and textbooks round out our native tongue.

The language used here is named X; it is inspired by the language used by Edsger Dijkstra in his book *A Discipline of Programming*. Do not look for declarations, data structures or explicit input/output. Otherwise your expectations from other programming languages will be close enough.

Reaching for the elegance of scientific notation, Dijkstra typeset his programs, using a variety of mnemonic glyphs. As a practical matter, compiler input must be readily prepared and modified. An ASCII compromise has been chosen for X source input. The compiler XCOM can, upon request, typeset its input program. If you extend XCOM, you should also extend the typesetting.

The input format (not the typeset format) is used for the examples.

Whitespace

Except to separate other symbols in X, whitespace symbols are ignored.

Comments

Text starting with ` to the end of line is a comment. A comment is equivalent to whitespace.

Names

The concept of a *name* – a sequence of letters and digits – is universal in programming languages. In X some names are reserved and the rest may be used for program constructs.

An *operator name* is similar – a sequence of operator symbols.[1] Thus `+`, `:=` and `<=` are operator names in X. The operator name concept provides a handy form for extensions to X. The operator symbols used in operator names in X are

```
& * + - / : < = > ? | ~
```

If other symbols are actually used, the list above is automatically extended. Extra whitespace is sometimes required. For example

```
x:=-3;
```

will cause a compile error because `:=-` is not an operator.

The rest of the symbols in X stand for themselves.

Types, Constants and Expressions

X has three data types: logical, integer, and real.[2] Each type has manifest constants (e.g. `true`, `1`, `1.0`). Integer constants cannot be used where real values are required.

The operators and builtin functions of X are given below. The signature *c(dd)* means the operator accepts two arguments of type *d* and returns type *c*. Most numeric operators have two signatures each (polymorphic). The letters *bir* stand for the three X types respectively.

signatures	op	*meaning*
i(i) r(r)	`-x`	minus `x`
i(ii) r(rr)	`x+y`	`x` plus `y`
i(ii) r(rr)	`x-y`	`x` minus `y`
i(ii) r(rr)	`x*y`	`x` times `y`
i(ii) r(rr)	`x/y`	`x` divided by `y`
i(ii)	`x//y`	remainder of `x` divided by `y`
b(ii) b(rr)	`x<y`	`x` less than `y`
b(ii) b(rr)	`x<=y`	`x` less than or equal `y`
b(ii)	`x=y`	`x` equal `y`
b(ii)	`x~=y`	`x` not equal `y`
b(ii) b(rr)	`x>=y`	`x` greater than or equal `y`
b(ii) b(rr)	`x>y`	`x` greater than `y`
b(b)	`~x`	not `x`
b(bb)	`x&y`	`x` and `y`

[1] Frank DeRemer (of LALR fame) suggested this idea.

[2] These three data types are needed to express decisions, counting and measurement, thus they are available in almost every programming language.

b(bb)	`x|y`	`x` or `y`
(bb) (ii) (rr)	`x:=y`	`x` assigned the value of `y`
i(b)	`b2i(x)`	integer form of logical `x`
r(i)	`i2r(x)`	real form of integer `x`
i(r)	`r2i(x)`	integer form of real `x`
r()	`rand`	random real

The type constraints of builtin signatures are used to infer and/or check types of operands. It is possible to write an X program where the type of some variables are ill-defined. For example, x and y are inferred to have the same type below, but that type itself is not yet determined.

```
x := y
```

This particular type ambiguity can be remedied by inserting non-effect arithmetic such as addition of zero.

```
x := y+0
```

forcing the type to integer.

Type inference provides the capability normally given to declarations in other programming languages.

There is one more type reserved for function names. Function names can only be assigned, passed as parameters or return values, or called.

Arithmetic is native to the hardware: IEEE floating point for real, 2's complement for integer, and true/false for logical. Comparisons for equality between real values are not allowed because such a comparison does not make numerical sense.

Assignment, Input and Output

If one writes a program consisting of a single assignment:

```
h := 1;                                 ` this is a comment
```

one infers that `h` is an integer-valued variable which will be assigned the value `1`. This assignment to `h` is apparently wasted, since the value is not subsequently used. When this program is run, it will present the output

```
h := 1
```

That is, output is achieved in X by reporting wasted assignments. One can now intuit that undefined variables in the program:

```
b := c | d < 3.1;                       `input required
```

will require input values for `c` and `d`. Indeed, when this program is run, the user will be asked, in turn:

```
logical input c := ?
real input d := ?
```

Upon receiving a logical value for `c` and a real value for `d`, the program will run and report the final value of `b` upon completion.

The form of an assignment is a list of variables followed by an equally long list of expressions separated by an assignment operator. The effect is as if all the right-hand side expressions were evaluated, and then each simultaneously assigned to its corresponding left-hand side. In particular, the assignment

```
x,y := y,x
```

exchanges the values of `x` and `y`.

The result of assignment where the left-hand side contains a single name more than once is not defined.

Statements

There are, in addition to assignments, three other kinds of statements in X. Here are some examples of statements:

assignment

```
r, i, b := 1.0, i+2, true
```

selection

```
if x < y ?  x := y-1
:: x = y ?
:: x > y ?  x := y+1
fi
```

iteration

```
do x < y ?  x := x+7.111
od
```

calling a subprogram

```
r, y := myfun := 3.14, 37, false
```

Programs

Where one statement can appear, a list of statements separated by semicolons can appear. Statements are executed in the order of the list containing them. A statement list can be empty; a trailing semicolon never causes trouble.

A program is a list of statements. The meaning of a program is its input/output mapping; the effect of running a program is to display an instance of that input/output mapping. All input is collected before program execution and no output is reported until the completion of execution.

The input requests and output reports appear in the order in which the variables appear in the program. The runtime system must supply sufficient information to make the input/output actions unambiguous — associating the name of the variable and its value is sufficient. (The wise programmer will pick mnemonic names.)

The instantaneous state of a program is a set of variable-value pairs. The values of variables change during execution. The dynamic sequence of changes is, by intention, invisible to the user. The initial state is provided by input; the final state is reflected as output; what happens in between is the compiler writer's private business.

Variables, Implicit Name Introduction

A name appearing in an X program has only one meaning. It must be reserved, or name a subprogram, or name a variable. The attributes of each name are derived from the context surrounding its uses. Any program variable appearing only in an expression is an *input variable*; one appearing only on the left of an assignment is an *output variable*. If a variable is used before it is assigned, the value is undefined.

Control: Selection and Iteration

There are two control constructs in X, selection (`if-fi`) and iteration (`do-od`). Within `if-fi` brackets there is a set of alternatives. Each alternative consists of a guarding logical expression followed by a statement list.

The meaning of a selection is the meaning of the statement list behind the first true guard and, if there are no true guards, the meaning is *abort*. The order of tests means that the final guard can be `true` which then behaves as `else` in other languages.

Except for the delimiters, an iteration has the same form as a selection, and also executes the statement list behind the first true guard. If there is no true guard, the iteration terminates, otherwise it repeats.

Dijkstra chose nondeterministic, as contrasted to first true value, for selection and iteration. That choice makes more sense for program generation than it does for programming.

Guards: Logical Operators

The guards are logical expressions, combining relations over numeric expressions and the constants `true` and `false` with logical operators. It is in the nature of X that guards cannot change the state of the program since there are no side effects (except for builtin `rand`).

Subprograms

Any program also defines a subprogram. In the calling syntax, the *actual input* expressions are on the right; the *actual output* variables are on the left. A subprogram named `xyz` is defined in file `xyz.x`. For example, if `xyz.x` has two outputs and three inputs, the following line of X code will call it.

```
a,b := xyz := a+1.0, true, 13;     `call xyz
```

The call must supply the expected number and type of inputs; the number and type of outputs must correspond to the left-side of the call syntax.

The function identifier can be a function type variable. Such a variable must have been assigned a literal function name, either directly or indirectly.

Any subprogram can be run standalone; it will request values for its inputs and report its outputs. All files that mutually reference each other must be compiled and run together; the last one mentioned is the first to be executed. Recursion is allowed.

Assertions

In C one can write `assert(expr)` to insure that the `expr` evaluates to `true` every time the assertion is executed. Programmers use this construct to document and automatically check the conditions under which this code should work long after the original author has turned attention elsewhere. In X one writes

```
if expr ?  fi;    ` reason for check
```

to achieve the same effect. If the `expr` evaluates to `false`, the selection fails and the X program aborts.

–contents of X Examples–

X Examples

Example – primes

Suppose you wish to test whether an integer is prime. A simple approach is to examine the remainder after division by each feasible divisor. One can quit after trying everything up to the square root of the candidate value. Since fixed size 2's complement arithmetic is used, this approach will eventually fail for large integers.

```
n := PrimeCandidate;                  ` input integer
trial := 2;                           ` smallest possible
do trial*trial < n & n//trial ~= 0 ?
  trial := trial + 1                  ` keep trying
od;
NumberIsPrime := trial*trial > n      ` output
```

In this case the input/output might appear as:

```
PrimeCandidate := 97;
NumberIsPrime := true;
```

The publication form of the prime program is

n ← PrimeCandidate; ` *input integer*
trial ← 2; ` *smallest possible*
do trial×trial < n ∧ n//trial ≠ 0 ⇒ trial ← trial+1 ` *keep trying*
od;
NumberIsPrime ← trial×trial > n ` *output*

Example – π

A simple (and inefficient) way to compute π is to sum the series
$4 * (1/1 - 1/3 + 1/5 - 1/7 \ldots)$
It turns out that the average of the last two partial sums is more accurate than the last sum by itself. On a machine using 32-bit IEEE floating point, one should not expect more than about 7 decimal digits accuracy.

```
old, sgn, step, lim := 0.0, 1.0, 1, 1000;
do step < lim ?
  new := old+sgn/i2r(step);
  step := step+2;
  sgn := -sgn
  pi := 2.0*(old+new);              ` result
  old := new;
od
```

The (perhaps surprisingly inaccurate) output is

```
pi := 3.1415913;
```

Given suitable choices in the typesetting options, the typeset version might look like:

```
old, sgn, step, lim ← 0.0, 1.0, 1, 1000;
do step < lim ⇒
  new ← old+sgn/i2r(step);
  step ← step+2;
  sgn ← −sgn;
  π ← 2.0×(old+new);                ` result
  old ← new;
od
```

Example − e

The base of natural logarithms can be expressed as a continued fraction:
2 1 2 1 1 4 1 1 6 1 1 ....
Because the pattern is regular after the first two items `2 1`, the rest can be generated on the fly. Here is a program that does it. [1]

```
t := tol+0.0;                       ` input
i := 2.0;                           ` start at 2 1 1 4 1 1 ...
ao, a := 2.0, 3.0;                  ` numerators
bo, b := 1.0, 1.0;                  ` denominators
diff := ao/bo - a/b;
do diff*diff < t*t ?
  ao, a := a, ao+a*i;               ` i
  bo, b := b, bo+b*i;
  ao, a := a, ao+a;                 ` 1
  bo, b := b, bo+b;
  ao, a := a, ao+a;                 ` 1
  bo, b := b, bo+b;
  i := i+2.0
```

[1] See Knuth, The Art of Computer Programming, volume 2 for more information.

```
  diff := ao/bo - a/b;
od;

e := a/b                              ` output
```