X Language Definition

David May: November 1, 2016

The X Language

X is a simple sequential programming language. It is easy to compile and an X compiler written in X is available to simplify porting between architectures. It is relatively easy to modify the compiler to target new architectures or to extend the language.

Notation

The following examples illustrate the notation used in the definition of X.

The meaning of

$$assignment \quad = \quad variable \ \texttt{:=} \ expression$$

is "An *assignment* is a *variable* followed by := followed by an *expression*"

The meaning of

$$literal \quad = \quad integer \mid byte \mid string$$

is "An *literal* is an *integer* or a *byte* or a *string*". This may also be written

$$literal \quad = \quad integer$$

$$literal \quad = \quad byte$$

$$literal \quad = \quad string$$

The notation { *process* } means "a list of zero or more *processes*".

The notation $\{_0$, *expression*$\}$ means "a list of zero or more expressions separated from each other by ,", and $\{_1$, *expression*$\}$ means "a list of one or more expressions separated from each other by ,".

The format of an X program is specified by the syntax. Space, tab and line breaks are ignored and can be inserted in text strings using the escape character *.

Comment

comment = | *text* |

text = {$_0$ *character*}

A comment is used to describe the operation of the program.

process = *comment process*

Let C be a comment and P be a process. Then $C\ P$ behaves like P.

Statement

process = `skip`
| `stop`
| *assignment*
| *sequence*
| *conditional*
| *loop*
| *call*

`skip` starts, performs no action, and terminates.

`stop` starts but never proceeds and never terminates.

assignment = *variable* `:=` *expression*

An assignment evaluates the expression, assigns the result to the variable, and then terminates. All other variables are unchanged in value.

sequence = `{` {$_0$ `;` *process*} `}`

A sequence starts with the start of the first process. Each subsequent process starts if and when its predecessor terminates and the sequence terminates when the last process terminates. A sequence with no component processes behaves like `skip`.

Conditional

conditional = `if` *expression* `then` *process* `else` *process*

Let e be an expression and let P and Q be processes. Then

`if` e `then` P `else` Q

behaves like P if the initial value of e is *true*. Otherwise it behaves like Q.

Loop

loop = `while` *expression* `do` *process*

A loop is defined by

`while` *e* `do` *P* = `if` *e* `then { ` *P*`; while` *e* `do` *P* `} else skip`

Scope

process = *declaration* `;` *process*

program = *declaration* `;` *program*
| {$_1$ *definition* }

A process or program *D* `;` *S* behaves like its scope *S*; the declaration *D* specifies a name which may be used with this specification only within *S*.

definition = `proc` *name* `(` {$_0$ `,` *formal*} `)` `is` *body*
formal = *name*

body = *process*

The definition

`proc` *n* `(` {$_0$ `,` *formal*} `)` `is` *B*

defines *n* as the name of a procedure. The value of *n* is the address of the procedure entrypoint.

A program D_0 D_1 ... D_n specifies the names of procedures N_0, N_1, ... N_n which may be used within the procedure bodies B_0, B_1, ... B_n. The names N_0, N_1, ... N_n must be distinct.

declaration = `var` *name*
| `var` *name* `:=` *expression*

A declaration `var` *n* or `var` *n* `:=` *e* defines *n* as the name of a variable. The declaration `var` *n* `:=` *e* declares *n* as the name of a variable; the initial value of *n* is the value of *e*.

declaration = `val` *name* = *expression*
| `array` *name* `[` *expression* `]`

A declaration `val` *n* = *e* defines *n* as the value of *e*.

The declaration `array` *n* `[` *e* `]` defines *n* as the address of the first component of an array. The number of components in the array is the value of *e*.

Let x and y be names and let $S(x)$ and $S(y)$ be scopes which are similar except that $S(x)$ contains x wherever $S(y)$ contains y, and vice versa. Let $D(x)$ and $D(y)$ be declarations which are similar except that $D(x)$ is a declaration of x and $D(y)$ is a declaration of y. Then

$D(x)$; $S(x) = D(y)$; $S(y)$

Using this rule it is possible to express a process in a canonical form in which no name is specified more than once.

Procedure Call

call = *name* ({$_0$, *actual*})

actual = *expression*

Let X be a program expressed in the canonical form in which no name is specified more than once. If X contains a procedure definition

`proc` P`(`F_0`,` F_1`, ...,` F_n`)` `is` B

then within the scope of P

P`(`A_0`,` A_1`, ...,` A_n`)` = `var` F_0 `:=` A_0 ; `var` F_1 `:=` A_1 ; ... `var` F_n `:=` A_n ; B

provided that each declaration $F_i = A_i$ is valid.

A procedure can always be compiled either by substitution of its body as described above or as a closed subroutine.

Variables and Arrays

element = *element*`[` *subscript* `]`
| *name*

Every variable has a value that can be changed by assignment or input. The value of a variable is the value most recently assigned to it, or is arbitrary if no value has been assigned to it.

Let a be an array with n components and e an expression of value s. Then v`[`e`]` is valid only if $0 \leq s$ and $s < n$; it is the component of v selected by s.

Let a be an array with n components, e be an expression of value s, and x be an expression. If $0 \leq s$ and $s < n$, then v`[`e`]` `:=` x assigns to v a new value in which the component of v selected by s is replaced by the value of x and all other components are unchanged. Otherwise the assignment is invalid.

Literal

literal = *integer* | *byte* | *string* | *table* | `true` | `false`

integer = *digits* | #*digits* | #b*digits*

byte = '*character*'

string = " {$_0$ *character* } "

table = [{$_0$, *expression* }]

An integer literal is a decimal number, # followed by a hexadecimal number or #b followed by a binary number. A byte literal is an ASCII character enclosed in single quotation marks: '.

A string literal is represented by a sequence of ASCII characters enclosed by double quotation marks: ". Let s be a string of n characters, where $n < 256$. The value of s is an array containing the value n, followed by ASCII values of the characters in the string. The string is packed into the array.

A table literal is represented by a sequence of expressions enclosed be brackets. The value of a table [E_0 , E_1 , ... E_n] is an array in which each component is the value of the corresponding expression.

The literal `true` represents the logical value *true*; numerically `true` = 1. The literal `false` represents the logical value *false*; numerically `false` = 0.

Expression

An expression has a data type and a value. Expressions are constructed from operands, operators and parentheses.

operand = *element* | *literal*
| (*expression*)

The value of an operand is that of an element, literal or expression.

expression = *monadic.operator operand*
| *operand diadic.operator operand*
| *operand*

The operator `.` is defined by $a.n$ = `a[n]`, where a and n are operands.

The arithmetic operators + and - produce the arithmetic sum and difference of their operands respectively. Both operands must be integer values and the

result is an integer value. The arithmetic operators treat their operands as signed integer values and produce signed integer results. If n is an operand, then -n = (0-n).

The logical operator and produces the logical and of its operands, both of which must have value true or false. If the value of the first operand is false, the result is false; otherwise the result is the value of the second operand.

The logical operator or produces the logical or of its operands, both of which must have value true or false. If the value of the first operand is true, the result is true; otherwise the result is the value of the second operand.

The logical operator not produces the logical not of its operand which must have value true or false:

not false = true not true = false

Let **O** be one of the associative operators +, and, or and o_1 ... o_n be operands. Then

o_1 **O** o_2 **O** ... **O** o_n = (o_1 **O** (o_2 **O** (... **O** o_n) ...))

The relational operators =, <>, <, <=, >, >= produce a result of true or false. The operands must both be integer values. The result of $x = y$ is true if the value of x is equal to that of y. The result of $x < y$ is true if the integer value of x is strictly less than that of y. The other operators obey the following rules:

$(x <> y)$	= not $(x = y)$	$(x > y)$	$= (y < x)$
$(x >= y)$	= not $(x < y)$	$(x <= y)$	$= (y >= x)$

where x and y are any values.

return	=	return *expression*
	|	{ {$_0$; *process*} ; *return* }
	|	if *expression* then *return* else *return*

return	=	*specification* ; *return*

expression	=	*return*

A return executes to produce a value. The return return E evaluates its expression E and the resulting value is the value of the return. The return

{ P_0 ; P_1 ; ... P_n ; R }

executes the processes in sequence; the resulting value is the value of the return R. The return

`if` C `then` R_t `else` R_f

evaluates the condition C. If the condition is true the result is the value of R_t; otherwise the result is the value of R_f.

Function

$definition$ = `func` $name$ `(` $\{_0$ `,` $formal\}$ `)` `is` $return$

$expression$ = $name$ `(` $\{_0$ `,` $actual\}$ `)`

The definition

`func` n `(` $\{_0$ `,` $formal\}$ `)` `is` $return$

defines n as the name of a function with a return that computes the value of the function.

Let X be a program expressed in the canonical form in which no name is specified more than once. If X contains a function definition

`func` F`(`F_0`,` F_1`,` ...`,` F_n`)` `is` R

then within the scope of F

F`(`A_0`,` A_1`,` ...`,` A_n`)` = `var` F_0 `:=` A_0 `;` `var` F_1 `:=` A_1 `;` ... `var` F_n `:=` A_n `;` R

provided that each declaration `var` F_i `:=` A_i is valid.

A function can always be compiled either by substitution of its body as described above or as a closed subroutine.

Character set

The characters used in X are as follows.

Alphabetic characters	`ABCDEFGHIJKLMNOPQRSTUVWXYZ`
	`abcdefghijklmnopqrstuvwxyz`
Digits	`0123456789`
Special characters	`!"#&'()*+,-_./:;<=>?[]{}`

Comments may contain any X character.

Strings and character constants may contain any X character except *, ' and ". Certain characters are represented in character constants and strings as follows:

`*c`	carriage return
`*n`	newline
`*t`	horizontal tabulate
`*s`	space
`*'`	quotation mark
`*"`	double quotation mark
`**`	asterisk

Any character can be represented by *# followed by two hexadecimal digits.

A name consists of a sequence of alphabetic characters, decimal digits and underscores (_), the first of which must be an alphabetic character. Two names are the same only if they consist of the same sequence of characters and corresponding characters have the same case.