CS4120/4121/5120/5121—Spring 2022
Xi Language Specification
Cornell University
Version of May 9, 2022

In this course you will start by building a compiler for a language called Xi, named after the Greek letter Ξ. (Like the Greeks, we will pronounce it like "ksee".) Xi is an imperative, procedural language, like C. The next generation of the language has additional features for building more complex programs. But at present, developers must make do without them.

0 Changes

- 3/1: Clarified handling of integer literals.
- 2/27: Clarified array initialization semantics.
- 2/20: Clarified (lack of) overloading

1 Overview of features

Xi programs consist of a single source file containing definitions of one or more functions. Execution of a program consists of evaluating a call to the function `main`.

The language has two primitive types: integers (`int`) and booleans (`bool`). The array type T`[]` exists for any type T, so T`[][]` represents an array of arrays.

Functions may return a value, but need not. A function that does not return a value is called a procedure. A function may take multiple arguments. Unlike in languages such as C and Java, a function may also return multiple results.

Statement and expression forms should largely be familiar to C programmers.

There is no string type, but the type `int[]` may be used for most of the same purposes. Literal string constants have this type.

2 Variables

Variables are declared by following them with a type declaration and an optional initialization expression. There are no holes in scope; a variable may not be declared when another variable of the same name is already in scope. Here are some examples of variable declarations in Xi:

```
x:int = 2;
z:int
b: bool, i:int = f(x)
s: int[] = "Hello";
```

A variable declaration need not initialize the variable, as the declaration of `z` shows. Use of the value of an uninitialized variable has undefined behavior. Xi compilers are not required to detect the use of uninitialized variables[1].

Identifiers, including variable names, start with any letter and may continue with any sequence of letters, numbers, underscore character (_), or single quote characters (').

As in Java, variables are in scope from the point of declaration till the end of their block. A variable declaration may occur in the middle of the block, as in Java. A variable declaration that is the last statement in its block is in scope nowhere.

[1]This would be a nice extension to the language. You are permitted to make backward-compatible extensions.

The value of a variable can be changed imperatively using an assignment statement, as in the following examples:

```
x = x + 1
s = {1, 2, 3}
b = !b
```

3 Function definitions

A program contains a sequence of function definitions, including the definitions of the function `main`. All functions in the program are in scope in the bodies of all other functions, even if the use precedes the definition.

A function definition starts with the name of the function, followed by its argument(s), its return type(s), and the definition of its code. Only one function with a given name can exist in a program. There is no overloading.

For example, below is a function to compute the GCD of two integers. The body of the function is a block of statements. The result of a function is returned using the `return` statement. To simplify parsing, a return statement must be the last statement in its block.

```
// Return the greatest common divisor of two integers
gcd(a:int, b:int):int {
  while (a != 0) {
    if (a < b) b = b - a
    else a = a - b
  }
  return b
}
```

3.1 Multiple results

Unlike in C or Java, a function may return multiple results, indicated in the function definition by giving a list of return types, separated by commas. If the list is absent, the function is a procedure. The result of a function is a list of expressions, separated by commas. For example, the following function returns a pair of integers.

```
// Add two rational numbers p1/q1 and p2/q2, returning
// two numbers p3, q3 representing their sum, p3/q3.
ratadd(p1:int, q1:int, p2:int, q2:int) : int, int {
    g:int = gcd(q1,q2)
    p3:int = p1*(q2/g) + p2*(q1/g)
    return p3, q1/g*q2
}
```

Results from a function that returns multiple values can be used only through a *multiple assignment* in which the left-hand side is a sequence of variable declarations. For example, line 1 in the following code has the effect of assigning 11 to `p` and 15 to `q`:

```
1 p:int, q:int = ratadd(2, 5, 1, 3)
2 _, q':int = ratadd(1, 2, 1, 3)
```

The pseudo-declaration _ can be used to discard one of the results, as in line 2, which assigns 6 to `q'` but discards the corresponding numerator. This kind of declaration can also be used to explicitly discard the result of a function call that returns a single result.

3.2 Global variables

A module can contain global variable declarations, written at the top level outside any function definition:

```
len: int = 100
n':␣int␣=␣-1
debug:␣bool␣=␣false
points:␣int[]
```

Integer variables may be initialized at the point of declaration to an integer literal, and boolean variables may be initialized to a boolean literal. Global arrays can be declared but cannot be initialized at the point of declaration. All global variables may have their value changed by assignments in functions, e.g.:

```
setDebug(b: bool) {
  debug = b
}
```

Global variables cannot be declared in interfaces, so they are private to a given module. Two different global variables in different modules are different global variables even if they happen to be declared with the same name.

4 Data types

4.1 Integers

The type `int` describes integers from -2^{63} to $2^{63}-1$. They support the usual operations: `+`, `-`, `*`, `/`, and `%`, which all operate modulo 2^{64}. In addition, the "high multiplication" operator `*>>` returns the high 64 bits of the 128-bit product of its operands. This operator is helpful for doing certain kinds of number crunching. Division by zero causes the program to halt with an error. Integers can be compared with the usual Java/C relational operators: ==, !=, <, <=, >, and >=.

A literal integer constant is denoted by an optional minus sign and a sequence of digits starting with a digit in 1–9. A character literal as in Java may be used to denote an integer, so `'a'` is the same as `97`. Character literals may signify any legal Unicode character, with codes ranging from `U+000000` to `U+10FFFF` permitted.

To keep the language simple, Xi does not support floating point numbers. Programmers wishing to compute on other kinds of numbers must use other numeric representations such as rational numbers or fixed-point representations.

4.2 Booleans

The type `bool` has two values, `true` and `false`. The binary operator `&` is a short-circuit 'and' and the operator `|` is short-circuit 'or'. The unary operation `!` is negation. Booleans can also be compared with == and `!=`.

4.3 Arrays

An array *T*`[]` is a fixed-length sequence of mutable cells of type *T*. If `a` is an array and `i` is an integer, then the value of the array index expression `a[i]` is the contents of the array cell at index `i`. To be a valid index, an index `i` must be nonnegative and less than the length of the array. If `i` is not valid, this is caught at run time and the program halts with an error message. The expression `length(`*e*`)` gives the length of the array *e*.

Array cells may be assigned to using an array index expression on the left-hand side of an assignment, as at lines 9 and 10 of the following procedure, whose effect is to insertion-sort its input array.

```
sort(a: int[]) {
  i:int = 0
  n:int = length(a)
  while i < n {
      j:int = i
      while j > 0 {
        if a[j-1] > a[j] {
            swap:int = a[j]
            a[j] = a[j-1]
            a[j-1] = swap
        }
        j = j-1
      }
      i = i+1
  }
}
```

An array can be constructed by using an *array constructor*, specifying its elements inside braces. Similar to the array initializer syntax in Java and C, elements of the array constructor are separated by commas, and the final element may be followed by a comma. Thus, `{}` can be used as an array of length zero, and `{2,}` can be used as an array of length 1. An array constructor can be used anywhere that an array is expected.

A string literal such as `"Hello"` may also be used as an array constructor. The following two array definitions are therefore equivalent:

```
a: int[] = { 72,101,108,108,111 }
a: int[] = "Hello"
```

String literals may not span multiple lines in the source file. Like character literals, each character of a string literal must be a legal Unicode character.

An array of arbitrary length `n`, whose cells are not initialized, may be created at the point of declaration by including the length in the declaration of the array. The length is not part of the array's type and it need not be a constant:

```
n: int = gcd(10, 2)
a: int[n]
while n > 0 {
  n = n - 1
  a[n] = n
}
```

Use of uninitialized array cells has undefined results.

Arrays may be compared with `==` and `!=` to determine whether they are aliases for the same array. Different arrays with the same contents are considered unequal.

Arrays are implemented by placing the representations of the values of each of their cells contiguously in memory. They also record their lengths.

The operator `+` may be used to concatenate two arrays whose elements are of the same type, which is particularly handy for arrays of `int` representing strings, e.g.:

```
  s: int[] = "Hello" + {13, 10}
```

Multidimensional arrays Multidimensional arrays are represented by arrays of arrays, as in Java. So the type `int[][]` is represented as an array of pointers to arrays. A multidimensional array can be initialized by providing some or all dimensions in its variable declaration. Consider the following four declarations:

```
1 a: int[][]
2 b: int[3][4]
3 a = b
4 c: int[3][]
5 c[0] = b[0]; c[1] = b[1]; c[2] = b[2]
6 d: int[][] = {{1, 0}, {0, 1}}
7 err1: int[3] = { 1, 2, 3 } // ILLEGAL
8 err2: int[][3]             // ILLEGAL
```

Line 1 leaves `a` uninitialized. To be used, `a` must be initialized with a pointer to an array of arrays, as on line 3. Line 2 sets `b` to a pointer to an array of 3 elements, each of which is initialized to point to an uninitialized array of 4 elements. Line 4 makes `c` a pointer to an array of 3 elements, but those elements are not initialized to point to arrays. Line 5 initializes the elements of `c` to share the same underlying arrays as `a` and `b`. Line 6 initializes `d` as a 2×2 array representing an identity matrix.

Lines 7 and 8 show illegal array declarations. The first one is illegal because it combines two different ways to initialize the array contents. The second one is illegal because it cannot be created as described: a dimension is specified after an unspecified dimension.

5 Precedence

Expressions in Xi have different levels of precedence. The following table gives the associativity of the various operators, in order of decreasing precedence:

Operator	Description	Associativity
	function call, `[]`	left
`-`, `!`	integer and logical negation	—
`*`, `*>>`, `/`, `%`	multiplication, high multiplication, division, remainder	left
`+`, `-`	addition, subtraction	left
`<`, `<=`, `>=`, `>`	comparison operators	left
`==`, `!=`	equality operators	left
`&`	logical and	left
`|`	logical or	left

6 Statements

The legal statements, also called commands, are the following:

- An assignment to a variable or to an array element. However, the left-hand side of an assignment (and thus the assignment itself) cannot begin with an open parenthesis (`(`), an open brace (`{`), or a quote (`"`)
- `if` and `while` statements with syntax similar to that in C and Java except that parentheses are not required in the guard expression of each.
- A `return` statement. In a procedure, this is written just as `return`; in a function with a return type, the value(s) to be returned follow the `return` keyword, separated by commas. Unlike in Java, a `return` statement may only be used inside a block and must be the last statement in its block.
- A call to a procedure (but not a function).
- A block of statements, surrounded by braces. A block may be empty or may contain a sequence of statements. Each statement in a block may be terminated by a semicolon, but semicolons are entirely optional, even if statements are on the same line. Anywhere a block of statements is expected, a single statement may be used instead, except in a function definition. However, a `return` statement may not be used in place of a block.
- A variable declaration, with an optional initialization expression. It may declare multiple variables, in which case there must be an initialization expression that is a function call with the appropriate return types.

7 Lexical considerations

The language is case-sensitive. An input file is a sequence of Unicode characters, encoded using UTF-8. Therefore ASCII input is always valid.

Comments are indicated by a double slash `//` followed by any sequence of characters until a newline character.

Keywords (`use`, `if`, `while`, `else`, `return`, `length`) may not be used as identifiers. Nor may the names or values of the primitive types (`int`, `bool`, `true`, `false`).

String and character literals should support some reasonable set of character escapes, including at least "`\\`", "`\n`", and "`\'`". In addition, an escape of the form "`\x{`*HHHHHH*`}`", where *HHHHHH* stands for 1–6 hexadecimal digits (upper or lower case), represents the Unicode character with the corresponding code. For example "`\x{0a}`" is the same as "`\n`".

You may be more successful parsing negative integer literals as the negation of a positive literal.

8 Source files and interfaces

The Xi compiler compiles a source file with extension `.xi` to runnable code. It may also read in interface files that describe external code to be used by the program.

Interface files contain a nonempty set of procedure and function declarations without implementations and may contain end-of-line comments. Interface files have the extension `.ixi`. To use the procedures and functions declared in interface file `F.ixi`, a source file includes the top-level declaration "`use F`", optionally terminated by a semicolon. This causes the compiler to look for `F.ixi`. All such "`use`" declarations must precede all procedure and function definitions.

Multiple "`use`" declarations are permitted within one source file. The same function or procedure may be declared in multiple `.ixi` files that are read in and may also be defined in the source file, but its signature must match everywhere it appears. Therefore, it is legal to reference an interface more than once in a source file.

In typical usage, a program defined in `F.xi` would contain a statement "`use F`", causing the compiler to read the interface `F.ixi` and to check its definitions against declarations appearing in the interface. However, an interface is not required; even if there is an interface, procedures and functions need not be declared in that interface.

9 Current library interfaces

Interfaces for I/O and corresponding libraries are available, including the following functions from interface file `io`:

```
// I/O support

print(str: int[])      // Print a string to standard output.
println(str: int[])    // Print a string to standard output, followed by a newline.
readln() : int[]       // Read from standard input until a newline.
getchar() : int        // Read a single character from standard input.
                       // Returns -1 if the end of input has been reached.
eof() : bool           // Test for end of file on standard input.
```

Using these functions, we can easily write the canonical "Hello, World!" program:

```
use io

main(args: int[][]) {
  println("Hello, World!")
}
```

Some utility functions are found in the interface file conv:

```
// String conversion functions

// If "str" contains a sequence of ASCII characters that correctly represent
// an integer constant n, return (n, true). Otherwise return (0, false).
parseInt(str: int[]): int, bool

// Return a sequence of ASCII characters representing the
// integer n.
unparseInt(n: int): int[]
```