Xi Language Specification

Computer Science 4120
Fall 2011
Cornell University

Version of November 20, 2011

In this course you will start by building a compiler for a language called Xi. This is an imperative, procedural language, like C. The next generation of the language will have object-oriented features. But at present, developers must make do without them.

1 Overview of features

Xi programs consist of a single source file containing definitions of one 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, which is equivalent to passing a tuple to the function. 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. 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 only statement in its block is in scope nowhere.

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

[1]This would be a nice extension to the language.

```
x = x + 1
  s = (1, 2, 3)
  b = !b
```

3 Function declarations

A program contains a sequence of function declarations, including the declaration 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 declaration.

A function declaration starts with the name of the function, followed by its argument(s), its return type, and the definition of its code. For example, here is a function to compute the GCD of two integers. The body of the function is a block of statements. Statements may be terminated by semicolons but need not be.

```
// 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)
}
```

The result of the function is returned using the `return` statement. To simplify parsing, a return statement must be the last statement in its block.

3.1 Multiple results

A function may return multiple results, unlike in C or Java. This is indicated in the function declaration by giving a list of return types. For example, the following function returns a pair of integers.

```
// Add two rational numbers p1/q1 and p2/q2, returning
// a pair (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 *tuple assignment* in which the left-hand side is a tuple 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 to `q'` but discards the corresponding numerator. A `_` can also be used similarly to discard the single result of a function that returns only one result.

4 Data types

4.1 Integers

The type `int` describes integers from -2^{63} to $2^{63} - 1$. They support the usual operations: `+`, `-`, `/`, `*`, `%`, which all operate modulo 2^{64}. Division by zero causes the program to halt with an error. Integers can be compared with the usual Java/C relational operators: `==`, `!=`, `<`, `<=`, `>`, `>=`.

An integer constant is denoted by a sequence of digits. Nonzero integers start with one of the digits 1–9. A character constant as in Java may be used to denote an integer, so `'a'` is the same as `96`.

Xi does not support floating point numbers[2].

4.2 Booleans

The type `bool` has two values, `true` and `false`. The operation `&` is a short-circuit 'and' and the operation `|` 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.

```
1  sort(a: int[]) {
2   i:int = 0
3   n:int = length(a)
4   while (i < n) {
5     j:int = i
6     while (j > 0) {
7       if (a[j-1] > a[j]) {
8         swap:int = a[j]
9         a[j] = a[j-1]
10        a[j-1] = swap
11      }
12      j = j-1
13    }
14    i = i+1
15  }
16 }
```

A tuple containing elements of the right type may be used in a context expecting an array. This results in creating a new array that has the same elements as the tuple. Elements of the tuple are separated by commas; 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.

A string constant such as `"Hello"` may also be used to create an array of integers. The following two array definitions are therefore equivalent:

```
a: int[] = (72,101,108,108,111)
a: int[] = "Hello"
```

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 type of the array. The length need not be a constant:

[2] This is to keep the language simple. Programmers wishing to compute on non-integers must use other numeric representations such as rational numbers or fixed-point representations.

```
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.

The operator `+` may be used to concatenate two arrays whose elements are of the same type. This 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 dimensions in the variable declarations. 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))
```

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.

As in Java, lengths need not be provided for all dimensions. The following declaration only creates the top-level array; its two elements of type `Point[]` are initialized to `null`.

```
twoarrays: Point[2][]; // similar to Java: new Point[2][]
```

In a multidimensional array definition, all dimensions that specify lengths must occur to the left of all dimensions that do not have lengths. This semantics may seem counterintuitive but it matches Java.

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

A function body is a block of statements, also called commands, in braces. A block may be empty or may contain a sequence of statements. Statements in the block may be separated with semicolons, but this is not necessary, even if they're on the same line.

The legal statements are the following:

- An `if` statement, with the same syntax as C and Java, modulo semicolons.
- A `while` statement, with the same syntax as C and Java, modulo semicolons.
- 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. 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, each optionally terminated by a semicolon. A block may be empty. Anywhere a statement is expected, a block may be used instead, except in a function declaration. However, a `return` statement may not be used in place of a block.
- A variable declaration, with an optional initialization expression.
- An assignment to one or more variables.

There are no empty statements as in C and Java.

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

String and character constants should support some reasonable set of character escapes, certainly including "`\\`", "`\n`", and "`\'`".

8 Source files

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 set of function declarations without implementations. Interface files have the extension `.ixi`. To use the functions declared in interface file `F.ixi`, a program contains the top-level declaration "`use F`". All such declarations must precede all function definitions.

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.
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[]
```