# Iota$_9$ Language Specification

Computer Science 4120
Cornell University

Version of September 21, 2009

In this course you will be building a compiler for a language called Iota$_9$. This is an imperative, procedural language with similarities to C. The next generation of the language, codenamed Iota$_X$, is rumored to have object-oriented features. But currently developers must make do without them, at least until you extend your compiler.

## 1 Overview of features

Iota$_9$ 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. Iota$_9$ also has *tuple types*, containing an ordered sequence of values of any type. A value of a tuple type `(`$T_1$`,...,`$T_n$`)` is a sequence of $n$ values with respective types $T_1$ through $T_n$. Therefore the type `(int,int)[]` is an array each of whose elements is a pair of integers.

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.

Statement and expression forms are fairly familiar to Java programmers. Brackets are not used in array indexing operations, and parentheses may be omitted from function applications (unless multiple arguments are passed) and from array indices.

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

## 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 Iota$_9$:

```
x:int = 2;
y:(bool, int) = (true, 1);
z:int;
(b: bool, i:int) = y;
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. Iota$_9$ 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, or underscore characters.

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

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:

```
x = x + 1;
y = (false, 0);
b = !b;
s = (1, 2, 3, 4);
```

## 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, following an equal sign.

```
gcd(a:int, b:int):int = {
  while (a != 0) {
    if (a<b) b = b - a;
    else a = a - b;
  }
  return b;
}
```

A function like `gcd` appears to take multiple arguments, but in this case the argument is actually a tuple. For example, both of the calls to `gcd` in the following example are valid, and essentially same thing will happen at run time:

```
args:(int,int) = (2,5);
ans1:int = gcd args;
ans2:int = gcd(2,5);
```

## 4 Data types

### 4.1 Integers

The type `int` describes integers from $-2^{31}$ to $2^{31}-1$. They support the usual operations: `+`, `-`, `/`, `*`, `%`, which all operate modulo $2^{32}$. 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`.

Iota$_9$ 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 `!=`.

[2]Because the x86 support for them is peculiar. This would be a natural extension of the language, however.

### 4.3 Tuples

Tuples are similar to `structs` in C, except that tuple elements do not have a name. A tuple value is written as a list of values in parentheses, separated by commas, e.g. `(1,2,3)`. Tuple types are written using similar syntax.

A variable of tuple type can be assigned to a new tuple. There is no syntax for updating a single element of a tuple, however.

The components of a tuple can be extracted by a simple form of pattern matching that introduces multiple variables at once. The pattern is a sequence of variable declarations of the same length as the tuple. For example, the first and third components of the tuple above could be extracted into variables `x` and `y` as follows:

```
(x:int, _, z:int) = (1,2,3);
```

New variables must be declared on the left-hand side. However, the special name _ may be used in place of a variable declaration, with the effect that the corresponding tuple component is not assigned to anything.

Tuples are passed to functions by value—that is, by copying the tuple contents to the parameter variable. Therefore, assignments to the parameter variable inside the function do not affect the calling function. For example, in the `gcd` function above, the contents of the variable `args` cannot be affected by the first call to `gcd`.

A tuple may contain tuples, which results in the same memory layout as having the tuples flattened into a single tuple.

### 4.4 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 expression `a i` is the contents of the array cell at index `i`. To be 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. 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)) {  // parens are optional on 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. It does not share storage with the tuple. 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:

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

Use of uninitialized 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. This is true of arrays of tuples as well. For example, if the type `(int,int)` were being used to represent complex numbers, the type `(int,int)[]` would compactly represent an array of complex numbers, without a level of indirection that would be incurred in, say, Java.

## 5 Precedence

Expressions in Iota$_9$ 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, array access | left |
| -, ! | integer and logical negation | — |
| *, /, % | 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 must be separated with semicolons, except that (as in C and Java) a statement that ends in a block itself need not be followed by a semicolon.

The legal statements are the following:

- An empty statement, which has no effect when executed.
- An `if` statement, with the same syntax as C and Java.
- A `while` statement, with the same syntax as C and Java.
- A `break` statement, with the same syntax as C and Java. It may only be used inside the body of a `while` statement. If executed, it causes immediate termination of the nearest enclosing `while` statement.
- A `return` statement. In a procedure, this is written just as `return`; in a function with a return type, the value to be returned follows the `return` keyword.

- A call to a procedure (but not a function).
- A block of statements. Anywhere a block of statements is expected, a single semicolon-terminated statement may be used instead. For example, the following is a legal function declaration:

  ```
  min(x:int, y:int) = if (x<y) return x; else return y;
  ```

- A variable declaration, with an optional initialization expression.
- An assignment to a single variable.

## 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 Interfaces

Interface files may be declared in Iota$_9$. They contain a set of function declarations without implementations. To use the functions declared in interface file `F.i9`, a program contains the top-level declaration "`use F;`". All such declarations must precede all function definitions.

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.
```

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[]
```

We can easily write the canonical "Hello, World!" program:

```
use io;

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