



bjou

Programming Language

Brandon Kammerdiener

An introduction to the language, its design, and motivation
through example.

Contents

Goals	2
Goals of this Document	2
Goals of this Language	2
Setup	3
Getting bjou	3
Setting Up Your Environment	3
The Language	4
Variables, Type Intelligence	4
Constants	6
Defining Types	7
Procedures	8
Talking to C	9
Interfaces	9
Templates	9
Modules	9
Non-Linear Compiler Logic	9
Beyond the Language	10
The Compiler as a Tool	10
Using bJou to Program the Compiler	10

Goals

Goals of this Document

This document aims to provide background and explain the motivations behind the bJou programming language. It will also describe and demonstrate the features of bJou by guiding the reader through code examples that can be compiled with the bJou compiler that goes with this document.

Goals of this Language

bJou is my attempt to create the programming language that I want to use. So, its features and design are almost entirely based around my specific needs and interests. I love lower-level programming. Stuff like, well, compilers. Things that are important to me in a language are direct access to memory and hardware, performance, and expressiveness. That list is probably not too surprising and truthfully, there are many languages that take these priorities and are great languages. C is incredibly fast. Writing Python is like writing poetry. bJou seeks to take what are, in my opinion, the best attributes from many languages and combine them into one. In short, bJou is a compiled, statically typed, multi-paradigm language with an emphasis in clear and intentioned abstraction techniques. bJou also takes an interesting approach to metaprogramming, which will be explored later.

Setup

Getting bJou

Visit <https://github.com/kammerdienerb/bJou.git> to find the latest version of the bJou compiler. Download the zip folder and decompress it somewhere convenient.

Setting Up Your Environment

...

The Language

Now that everything is up and running, we can look at some specific examples of what the language is and what it can do. One important thing to mention before we continue is that many syntax choices of the language in its current state are temporary and will most likely change. The features and ideas are more important at this point anyway. Onwards!

Variables, Type Intelligence

```
1  # demo1.bjou
2  # Variables, Type Intelligence
3
4  (proc main() {
5      num : int
6      num = 12345
7      word : char* = "Foo"
8
9      new_num : int* = new int
10
11     floatingpt := 56.789
12     new_char := new char
13
14     @new_char = 'b'
15
16     print "num: %, word: %, floatingpt: %, new_char: %", num, word,
↵     floatingpt, @new_char
17
18     delete new_num
19     delete new_char
20
21     printf("%c\n", "string"[3])
22     printf("%f\n", { 1.23, 4.56, 7.89 }[1])
23
24     # array0 : int[num]
25     array1 : int[3 + 2]
26     array2 := { 1, 2, 3, 4, 5 }
27     # i := array2[1+6]
28     print "array[3] = %", array2[1+2]
29 })()
```

The code located in the file ‘tests/test/demo1.bjou’ is shown above. The purpose of this first demo file is to demonstrate basic variable declaration/use in bJou and, more interestingly, the compiler’s utilization of type information.

Before we dive into that, let's look at the first couple of lines and figure some things out. Lines 1 and 2 are comments. Comments can be made anywhere with the `#` character and tell the parser to ignore the rest of the line. Line 4 (closing on line 29) has some interesting details that we will hand-wave for now. If you are really curious, you can skip ahead to the section on Procedures later in this document, but for now all we need to know is that a procedure called `main` is being declared, defined, and called and that everything else in this file is taking place in that procedure. If that's still confusing, don't worry about it – it's not that important in this context.

Now take a look at line 5. This is a variable declaration that says `num` is a variable that is of type `int`. The next line describes a simple assignment, telling `bJou` to set `num` equal to the integer `12345`. The complete syntax for a variable declaration follows this pattern:

$$identifier : type\ declarator = initialization$$

This pattern in its entirety is shown on line 7. I say entirety because it is not always necessary to write all three of those components of a variable declaration. As you may have noticed, the first declaration on line 5 does not have an initialization. Lines 11 and 12 show variable declarations without type declarators. The rule of thumb for variable declarations is that it must have at least a type declarator or an initialization if not both.

The reason for the aforementioned rule is that the compiler *must* be able know the type of the variable. The cool thing is that an initialization says enough about the variable that the compiler can figure out the required type from it. So, on line 11, since `floatingpt` is initialized to `56.789`, the compiler decides that it must be of type `float`.

Line 12 introduces the `new` keyword. `new` is similar to the C++ concept, but does not call constructors (`bJou` does not share the C++ constructor model – see the Defining Types section). `new` allocates space on the heap and returns a pointer to it. `delete` – shown on lines 18 and 19 – frees that space up. So with that in mind, we can conclude that the compiler assigns the type `char*`, or a pointer to a character, to the variable `new_char`. The details of pointers and memory access in general are outside of the scope of this document, so I won't get into that. I will say, however, that other than a slight syntactic change where `@` dereferences pointers instead of `*` (shown on lines 14 and 16), pointers work the same way as in C.

A `print` statement is on line 16. If you are familiar with C's `printf()`,

this may look familiar. There are a couple of differences that I will highlight. The first is that `print` is a statement internal to the compiler rather than a function like `printf()`. The second difference is a display of the compiler's ability to use type information in other places – not just variable declarations. C's `printf()` uses a format string as the first argument that tells the program how to print the data that you give it. For example, if line 16 was written with `printf()`, the format string would look something like "num: %d, word: %s, floatingpt: %f, new_char: %c". The letters following the % character describe the types of the arguments you pass to it. With the bJou `print` statement, the compiler knows the types of the arguments you pass and can figure out the appropriate way to print them, which allows us to omit the extra letters in the format string.

bJou's type awareness extends to any expression and even things like array literals as shown in line 26. As an exercise, uncomment lines 24 and 27 to see how bJou's type system will detect simple errors with static arrays.

Constants

```

1  # demo2.bjou
2  # Constants
3
4  (proc main() {
5      const PI : float = 355.0 / 113.0
6      print "pi = %", PI
7      pi := PI
8      const TWO_PI := 2.0 * PI
9      print "2 x % = %", PI, TWO_PI
10     i := 1
11     # const ZERO := i - 1 # not a constant expression -- will not
    ↪ compile
12 })()
```

Constants can be thought of as labels for pre-computed (where applicable) expressions. On line 5 of 'tests/test/demo2.bjou', PI is defined as the constant expression 3.141593 computed from 355.0 / 113.0. This computation is made at compile time so that the program suffers no run-time performance costs. Constants take advantage of bJou's type awareness to deduce their type since an initialization is always required for constants. Initialization is probably a bad word to use since you cannot modify constants and they don't actually 'store' anything like variables do, but it is easier to

think of them similarly to variables to understand how they are created and used.

Lines 6 through 9 show the constant `PI` being used in places where any expression could be used – even in the declaration of another constant. I find it easiest to think about constants as placeholders for expressions that don't change throughout the program. Line 11 shows a situation where the constant `ZERO` is being set as an expression that contains a reference to a variable `i`. This is incorrect and will not compile since `i` is a variable, i.e. it may change at run-time.

Defining Types

```
1  # demo3.bjou
2  # Defining Types
3
4  type Base {
5      c : char
6      message : char*
7  }
8
9  type Derived extends Base {
10     num : int
11
12     proc create(c : char, message : char*, num : int) : Derived
13         return { Derived:
14             .c = c.(extern toupper(char) : char)(),
15             .message = message,
16             .num = num
17         }
18
19     proc printall(this)
20         print "{ %, %, % }", this.c, this.message, this.num
21 }
22
23 (proc main() {
24     d := Derived.create('b', "derived type", 12345)
25     d->printall()
26 })()
```

The code above demonstrates a few features that center around the idea of defining custom types. We'll begin on line 4 where a type called `Base` is defined. It has two data fields `c` and `message` which are declared with the standard variable declaration syntax inside of a type definition. The only 'gotcha' is that initialization expressions are not allowed within types.

The next type, `Derived` (line 9), show us how bJou allows for single inheritance. The `extends` keyword says that a type inherits from some base type and gains its members implicitly. On line 10 a field `num` is added to `Derived`, so it has `c`, `message`, and `num` in total. If you are curious about multiple inheritance, please see the section on Interfaces. Now notice lines 12 and 19. On these lines, two procedures are being defined for the `Derived` type. `create()` is similar to the concept of a ‘constructor’ in that it is a procedure that returns a new `Derived` instance, but that is really where the commonalities with C++ constructors end. Within a type definition, `this` can be used as syntactic sugar to create parameter declarations corresponding to a pointer to the instance of the type as shown on line 19 for `printall()`. In other words, in the context of a type definition for some `T`, `this` translates to `this : T*`.

Lines 24 and 25 show an example of creating an instance of a type and interacting with it. First notice the call `Derived.create(...)`. Here, we see that any procedure defined for a type can be referenced from the type name or through an instance as shown on the next line with `d->printall()`. The arrow operator does something related, but much different than the same operator in C. In bJou, the `->` operator acts like the access operator `(.)`, which works for both structure types and pointers to them. In bJou, the `.` operator also forwards the left hand side to the next call as the first argument. The difference, however is that `->` is used to take the address of the left hand side before the argument is forwarded to the next procedure call. See this in action on line 25 where `printall()` expects a `Derived*` as its first and only argument (from `this`). The `->` takes `d` (of type `Derived`) and forwards its address (type `Derived*`) as the argument to the call to `printall()`.

It is a critical point that procedures defined for types in bJou can always be referenced without a specific instance of the type. The importance of this is to allow for procedures to be used as first-class-citizens in all cases. This is explored in the next section.

Procedures

Talking to C

Interfaces

Templates

Modules

Non-Linear Compiler Logic

Beyond the Language

The Compiler as a Tool

Using bJou to Program the Compiler