# Programming Patterns for ASR

Brian Beckman, Ondřej Čertik

20 Mar 2023

## Contents

**1 Change Log** **1**

**2 What?** **1**

**3 Dynamically Bound Variables** **4**
3.1 Review of Lexical Binding . . . . . 4
3.2 Dynamically Binding `y` . . . . . 6

## 1 Change Log

2023-21-Mar :: Add section on dynamic binding.

## 2 What?

Let's simplify some C++ code. We can't easily run it in this document, because it depends on a big tree of includes, but we can look at it and figure out what it's trying to do.

The first thing to do is to indent this properly, to make the structure obvious and to expose patterns. Standard C++ indentation is less than helpful.

In Listing 1, we see pushing back (appending) an assignment statement to the body of something we're building up. The assignment statement has five parameters:

1. an *allocator*: `Allocator & al`
2. a *location*, e.g., `file`, `line`, `column`: `const Location & a_loc`
3. a *target* expression: `expr_t * a_target`, usually a `Var`
4. a source or *value* expression: `expr_t * a_value`
5. an *overloaded*, `stmt_t * a_overloaded`, meaning obscure at present

This particular assignment statement assigns or *binds* the result of a function call to a variable.

```
body.push_back(
    al, // Allocator  & al
    ASRUtils::STMT(
        ASR::make_Assignment_t(
            al,     // Allocator      & al
            loc,    // const Location & a_loc
            // expr_t * a_target
            ASRUtils::EXPR(
                ASR::make_Var_t(
                    al,          // Allocator      & al
                    loc,         // const Location & a_loc
                    return_var)),// symbol_t       * a_v
            // expr_t * a_value
            ASRUtils::EXPR(
                ASR::make_FunctionCall_t(
                    al,          // Allocator      & al
                    loc,         // const Location & a_loc
                    s,           // symbol_t       * a_name
                    s,           // symbol_t       * a_original_name
                    call_args.p, // call_arg_t     * a_args
                    call_args.n, // size_t           n_args
                    arg_type,    // ttype_t        * a_type
                    nullptr,     // expr_t         * a_value
                    nullptr)),   // expr_t         * a_dt
            nullptr // stmt_t * a_overloaded
            )));
```

Listing 1: Typical Node Creation

The `STMT` and `EXPR` parameters and arguments correspond to *terms* in the ASR grammar (see the file `asr.asdl`):

```
expr
  = ...
  | FunctionCall(symbol name, symbol? original_name,
         call_arg* args, ttype type, expr? value, expr? dt)
  | ...
  | Var(symbol v)
  | ...

stmt
  = ...
  | Assignment(expr target, expr value, stmt? overloaded)
  | ...
```

Note that the "factory functions"

- `ASR::make_FunctionCall_t`
- `ASR::make_Var_t`
- `ASR::make_Assignment_t`

are automatically generated from the ASR grammar.

Our job is to simplify the *use cases* for these functions – the way they're called in, for example, Listing 1, whilst preserving automatic generation from the grammar.

We'd like to see something like Listing 2, which removes two levels of indentation, removes the "always" parameters `al` and `loc`, and defaults several arguments of pointer type to **`nullptr`**. This is obviously easier on the eyes.

```
body.push_back_ALLOC(
    ASRUtils::make_Assignment_STMT(
        ASRUtils::make_Var_EXPR(
            return_var), // symbol_t       * a_v
        // expr_t * a_value
        ASRUtils::make_FunctionCall_EXPR(
            s,           // symbol_t       * a_name
            s,           // symbol_t       * a_original_name
            call_args.p, // call_arg_t     * a_args
            call_args.n, // size_t           n_args
            arg_type))); // ttype_t        * a_type
```

Listing 2: Simplified Node Creation

In these figures, I supplied the indentation and commentary with types and names of parameters by hand. Such indentation and commentary is very helpful and straightforward to implement in the ASR processor.

How do we get from Figure 1 to Figure 2? There are several patterns to exploit to reduce the size and noisiness of this code:

1. Every level, including the $0^{th}$ level, `push_back`, has an allocator ref, `& al`. This is probably either always the same or it can be supplied as a stack-disciplined, dynamically bound free variable.
2. All levels below the first have location arguments. These will be likely different for each sub-term – each `STMT` and `EXPR`. These should certainly be supplied as a stack-disciplined, dynamically bound free variable.
3. The immediate construction of `EXPR`'s around the `FunctionCall` node and `Var` nodes to `EXPR` is pure noise. These can be replaced by higher-level calls. Likewise with the immediate construction of a `STMT` from the `Assignment` node.
4. There are multiple ways to default arguments in C++. In our case, because the defaulted arguments are last in the parameter lists, *overloads* might be the easiest way to specify them. See this reference for more.[1]

# 3 Dynamically Bound Variables

## 3.1 Review of Lexical Binding

*Lexical binding* is the norm. Roughly, it means "the place where a variable acquires its value is obvious from just looking at the source code." It's a simple concept, but tied to a bunch of unfortunate, but necessary, terminology.

Consider a function `f` that multiplies its argument, the *bound variable* `x`, by the *free variable* `y`, and returns the product. Let `y` be a global variable for the moment; we'll modify that later. The function `f` is in *the scope*[2] of the global variable `y`.

```
int y = 6;

int f (int x) { int result = x * y; return result; }

int main () { printf ("yfree = %d, f(xbound=%d) ~~> %d\n",
                      y, 7, f(7)); }
```

```
yfree = 6, f(xbound=7) ~~> 42
```

- Summary of confusing terminology:
  - `y` is free in the body of `f`, the opposite of *bound*.
  - `y` is *not* bound in the body of `f`.
  - `y` is *lexically bound* in the body of `f` and in the bodies of any other functions in the *scope* of `y`.
  - `y` is *globally bound*, meaning its scope is at least the entire file below the lexical position at which `y` acquires a value. Its scope can be enlarged at link time with the help of `extern` declarations in other files.

[1] `https://en.cppreference.com/w/cpp/language/default_arguments`
[2] `https://en.wikipedia.org/wiki/Scope_(computer_science)`

- `x` more precisely, `x` is a *parameter* of the function `f`. When speaking imprecisely of `x` as an *argument*, we mean

  *the current value of the variable `x` in a particular invocation of `f`.*

  In the example above, it's clear that the argument `7` is the value of the parameter `x` in the invocation `f(7)`.

  - It's best to be very careful to distinguish parameters, which are variables, from arguments, which are values.
  - The process by which variables acquire values is called ***binding*** the variables. In the example above, all binding of `x` and `y` is lexical binding: it's obvious from reading the source code wherefrom `x` and `y` acquire their values.

- `x` is called a ***bound variable*** in the body of the function `f` because `x` is a parameter in the parameter list of `f`. That's the only thing that *bound variable*, as a single phrase, means: *in the parameter list*.

  - One must answer "in the parameter list of *what function*?" when speaking of a bound variable.
  - The term "bound variable" actually has another meaning in another context: it can mean that the variable has a value, without pertaining to any other properties of the variable. This is very bad usage of terminology, because it makes the term "bound variable" ambiguous without the context. We shall be very careful to avoid that ambiguity.

- `y` is called a ***free variable*** in the body of the function `f` because `y` is not a parameter in the parameter list of `f`. That's the only thing that *free variable* means: *not in the parameter list*. Thus, a variable may be free in the body of one function and bound in the body of another function.

  - Technically, `y` in some free occurrence and `y` in some bound occurrence are *not* the same variables, but the same *names* referring to different variables. This is a common source of confusion. One must take care to distinguish *names of variables*, which are symbols, or sometimes strings, from *names of storage locations*, which are the variables themselves. Yes, one may have the same name for different variables. The situation becomes more piquant with pointers and with C++ `&` references, which permit multiple names for the same variable.
  - One must answer "not in the parameter list of *what function*?" when speaking of a free variable.
  - The adjectives *bound* and *free*, without other qualifications, mean *"bound variable" in the body of some function* and *"free variable" in the body of some function*, with the terms "bound variable" and "free variable" taken as whole phrases.

- `y` is *also* a ***global variable*** in that little program above. That means that its value, when `y` occurs free in the body of the function `f` or in the body of any other function, is looked up in the global environment.

- `y` is called ***lexically bound*** in the body of `f`. That means that a human reading the program can look around the source code and find the places where `y` acquired a value. In this case, `y` is statically assigned the value `6` in the global environment.

  - Again, the terminology is confusing, and we avoid ambiguity by never taking shortcuts with the language. *Bound*, by itself, means one thing, the opposite of *free*, and *lexically bound* is not just the adverb *lexically* modifying the adjective *bound*, but a whole phrase denoting an entirely separate concept.

## 3.2 Dynamically Binding y

```
int y = 6;

static int push_y = y; // generalize this to a run-time stack

void bind_y(int value) { push_y = y; y = value; }

void unbind_y() { y = push_y; }

int f (int x) { int result = x * y; return result; }

int main () {
    bind_y(17);  // in Python, this would start a "with" block.
    printf ("yfree = %d, f(xbound=%d) ~~> %d\n",
                      y, 7, f(7));
    unbind_y();  // this would end the "with" block.
    printf ("but look, yfree is still %d!\n", y); }
```

```
yfree = 17, f(xbound=7) ~~> 119
but look, yfree is still 6!
```

Dynamic binding effects a temporary change to the value of a free variable, global or not, at run time. The value of a variable is no longer lexically obvious: one must look up a run-time stack and not just a lexical nest of bindings. In the example above, we have a pair of functions that work only for the variable `y`, and only for one level of dynamic binding, but that illustrate the concept.

- The scope of the dynamic binding of `y` is all the code between the call of `bind_y` and the call of `unbind_y`.
- In the scope of the dynamic binding of `y`, all free occurrences of `y` in the bodies of all functions, no matter how deeply nested, will have the value `17`, in this instance.
- To support nesting of binding scopes, replace the implementations of `bind_y` and `unbind_y` with implementations that employ a stack[3].
- Generalizing dynamic binding to any variable requires either special syntax and compiler help in the programming language, as with `binding` in Clojure[4] or special variables in Common Lisp[5], or a global dictionary between variable names as strings, e.g., `"x"`, and their variables, say `x`, along with a stack for each variable.
  - Both the footnoted articles above are worth your time to read if you're not familiar with dynamic binding.

[3] https://cplusplus.com/reference/stack/stack/
[4] https://clojuredocs.org/clojure.core/binding
[5] https://wiki.c2.com/?SpecialVariable