CSC3F002EP: Compilers | Project

Higher-order parameters & inner procedures

Starts: `2025-11-20`

---

1 SYNOPSIS

Throughout the semester, you have been working with the BX language, a small teaching language designed to explore the essential concepts behind compilation. You have already implemented a basic compiler for BX, giving you experience with parsing, type checking, intermediate representations, and code generation. In this project, you will build on that foundation by extending BX with two important features: higher-order parameters and inner function definitions.

In its current form, BX supports only first-order functions: functions can neither be passed around nor returned. Your task is to relax this restriction by allowing functions to be passed as parameters. BX will still prohibit returning functions or storing functions in variables. This constraint is deliberate: it ensures that higher-order behavior can be implemented using static links rather than general closure objects. Static links make it possible to preserve lexical scoping while keeping the runtime model simple and transparent.

You will also extend BX with inner function definitions, making it possible to declare functions inside other functions. Inner functions may reference variables from their surrounding scope, so your compiler will need to construct and maintain an appropriate chain of static links. With these links, nested functions will always be able to access the environment in which they were defined, even after several calls have occurred.

By completing this project, you will gain experience with:

- Updating the BX type system to support function-typed parameters,
- Handling nested scopes and name resolution for inner function definitions,
- Managing static links in function calls to maintain lexical scoping,
- Representing and invoking function values within the BX runtime model, and
- Extending your compiler's IR and backend to support higher-order parameters.

This following example illustrates how BX behaves once it supports higher-order parameters and nested function definitions. The function `apply_twice` expects two arguments: a function `f` of type `function(int) -> int` and an integer `x`. Its behavior is straightforward: it computes `f(x)` and then applies `f` again to that result. In other words, it returns `f(f(x))`.

Inside `main`, the variable `y` is initialized to `5`. Two nested functions are then defined. The first one, `increment`, simply adds `1` to its argument. Although it is a nested function, it does not depend on any variables from the surrounding scope. The second nested function, `add_y`, does use the variable `y` from

main. Because of this, each call to `add_y` must be able to access the value of `y` through the static link established when the function is created.

The call `print(apply_twice(increment, 10))` demonstrates passing a nested function as a parameter. The result is `increment(increment(10))`, which evaluates to `12`. The next call, `print(apply_twice(add_y, 16))`, shows the effect of a nested function that captures an enclosing variable. The first application of `add_y` produces `16 + y = 21`, and applying `add_y` again yields `21 + y = 26`. The program therefore prints `12` and `26`, confirming that function parameters and static scoping work as expected.

```
def apply_twice(f : function(int) -> int, x: int) : int {
  // Apply f to x two times
  return f(f(x));
}

def main() {
  var y = 5 : int;

  // A nested function inside Main
  def increment(n:  int) : int {
    return (n + 1);
  }

  // Another nested function that uses a captured variable
  def add_y(n : int) : int {
    return (n + y);
  }

  // Pass a nested function as a parameter
  print(apply_twice(increment, 10)); // prints 12

  // Pass another nested function that captures y
  print(apply_twice(add_y, 16)); // prints 26
}
```

2 EXTENDING THE BX LANGUAGE

The BX language in this project is a strict superset of that of lab 4. You are free to start from the reference implementation.

To support higher-order parameters and nested function definitions, the BX syntax requires only two extensions. First, the type syntax must be expanded to allow function types. In addition to the base types `int` and `bool`, you may now write types of the form:

```
  function(T1, T2, ..., Tn) -> T
```

The parameter types `T1`, `T2`, $\cdots$, `Tn` may themselves be higher-order types (including other function types). The return type `T` must be a first-order type, which now includes int, bool, or the new keyword `void`. The type `void` is used to indicate procedures. As before, function types may be used only in parameter declarations; return types and variable declarations continue to forbid higher-order types.

Second, `def` declarations may now appear inside the body of another function or procedure. A nested

`def` uses the same syntax as a top-level definition and may be placed anywhere a statement is allowed, including inside nested blocks. For example:

```
def main() {
  var y = 5 : int;

  def inc(n : int) : int {
    return n + 1;
  }

  if (y < 10) {
    def show(n : int) : void {
      print(n);
    }
    show(y);
  }
}
```

We provide below the extensions to the BX grammar in EBNF form:

⟨funrettype⟩ ::= ··· | `"int"` | `"bool"` | `"void"`

⟨funtype⟩ ::= ··· | `"function"` `"("` ⟨typelist⟩$^{?}$ `")"` `"->"` ⟨funrettype⟩

⟨type⟩ ::= ··· | ⟨funtype⟩

⟨typelist⟩ ::= ⟨type⟩ (`","` ⟨type⟩)*

⟨stmt⟩ ::= ··· | ⟨procdef⟩

⟨procdef⟩ ::= ···

3 EXTENDING THE TYPE CHECKER

With the introduction of function types, higher-order parameters, and nested declarations, the type-checker must enforce the following rules:

Function types. The type-checker must support function types of the form `function(T1, …, Tn) -> T`. Parameter types may themselves be function types, but the return type must always be first-order (`int`, `bool`, or `void`).

Where function types may appear. Function types are allowed only in parameter declarations. Variable declarations and function return types must remain first-order. Programs violating this restriction must be rejected.

Nested definitions and lexical scope. Nested `def` declarations extend the current environment according to lexical scope. Each nested function has access to the variables of the scopes that surround it.

Uniqueness of names in a scope. Two inner function declarations in the same lexical scope may not have the same name. The type-checker must detect and reject redeclarations in a single scope.

No recursion among inner functions. Inner functions cannot be recursive (neither directly nor mutually). They are type-checked in a top-down order; only outer functions and previously declared siblings are visible at a given point.

Lifetime of inner functions. An inner function does not survive beyond the lexical scope in which it is declared. It may not be returned or stored (both are impossible anyway because only parameters may have function types). Once the enclosing block ends, the inner function name becomes out of scope.

Function values and calls. Each `def` introduces an identifier with a corresponding function type. A call must be of the form `x(...)` where x is such an identifier bound to a function type. Argument types must match parameter types, and the call must be used in a context compatible with its return type.

4 VARIABLE CAPTURE

A variable capture occurs when a nested function refers to a variable that is declared in one of its surrounding lexical scopes rather than inside the function itself. In this situation, the inner function *captures* the variable from the environment in which it is defined.

Since this project will use capture by reference, the inner function does not receive its own copy of the variable. Instead, it refers directly to the same storage location as the variable in the outer scope. Any read or write performed by the inner function affects the original variable.

For example:

```
def main() {
  var y = 5 : int;

  def inc_y() : void {
    y = y + 1;       // captures and read/modifies y
  }

  inc_y();
  print(y);          // prints 6
}
```

Here, `inc_y` captures `y`. Updating `y` inside `inc_y` changes the `y` declared in `main`, because both refer to the same memory location.

Every nested function may perform variable captures. For the later phases of the compiler, the type-checker should explicitly determine which variables are captured by each inner function. To support this, the type-checker should compute, for every inner function, the set of captured variables and record this set directly in the corresponding AST node. Later compiler passes can then rely on this information without recomputing it.

When building these capture sets, it is strongly recommended not to record variables by their source names alone. Different lexical scopes may declare variables with the same name, so names can be ambiguous. Instead, the type-checker should assign each declared local variable a unique internal identifier, and all captures should be recorded using these identifiers. This guarantees an unambiguous mapping between

captured variables and their defining scopes, and it simplifies all subsequent analyses and transformations of the program.

5 STATIC LINKS

A static link is a run-time mechanism that allows a nested function to access the variables of the lexical scopes in which it was defined. It connects each activation record (stack frame) to the frame of the function's lexical parent, that is, the function inside which it was syntactically declared.

You can think of a static link as a pointer stored in each frame. When a function `f` is called, its frame contains a static link that points to the frame of the function that lexically encloses `f`. If `f` is declared inside `g`, then every call to `f` has a static link referring to the most recent activation of `g`. If `f` is declared at the top level, its static link is usually null or irrelevant, since it has no enclosing scope.

When a nested function accesses a captured variable, it follows its static link zero or more times to reach the correct frame. For example, if a function `h` is declared inside `g`, and `g` is declared inside `f`, then a reference in `h` to a variable of `f` requires following two static links: first from `h`'s frame to `g`'s frame, then from `g`'s frame to `f`'s frame.

In this project, static links implement capture by reference: the inner function does not hold copies of outer variables, but instead uses static links to locate the original variables in their respective frames. This preserves lexical scoping at run time and ensures that any modification to a captured variable affects the variable in the outer scope itself.

Static links are determined lexically, not dynamically: the static link of a function depends only on where it is defined in the source program, not on where or by whom it is called. This is what makes lexical scoping work correctly.

When a variable is captured by an inner function, it must always be reachable through the static-link chain. This implies that every captured variable must have a reliable stack location in its activation record. Even if the compiler keeps a copy of the variable in a register for optimization purposes, the version stored in the activation record is the one that inner functions will see when called through a fat pointer.

Because a higher-order call may occur at any point, the compiler must maintain the following invariant: if a variable is captured and also lives in a register, then its stack slot must always contain the most recent value before any higher-order call can occur.

In practice, this means that the compiler must ensure that the stack slot is updated whenever the register value changes before any instruction that might invoke a function through a parameter. This guarantees that inner functions, which access captured variables via the static links, always observe the correct, up-to-date value, regardless of register allocation decisions.

6 FAT POINTERS

A fat pointer is the runtime representation of a function value in the extended BX language. Since functions may be passed as parameters, the compiler must treat them as values. However, a function value is not just a code address: if the function is nested, it must also carry the static link that allows it to access variables from the lexical scope in which it was defined.

For this reason, a fat pointer is defined as a pair consisting of 1) A code pointer, i.e. the address of the function's entry point in the generated code, and 2) A static link, i.e. a pointer to the activation record of the function's lexical parent at the moment the function value is created.

Whenever a nested function is passed as an argument, the compiler constructs such a pair. When the function is later invoked through a parameter, the call uses the code pointer to jump to the correct function body and uses the stored static link as the function's static link for that invocation. This ensures that all captured variables are accessed in the correct lexical environment.

Thus, in this project, a function value is always represented at runtime as a fat pointer = (code pointer, static link), and higher-order calls always operate on such pairs.

7 EXTENDING TAC WITH FAT-POINTERS AND THE MAXIMAL-MUNCH ALGORITHM

To come.

8 FINAL DELIVERABLE

The final deliverable for this project is a fully functional compiler for the extended BX language. You may implement their compiler in `Python` or in *any other programming language* of your choice. Your submission must include a one-page README that briefly explains your design choices and directs the reader to the relevant parts of your implementation.

If the compiler is written in Python:

- You must submit a file named `bxc.py`.
- The compiler must take as input the filename of a BX program, written as `name.bx`, and must generate the corresponding assembly code in a file named `name.s`.
- If your Python version requires additional files or modules, you may instead submit a ZIP archive named `bxc.zip`, containing `bxc.py` in the root directory along with all necessary dependencies.

If the compiler is written in another programming language:

- You must submit a ZIP archive whose root directory contains a `Makefile`.
- The default rule of the `Makefile` must compile your project and produce an executable named `bxc.exe` in the same directory.
- Running `bxc.exe name.bx` must generate the corresponding file `name.s`.

If you want the generated assembly to be compiled on the submission server, place a file named `ARCH` at the root of your source directory. This file should contain either `x86_64-linux-gnu` (for Linux `x86_64`) or `aarch64-linux-gnu` (for Linux `Aarch64`). Note that Linux `Aarch64` differs from Darwin `Aarch64` in that symbol names (functions and global variables) do *not* require a leading underscore.

If you specify `x86_64-linux-gnu`, your program will be executed on the submission grader, and its output will be visible in the grader log.

You may also include a runtime library by adding a file named `bxruntime.c` at the root of the source tree. This runtime will be compiled and linked with your program automatically.

Collaboration Policy. You must work alone on the implementation. Thinking collectively and discussing ideas or strategies is allowed, but sharing code is strictly forbidden. Two submissions containing identical or nearly identical code will be treated as a violation of academic integrity.

Deadline. The final version of your project must be submitted no later than **December 22, 2025**.