BX Project – Part 01–03 (Parsing and Front-End)

Jagan Koipalil
Compilers 2026

December 1, 2025

1 Overview

In this report I describe the changes made in `bxc.py` (implemented in the file `bxc_v2.py`) to satisfy the requirements of the project parts 01–03:

- Extend the BX language with function types and nested procedure definitions.
- Extend the type checker to support higher–order parameters while keeping returns and local variables first–order.
- Compute variable capture sets for nested procedures, using a unique identifier for each variable binding.

At this stage there is *no change* to the intermediate representation or code generation: the work is purely on the front–end (parsing and static analysis). The implementation remains backwards–compatible with the original Lab 3/4 test programs.

2 Language Extensions

2.1 Function Types

The language now supports higher–order function types in parameter position. Syntactically, a function type is written

$$\texttt{function}(T_1, \ldots, T_n)\ \texttt{->}\ R$$

with the following constraints:

- Each argument type T_i may itself be a function type, allowing arbitrarily nested higher–order types.
- The result type R must be *first–order*: one of `int`, `bool`, or `void`.

These types are represented in the AST by the type

```
Ty = Union[str, FunTy]

@dataclass(frozen=True)
class FunTy(AST):
    param_tys: Tuple[Ty, ...]
    ret_ty: str  # 'int' | 'bool' | 'void'
```

The grammar is extended with a new nonterminal `type`:

```
def p_type_int(p):
    'type : INT'
    p[0] = 'int'

def p_type_bool(p):
    'type : BOOL'
    p[0] = 'bool'
```

```
def p_type_fun(p):
    'type : FUNCTION LPAREN type_list_opt RPAREN ARROW funrettype'
    p[0] = FunTy(tuple(p[3]), p[6])
```

Here `funrettype` is restricted to `int`, `bool` or `void`, enforcing that function return types are first–order.

2.2 Placement of Function Types

The type checker enforces the placement constraints:

- **Parameters:** may be first–order or function types, recursively.
- **Return types:** first–order only (`int`, `bool`, `void`).
- **Local variables:** only first–order, and in this fragment only `int` and `bool` are allowed.

This is captured by predicates:

```
def is_first_order_type(ty: Ty) -> bool:
    return isinstance(ty, str) and ty in ('int', 'bool', 'void')

def is_valid_param_type(ty: Ty) -> bool:
    if isinstance(ty, str):
        return ty in ('int', 'bool')
    if isinstance(ty, FunTy):
        return all(is_valid_param_type(pt) for pt in ty.param_tys) \
               and is_first_order_type(ty.ret_ty)
    return False
```

and by explicit checks in parameter, local variable, and return type positions.

2.3 Nested Procedure Definitions

The language now allows nested procedures via a new statement form:

```
@dataclass
class SProcDef(Stmt):
    proc: ProcDecl
```

The grammar contains:

```
def p_stmt_procdef(p):
    'stmt : DEF IDENT LPAREN params RPAREN ret_annot block'
    p[0] = SProcDef(ProcDecl(p[2], p[4], p[6], p[7]))
```

so any block can contain inner definitions of the form

```
def inner(x: int): int {
  ...
}
```

Inner procedures are lexically scoped: they can see surrounding parameters and local variables according to the usual block structure.

To handle braces as statements (e.g. in `if` or `while`), I also added the rule:

```
def p_stmt_block(p):
    'stmt : block'
    p[0] = p[1]
```

2.4 Syntactic Niceties and Backwards Compatibility

To maintain compatibility with the lab tests:

- I treat `ret;` as shorthand for `return;` via:

```
reserved['ret'] = 'RET'

def p_stmt_ret_short(p):
    'stmt : RET SEMI'
    p[0] = SReturn(None)
```

- I resolve the dangling-`else` ambiguity using precedence:

```
precedence = (
    ...,
    ('right', 'ELSE'),
)

def p_stmt_if(p):
    'stmt : IF LPAREN expr RPAREN stmt %prec ELSE'
    p[0] = SIfElse(p[3], p[5], None)

def p_stmt_if_else(p):
    'stmt : IF LPAREN expr RPAREN stmt ELSE stmt'
    p[0] = SIfElse(p[3], p[5], p[7])
```

All original Lab 3/4 test cases (`cfg_diamond.bx`, `cfg_loop_break.bx`, etc.) still parse and type–check.

3 Type Checker Extensions

3.1 Environments and Function Values

There are now two kinds of environments:

- **Variable environments** `var_env_stack`: a stack of frames mapping variable names to pairs (*type*, *varID*).
- **Function environments** `fun_env_stack`: a stack of frames mapping function names to `FunTy`.

Top–level procedures are collected in a global `fun_env_global` mapping from name to `FunTy`. The entry point `main` must be present.

Every `def` (top–level or nested) introduces a *function value* of type `FunTy`. Calls are allowed whenever an expression has a function type:

- `x(...)` where `x` is a function–typed variable (parameter or captured outer variable).
- `f(...)` where `f` is a function name present in the function environment.

The type checker enforces that arguments match parameter types and that the declared return type is respected on all paths (for non–void functions).

The built–in `print` is treated specially:

- `print(e)` is rewritten to `__bx_print_int(e)` or `__bx_print_bool(e)` depending on the type of `e`.
- `print` returns `void`.

3.2 Lexical Scope and Nested Definitions

For each procedure `ProcDecl`, including nested ones, I invoke:

```
typecheck_proc(pd, fun_ty, outer_var_env, fun_env_stack)
```

where:

- `outer_var_env` is a list of variable environment frames visible from outside the procedure (its lexical context).
- `fun_env_stack` is the function environment visible inside the procedure.

Within `typecheck_proc`:

- The parameters are added as a new frame with fresh variable IDs.
- New locals are introduced in the top frame of `var_env_stack`.
- Nested `def` statements are type–checked in a fresh copy of the current environments; once successfully checked, the nested function is bound in the current function environment for subsequent statements.

This ordering ensures that:

- Top–level recursion is allowed (procedures see each other through the global environment).
- Nested procedures cannot be recursive or mutually recursive: the body of a nested procedure is type–checked before the procedure's name is inserted into the function environment of its enclosing scope.

3.3 First–Order Restrictions

The type checker enforces that:

- Procedure results are first–order (`int`/`bool`/`void`).
- Local variables `var x = ... : T;` are restricted to `T` equal to `int` or `bool`. Attempting to declare a function–typed local variable is rejected.
- Non–void procedures must return on all control–flow paths; this is checked using a boolean flag returned by the statement checker.

4 Variable Identifiers and Capture Sets

4.1 Unique Variable IDs

To follow the specification's recommendation, each variable binding (parameter or local) is assigned a unique integer identifier (`vid`):

```
next_var_id = 0
def fresh_var_id() -> int:
    nonlocal next_var_id
    vid = next_var_id
    next_var_id += 1
    return vid
```

The AST nodes are extended with these IDs:

```
@dataclass
class Param(AST):
    name: str
    ty: Ty
    vid: int = -1

@dataclass
class SVar(Stmt):
    name: str
    init: Expr
    ty_annot: Ty
    vid: int = -1
```

When parameters and local variables are inserted into the environment, they are assigned a fresh `vid`. Variable expressions `EVar(name)` look up the corresponding pair (*type*, *vid*) in the environment.

4.2 Definition of Captures

Each `ProcDecl` now has a field:

```
@dataclass
class ProcDecl(AST):
    name: str
    params: List[Param]
    ret_ty: Ty
    body: SBlock
    captures: Set[int] = field(default_factory=set)
```

The set `captures` is defined as follows: a procedure P captures a variable with ID v iff:

- v is declared in a lexically enclosing scope, i.e. not in the parameter or local frames of P itself, and
- P reads from, writes to, or calls a function via that variable.

Concretely:

- We maintain a set `local_ids` containing all variable IDs declared as parameters or locals of the current procedure.
- Whenever we resolve a variable occurrence `EVar(name)` or the LHS of an assignment:
  - We look up its `vid`.
  - If `vid` is *not* in `local_ids`, we add it to `pd.captures`.
- Function–typed variables used as callees (e.g. `f(x)` where `f` is a parameter of function type) are treated identically: if `f` comes from an outer scope, its ID is added to the capture set.

This definition agrees with the intuitive notion of a *captured* variable: variables of the enclosing scopes that must remain live as long as the function value exists.

4.3 Inspecting Captures

For debugging, I added a `-dump-captures` mode. It first builds a mapping from variable IDs to names by walking parameters and local declarations, and then prints for each procedure:

```
def <name> captures {x, y, ...}
```

with indentation reflecting nesting. For example, for:

```
def apply_twice(f : function(int) -> int, x: int) : int {
  return f(f(x));
}

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

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

  def add_y(n: int): int {
    return n + y;
  }

  print(apply_twice(increment, 10));
  print(apply_twice(add_y, 16));
}
```

the compiler reports:

```
def apply_twice captures {}
def main captures {}
  def increment captures {}
  def add_y captures {y}
```

which matches the expected capture behaviour: only `add_y` uses the outer variable `y`, and none of the procedures capture `x` or `f` because these are parameters of their own scopes.

5 Testing and Compatibility

I tested the implementation on:

- The original Lab 3/4 tests: `cfg_loop_break.bx`, `cfg_diamond.bx`, `cfg_unreachable.bx`, etc. All parse and type–check and the driver prints `OK`.
- Higher–order examples involving function parameters, nested definitions, and different patterns of captures, verifying the reported capture sets.

The extended front–end is therefore backwards–compatible and satisfies the project requirements for parts 01–03. Later parts of the project (closure representation, static links, fat pointers, and their impact on the intermediate language and code generation) can build directly on this front–end.

6 Conclusion

In this phase I extended the BX compiler front–end to support higher–order function parameters and nested procedure definitions while preserving a first–order execution model (no functions returned or stored in variables). The type checker enforces the placement restrictions for function types and ensures sound use of function values. Each variable binding receives a unique identifier, and each procedure is annotated with an explicit capture set of outer variables it depends on.

This information will be essential in the next phases of the project, where we will introduce closure representations and adapt the calling convention and code generation to handle higher–order functions efficiently without violating the stack discipline.