Symbol Table for SPL (COS341 Project)

This SymbolTable provides **static scoping** for SPL programs and supports **semantic checks** and **intermediate representation (IR) renaming**. It manages variables, parameters, procedures, and functions while maintaining unique internal names for code generation.

Core Concepts

1. **Scopes**
   - Represented as a **stack of dictionaries**, each mapping names to SymbolInfo.
   - New scopes are pushed with enter_scope(kind, node) and popped with exit_scope().
   - Scope levels:
     - 0: No scope yet
     - 1: Global scope
     - 2+: Nested scopes (functions, procedures, blocks)
2. **Symbols**
   - Each declared name is represented by a SymbolInfo object:

     ```
     @dataclass
     class SymbolInfo:
       name: str          # original source name
       kind: str          # 'var', 'proc', 'func', 'param'
       decl_type: str|None  # 'numeric', 'boolean', 'string', or None for type-less
       scope_level: int     # depth in the scope stack
       unique_name: str     # unique IR name (v_name_1, v_name_2, ...)
       node_id: int         # id of the AST node
       extra: Dict[str, Any] = None  # optional metadata
     ```
3. **Unique Naming**
   - _gen_unique_name(base_name) ensures each declaration gets a unique IR-safe name.
   - Example: v_x_1, v_x_2, etc.

API Overview

Scope Management

```
st.enter_scope("Global", program_node)  # push new scope
st.exit_scope()                   # pop current scope
st.current_scope_level()           # get depth
st.current_scope_name()            # get kind of current scope
```

Declarations

```
st.declare_var(name, node, decl_type="numeric")

st.declare_param(name, node, decl_type="numeric")

st.declare_proc(name, node)

st.declare_func(name, node)
```

- Variables and parameters can have types; procedures and functions are type-less.
- Duplicate names **within the same scope** are prohibited.

Lookup

```
st.lookup(name)        # returns SymbolInfo or None

st.assert_exists(name)  # raises if name not found
```

- Searches **from innermost to outermost scope**.

Semantic Checks

- **Global name clashes**:

```
st.check_no_global_name_clashes()

# Enforces: no variable/procedure/function name conflicts at global level
```

- **Parameter shadowing**:

```
st.check_no_shadowing_of_params(param_names, local_names)

# Ensures local variables do not shadow function/procedure parameters
```

Utilities

```
st.get_unique_name(name)      # return IR-safe unique name

st.get_scope_snapshot()       # shallow copy of all scopes

st.find_symbol_by_node(node)  # find SymbolInfo by AST node
```

Internal Structure

- _scopes: list of dictionaries (scope stack)
- _scope_meta: scope kind + node ID (debugging/inspection)
- _name_counters: global counter for generating unique names
- _global_procs / _global_funcs: track globally declared procedures/functions

---

Example Usage

```
st = SymbolTable()
st.enter_scope("Global", program_node)
x_info = st.declare_var("x", var_node)
st.declare_proc("print", proc_node)

# Nested function scope
st.enter_scope("Function", func_node)
st.declare_param("y", param_node)
st.assert_exists("x")  # looks up in outer scope
st.exit_scope()
st.exit_scope()
```

Notes

- **Static scoping**: each lookup traverses the stack from inner to outer scopes.
- **IR renaming**: unique_name ensures no conflicts during code generation.
- **Checks**: check_no_global_name_clashes and check_no_shadowing_of_params help enforce SPL's semantic rules.

Task 4 requirements:

1. **Data Structure**

Task requirement: "hash map of stacks" for nested scopes, push on entry, pop on exit.

Implementation:

- _scopes is a **list of dictionaries** (List[Dict[str, SymbolInfo]]).
- Each dictionary represents a **scope**, mapping identifier names → SymbolInfo.
- enter_scope() **pushes** a new dictionary.
- exit_scope() **pops** the dictionary.

This **naturally implements static scoping**, as inner scopes override outer scopes, and lookups search from innermost → outermost (lookup() uses reversed(self._scopes)).

2. **Information Stored Per Identifier**

Task requirement: Store type and scope level, plus a unique name for IR.

Implementation: SymbolInfo contains:

- decl_type: 'numeric' | 'boolean' | 'string' | None (type-less for procs/funcs)
- scope_level: integer representing nesting depth
- unique_name: IR-safe generated name (v_x_1, v_x_2, etc.)

Other fields (kind, name, node_id, extra) are extra but useful for semantic checks and AST linking.

3. **Nested Scopes & Static Scoping**

- Nested procedures/functions automatically create **inner scopes**.
- Lookups traverse the stack **inner → outer**, exactly as required for static scoping.

4. **IR Name Generation**

- _gen_unique_name() ensures **each identifier gets a unique internal name**, which can be used directly in IR code generation.
- Maintains global counters per original name.