README_A4_Symbol_Table

Ada Symbol Table Implementation

Overview

This project implements a symbol table for an Ada compiler, a crucial component for managing identifiers and their associated information during the compilation process. The symbol table tracks variables, constants, procedures, and their attributes across different scopes.

Key Features

- Hash Table Implementation: Efficient lookup using a hash table with chaining for collision resolution
- Scope Management: Support for lexical scoping via depth parameters
- Entry Types: Support for variables, constants, and procedures with type-specific information
- Integration Support: Compatible with the compiler's token system

---

Core Components

AdaSymbolTable Class

The main implementation of the symbol table, providing the following operations:

- insert: Add a new entry to the symbol table
- lookup: Find an entry by its lexeme (identifier)
- deleteDepth: Remove all entries at a specific depth (scope)
- writeTable: List all entries at a specific depth

TableEntry Class

Represents individual entries in the symbol table with the following attributes:

- Common: lexeme, token type, scope depth

- **Variable-specific**: type, memory offset, memory size
- **Constant-specific**: type, value
- **Procedure-specific**: local variable size, parameter count, return type, parameter list

Supporting Types

- **VarType**: Enumeration for variable types (CHAR, INT, FLOAT)
- **EntryType**: Enumeration for entry types (VARIABLE, CONSTANT, PROCEDURE)
- **ParameterMode**: Enumeration for parameter passing modes (IN, OUT, INOUT)
- **Parameter**: Class for procedure parameters with type and passing mode

---

Usage Examples

Creating a Symbol Table

```python
from Modules.AdaSymbolTable import AdaSymbolTable, VarType, EntryType,
ParameterMode, Parameter

# Create a new symbol table (default size is 211)
symbol_table = AdaSymbolTable()

# Optionally specify a custom table size
small_table = AdaSymbolTable(table_size=101)
```

Adding Entries

```python
# Insert a variable
var_entry = symbol_table.insert("counter", token, 1)
var_entry.set_variable_info(VarType.INT, 0, 4)

# Insert a constant
const_entry = symbol_table.insert("PI", token, 1)
const_entry.set_constant_info(VarType.FLOAT, 3.14159)

# Insert a procedure
params = [
    Parameter(VarType.INT, ParameterMode.IN),
    Parameter(VarType.FLOAT, ParameterMode.OUT)
]
proc_entry = symbol_table.insert("calculate", token, 1)
proc_entry.set_procedure_info(16, 2, VarType.INT, params)
```

Looking Up Entries

```
# Look up an entry by lexeme
entry = symbol_table.lookup("counter")
if entry:
    print(f"Found entry: {entry}")

    # Check entry type
    if entry.entry_type == EntryType.VARIABLE:
        print(f"Variable type: {entry.var_type.name}")
    elif entry.entry_type == EntryType.CONSTANT:
        print(f"Constant value: {entry.const_value}")
    elif entry.entry_type == EntryType.PROCEDURE:
        print(f"Procedure parameters: {len(entry.param_list)}")
```

Managing Scopes

```
# Add entries at different depths
symbol_table.insert("global_var", token, 1).set_variable_info(VarType.INT, 0,
4)
symbol_table.insert("local_var", token, 2).set_variable_info(VarType.INT, 4,
4)

# List entries at depth 2
depth2_entries = symbol_table.writeTable(2)
print(f"Entries at depth 2: {depth2_entries}")

# Delete all entries at depth 2 (when exiting the scope)
symbol_table.deleteDepth(2)
```

Implementation Details

Hash Function

The symbol table uses a variation of the hashpjw algorithm, which is optimized for compiler symbol tables:

```
def _hash(self, lexeme: str) -> int:
    h = 0
    g = 0
    for c in lexeme:
        h = (h << 4) + ord(c)
        g = h & 0xF0000000
        if g != 0:
            h = h ^ (g >> 24)
            h = h ^ g
    return h % self.table_size
```

Collision Resolution

Collisions are resolved using chaining, where entries with the same hash value are linked together in a singly linked list. This approach is memory-efficient and performs well for the expected number of entries in a typical program.

Testing

The implementation includes comprehensive tests covering:

- Basic insertion and lookup
- Scope management with deleteDepth
- Entry management with writeTable
- Hash collision handling
- Integration with the compiler's token system
- Stress testing with many entries

Run the tests with:

```
python A4_Ada_Symbol_Table/TestSymbolTable.py
```

Integration with Other Compiler Components

The symbol table is designed to integrate seamlessly with other compiler components:

- Lexical Analyzer: Works with the Token objects produced by the lexical analyzer
- Parser: Supports the parser by providing fast symbol lookup and scope management
- Semantic Analyzer: Stores type information needed for semantic checks
- Code Generator: Provides memory offsets and sizes for code generation

---

Author

John Akujobi