Priniciple of Compiler Designer
Individual Assignment
Assignment 1:
NO:42

Helen Adugna:
Id:1506501
Section:B

Subission Date:27/04/2018 ec
Submitted to:Wondmu Baye

1.Handle symbol table for modules and imports. You will implement symbol resolution across modules, managing exported and imported symbols. The assignment includes detecting conflicts and ambiguous references.

Scope & Symbol Table for Modules and Import Detailed Explanation

Introduction to Symbol Tables and Scope

- ✓ Symbol Table: A central data structure in a compiler that stores information about identifiers (variables, functions, classes, modules).
- ✓ Scope: Defines the region in a program where an identifier is visible and accessible.

- ✓ Modules & Imports: Modern programming languages organize code into modules, which can export symbols and import symbols from other modules.

Importance:

- ✓ Ensures identifiers are correctly declared and used.
- ✓ Prevents naming conflicts and ambiguous references.
- ✓ Supports semantic analysis, type checking, and correct code generation.

Structure of a Symbol Table

A symbol table typically includes:

Field	Description
Name	Identifier name (variable, function, module)
Type	Data type or function signature
Scope/Module	Scope where the identifier is valid
Access/Export	Indicates if the symbol is accessible outside module
Memory Location	Offset for code generation
Attributes	Const, static, visibility, parameters (for functions)

Hierarchical Symbol Table Design

- ✓ Global Table: Stores all module-level symbols.
- ✓ Local Table: Stores block-level symbols, e.g., within functions.
- ✓ Module Table: Tracks exported/imported symbols.

Diagram Idea:

Global Table (Module A)
  ├– function foo
  │ └– local variables
  ├– variable x
Module Table (Module B)
 └– imported symbols: foo from Module A

Explanation:

✓ Local symbol tables override module/global symbols (shadowing).

✓ Module symbol tables maintain separation for modular programming.

Exporting and Importing Symbols

- Exporting Symbols:

✓ A module declares which symbols can be accessed by other modules.

Example:

```
export function add(x, y)
export const PI = 3.14
```

- Importing Symbols:

✓ A module can import specific symbols or all symbols from other modules.

Example:

```
import { add, PI } from ModuleA
```

- Symbol Resolution Steps:

✓ Check local symbol table.
✓ Check imported symbols.
✓ Check global/module symbols.
✓ Detect conflicts or ambiguous references

Diagram Idea:

Module B
 └– import add, PI from Module A
  └– checks Module A's symbol table

Conflict and Ambiguity Detection

Common Conflicts:

1.Duplicate Definitions in Same Module:

```
var x
var x  # Error: redeclaration
```

2.Conflicting Imports:

```
import { add } from ModuleA
import { add } from ModuleC  # Conflict: ambiguous reference
```

3.Shadowing:

- ✓ Local variable can hide a module/global symbol.

```
import PI from ModuleA
var PI = 3  # local PI shadows imported PI
```

Resolution Strategies:

- ✓ Fully qualified names: ModuleA.add()
- ✓ Namespace aliasing:

```
import { add as addA } from ModuleA
```

- ✓ Static checking: Compiler reports errors/warnings for ambiguity.

Step-by-Step Example of Symbol Resolution

Scenario:

- ✓ Module A exports: function add(x, y)
- ✓ Module B exports: function multiply(x, y)
- ✓ Module C imports both:

```
import { add } from ModuleA
import { multiply } from ModuleB
var result = add(5, 3) + multiply(2, 4)
```

Symbol Table Construction:

Module	Symbol	Type	Scope	Exported
A	add	func	A	yes
B	multiply	func	B	yes
C	result	int	C	no
C	add	func	imported	yes
C	multiply	func	imported	yes

Resolution Steps:

1. add(5,3) → Check local table → imported table → ModuleA → resolved.
2. multiply(2,4) → Check local → imported table → ModuleB → resolved.
3. No conflicts → symbol resolution successful.

Diagram Idea:

C imports:
add → ModuleA.add
multiply → ModuleB.multiply

Semantic Errors Related to Symbol Tables

Undeclared Variable:

```
x = 5  # Error: x not declared
```

Type Mismatch:

```
var x: int
x = "hello"  # Error: incompatible types
```

Ambiguous References:

```
import { add } from ModuleA
import { add } from ModuleC
add(2,3)  # Error: ambiguous reference
```

Illegal Operations:

- ✓ Calling a non-function
- ✓ Using symbols outside their scope

Compiler Action:

- ✓ Detect errors during semantic analysis, before code generation.
- ✓ Emits clear error messages with module and line number.

Integration with Compiler Pipeline

Flow:

1. Lexical Analysis: Source code → Tokens
2. Syntax Analysis: Tokens → AST
3. Semantic Analysis:
   - ✓ Construct symbol tables
   - ✓ Manage scope hierarchies
   - ✓ Resolve imported/exported symbols
   - ✓ Type checking and validation
4. IR Generation: AST → Three-Address Code (TAC), SSA
5. Optimization: Local/global/loop/interprocedural optimizations
6. Code Generation: Generate target assembly using symbol table info

Diagram Idea:

Source Code
↓
Lexical Analyzer → Tokens
↓
Parser → AST
↓
Semantic Analyzer → Symbol Table + Scope + Type Checking
↓
IR Generator → Optimization → Code Generator → Executable

Best Practices

- ✓ Separate symbol tables per module to maintain modularity.
- ✓ Track exports and imports explicitly.
- ✓ Use qualified names or aliases to resolve conflicts.
- ✓ Detect ambiguities and redeclarations during semantic analysis.
- ✓ Maintain type information, visibility, and other attributes.
- ✓ Apply static checks to enforce rules before code generation.

Additional Practical Example:
Scenario: Shadowing + Import Conflict

```
ModuleA:
  export var x = 10
  export function add(a,b) return a+b
ModuleB:
  export var x = 20
ModuleC:
  import { x, add } from ModuleA
  import { x } from ModuleB
  var y = x  # Ambiguity: which x?
```

Solution:
Fully qualify symbols:

```
var y = ModuleA.x
```

Or use aliasing:

```
import { x as xA } from ModuleA
import { x as xB } from ModuleB
var y = xA
```

Diagram:
ModuleC symbol table:
- add → ModuleA.add
- xA → ModuleA.x
- xB → ModuleB.x

References

1. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). Compilers: Principles, Techniques, and Tools (2nd Edition). Pearson.
2. Appel, A. W. (2004). Modern Compiler Implementation in C. Cambridge University Press.
3. Cooper, K., & Torczon, L. (2012). Engineering a Compiler. Morgan Kaufmann.
4. Tremblay, J. P., & Sorenson, P. G. (1985). The Theory and Practice of Compiler Writing. McGraw-Hill.