Bab 10

Symbol Table dan Scope Management

10.1 Tujuan Pembelajaran

Setelah mempelajari bab ini, mahasiswa diharapkan mampu:

1. Memahami konsep symbol table dan perannya dalam kompilator
2. Mengimplementasikan symbol table menggunakan hash table dalam C/C++
3. Memahami dan mengimplementasikan nested scopes (stack of symbol tables)
4. Melakukan name resolution dengan benar mengikuti aturan scoping
5. Menangani scope entry dan exit untuk berbagai konstruk bahasa (function, block, loop)
6. Mengidentifikasi dan menangani shadowing (pengaburan identifier)
7. Membuat visualisasi symbol table untuk debugging

10.2 Pendahuluan

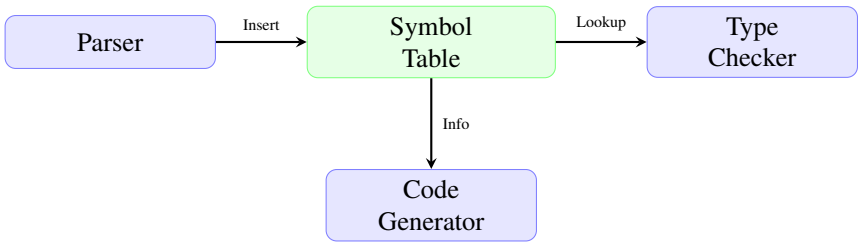
Dalam proyek compiler subset C, symbol table menyimpan informasi tentang identifier (variabel `int`/`float`) yang dideklarasikan dan dipakai dalam program. AST proyek (Bab 9, Bagian **??**) menjadi input: deklarasi ditambahkan ke symbol table saat traversal; referensi identifier diselesaikan melalui lookup. Struktur `symtab.h`/`symtab.c` di folder proyek mengimplementasikan symbol table untuk subset C.

Symbol table adalah struktur data fundamental dalam kompilator yang menyimpan informasi tentang identifier yang digunakan dalam program. Menurut sumber terbuka:

> "Symbol tables: data structures mapping names to declarations, with nested scopes. Semantic analysis includes name resolution: every use of a variable, function, type must refer to a declaration."[1]

Symbol table berfungsi sebagai *database* yang menghubungkan setiap penggunaan identifier dengan deklarasinya. Tanpa symbol table, kompilator tidak dapat memverifikasi apakah variabel yang digunakan sudah dideklarasikan, apakah tipe data sesuai, atau apakah fungsi dipanggil dengan parameter yang benar.

Gambar 10.1 menunjukkan peran symbol table dalam kompilator.



Gambar 10.1: Peran symbol table dalam kompilator

10.2.1 Informasi yang Disimpan dalam Symbol Table

Setiap entry dalam symbol table menyimpan berbagai informasi tentang identifier:

- **Nama Identifier**: String yang merepresentasikan nama variabel, fungsi, atau tipe
- **Tipe Data**: Tipe dari identifier (int, float, function, struct, dll.)
- **Scope Level**: Level nesting scope di mana identifier dideklarasikan
- **Memory Location**: Alamat atau offset memory untuk variabel (digunakan dalam code generation)
- **Line Number**: Posisi deklarasi dalam source code (untuk error reporting)
- **Attributes Tambahan**:
  - Untuk fungsi: parameter list, return type, calling convention
  - Untuk variabel: storage class (static, auto, register), initial value
  - Untuk array: dimensi dan ukuran
  - Untuk struct: field list

10.2.2 Operasi Dasar pada Symbol Table

Symbol table harus mendukung operasi-operasi berikut:

1. **Insert**: Menambahkan entry baru untuk identifier yang dideklarasikan

2. **Lookup**: Mencari identifier dalam symbol table untuk name resolution
3. **Delete**: Menghapus entry ketika scope berakhir (untuk nested scopes)
4. **Update**: Memperbarui informasi identifier (misalnya setelah type inference)

10.3 Implementasi Symbol Table dengan Hash Table

Hash table adalah struktur data yang efisien untuk implementasi symbol table karena memberikan waktu akses rata-rata O(1) untuk operasi insert dan lookup. Dalam konteks kompilator, hash table menggunakan nama identifier sebagai key.

10.3.1 Struktur Data Dasar

Berikut adalah struktur data dasar untuk symbol table menggunakan hash table dalam C++:

Listing 10.1: Struktur Data Symbol dan Scope

```
// Informasi tentang sebuah symbol
struct Symbol {
    std::string name;           // Nama identifier
    std::string type;           // Tipe data
    int scope_level;            // Level scope
    int line_number;            // Baris deklarasi
    void* memory_location;       // Alamat memory (untuk code gen)
    // Attributes tambahan sesuai kebutuhan
};

// Satu scope (satu hash table)
class Scope {
private:
    std::unordered_map<std::string, Symbol*> table;
    Scope* parent;              // Scope yang membungkus (enclosing scope
↪ )
    int level;                  // Level nesting (0 untuk global)

public:
    Scope(Scope* p = nullptr, int l = 0)
        : parent(p), level(l) {}

    bool insert(const std::string& name, Symbol* sym);
    Symbol* lookup(const std::string& name);
    Symbol* lookupLocal(const std::string& name);  // Hanya di scope ini
    Scope* getParent() { return parent; }
    int getLevel() { return level; }
};

// Symbol table utama (stack of scopes)
class SymbolTable {
private:
    Scope* current_scope;
    int next_level;
```

```
34
35 public:
36     SymbolTable();
37     ~SymbolTable();
38
39     void beginScope();           // Masuk ke scope baru
40     void endScope();             // Keluar dari scope
41     bool insert(const std::string& name, const std::string& type, int
   ↪ line);
42     Symbol* lookup(const std::string& name);
43     Symbol* lookupCurrentScope(const std::string& name);
44 };
```

10.3.2 Implementasi Operasi Dasar

Begin Scope

Ketika memasuki scope baru (misalnya saat menemukan { atau function declaration), kita membuat scope baru:

Listing 10.2: Implementasi beginScope

```
1 void SymbolTable::beginScope() {
2     Scope* new_scope = new Scope(current_scope, next_level++);
3     current_scope = new_scope;
4 }
```

End Scope

Ketika keluar dari scope (misalnya saat menemukan }), kita menghapus scope tersebut:

Listing 10.3: Implementasi endScope

```
1 void SymbolTable::endScope() {
2     if (current_scope == nullptr) return;
3
4     Scope* parent = current_scope->getParent();
5     delete current_scope;
6     current_scope = parent;
7     next_level--;
8 }
```

Insert

Menambahkan symbol ke scope saat ini. Perlu memeriksa duplikasi dalam scope yang sama:

Listing 10.4: Implementasi insert

```
1 bool SymbolTable::insert(const std::string& name,
2                          const std::string& type,
3                          int line) {
4     if (current_scope == nullptr) {
```

```
        // Error: tidak ada scope aktif
        return false;
    }

    // Cek duplikasi dalam scope saat ini
    if (current_scope->lookupLocal(name) != nullptr) {
        // Error: duplicate declaration
        return false;
    }

    Symbol* sym = new Symbol();
    sym->name = name;
    sym->type = type;
    sym->scope_level = current_scope->getLevel();
    sym->line_number = line;

    return current_scope->insert(name, sym);
}
```

Lookup

Mencari symbol mulai dari scope saat ini, kemudian naik ke parent scope jika tidak ditemukan:

Listing 10.5: Implementasi lookup dengan nested scopes

```
Symbol* SymbolTable::lookup(const std::string& name) {
    Scope* scope = current_scope;

    while (scope != nullptr) {
        Symbol* sym = scope->lookupLocal(name);
        if (sym != nullptr) {
            return sym;  // Ditemukan di scope ini
        }
        scope = scope->getParent();  // Cari di parent scope
    }

    return nullptr;  // Tidak ditemukan di semua scope
}
```

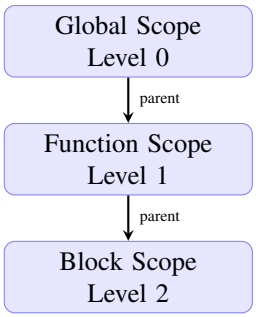
Ini mengimplementasikan aturan scoping: pencarian dimulai dari scope paling dalam (current) dan bergerak ke luar sampai menemukan deklarasi atau mencapai global scope.

Gambar 10.2 menunjukkan hierarki nested scopes.

10.4 Nested Scopes dan Scoping Rules

Nested scopes (scope bersarang) adalah fitur penting dalam bahasa pemrograman modern. Setiap konstruk bahasa tertentu menciptakan scope baru:

- **Global Scope**: Scope terluar, berisi deklarasi global



Gambar 10.2: Hierarki nested scopes

- **Function Scope**: Setiap fungsi memiliki scope sendiri
- **Block Scope**: Setiap blok `{ }` menciptakan scope baru
- **Loop Scope**: Beberapa bahasa (seperti C++ dengan for-loop) menciptakan scope untuk variabel loop
- **Class Scope**: Dalam bahasa OOP, class menciptakan scope untuk member-nya

10.4.1 Contoh Nested Scopes

Perhatikan contoh program berikut:

Listing 10.6: Contoh program dengan nested scopes

```
int x = 10;            // Global scope (level 0)

void func() {          // Function scope (level 1)
    int y = 20;        // Local variable di func
    int x = 30;        // Shadowing: x di scope ini

    {                  // Block scope (level 2)
        int z = 40;    // Local di block
        int y = 50;    // Shadowing: y di scope ini
        // Di sini: x=30 (dari func), y=50 (dari block), z=40
    }
    // Di sini: x=30 (dari func), y=20 (dari func), z tidak ada
}
// Di sini: x=10 (global), y dan z tidak ada
```

Symbol table untuk program di atas akan memiliki struktur seperti:

```
Level 0 (Global):
  x -> int (line 1)

Level 1 (func):
  y -> int (line 4)
  x -> int (line 5)  [shadows global x]
```

```
Level 2 (block):
  z -> int (line 8)
  y -> int (line 9)  [shadows func y]
```

10.4.2 Aturan Scoping

Ada dua aturan scoping utama:

1. **Static Scoping (Lexical Scoping)**:
    - Scope ditentukan oleh struktur program (lexical structure)
    - Lookup dimulai dari scope saat ini, kemudian naik ke enclosing scopes
    - Digunakan oleh sebagian besar bahasa modern (C, C++, Java, Python)
2. **Dynamic Scoping**:
    - Scope ditentukan oleh urutan eksekusi program
    - Lookup dimulai dari scope saat ini, kemudian naik ke caller's scope
    - Jarang digunakan (beberapa bahasa scripting seperti Perl dalam mode tertentu)

Buku ini fokus pada static scoping yang merupakan standar dalam bahasa pemrograman modern.

Gambar 10.3 menunjukkan perbandingan static dan dynamic scoping.

Gambar 10.3: Perbandingan static vs dynamic scoping

10.5 Name Resolution

Name resolution adalah proses menemukan deklarasi yang sesuai untuk setiap penggunaan identifier. Proses ini harus mengikuti aturan scoping bahasa.

10.5.1 Algoritma Name Resolution

Algoritma name resolution untuk static scoping:

1. Mulai dari scope saat ini (current scope)
2. Cari identifier dalam hash table scope tersebut
3. Jika ditemukan, return symbol tersebut
4. Jika tidak ditemukan, pindah ke parent scope (enclosing scope)
5. Ulangi langkah 2-4 sampai ditemukan atau mencapai global scope
6. Jika tidak ditemukan di semua scope, identifier tidak dideklarasi (error)

Implementasi algoritma ini sudah ditunjukkan dalam fungsi `lookup()` sebelumnya.

10.5.2 Shadowing (Pengaburan Identifier)

Shadowing terjadi ketika identifier dalam scope dalam memiliki nama yang sama dengan identifier di scope luar. Identifier dalam scope dalam "mengaburkan" (shadow) identifier di scope luar.

Listing 10.7: Contoh shadowing

```
int x = 10;         // Global x

void func() {
    int x = 20;     // Local x shadows global x
    // Penggunaan 'x' di sini merujuk ke local x (20)
    {
        int x = 30;  // Inner x shadows outer x
        // Penggunaan 'x' di sini merujuk ke inner x (30)
    }
    // Penggunaan 'x' di sini kembali merujuk ke local x (20)
}
```

10.5.3 Deteksi Shadowing

Beberapa kompilator memberikan peringatan ketika terjadi shadowing karena dapat menyebabkan kebingungan. Kita dapat mendeteksi shadowing saat insert:

Listing 10.8: Deteksi shadowing saat insert

```
bool SymbolTable::insert(const std::string& name,
                         const std::string& type,
                         int line) {
    // Cek duplikasi dalam scope saat ini
    if (current_scope->lookupLocal(name) != nullptr) {
        return false;  // Error: duplicate
```

```
    }

    // Cek shadowing (optional warning)
    Scope* parent = current_scope->getParent();
    while (parent != nullptr) {
        if (parent->lookupLocal(name) != nullptr) {
            // Warning: shadowing outer declaration
            std::cout << "Warning: '" << name
                      << "' shadows declaration at line "
                      << parent->lookupLocal(name)->line_number
                      << std::endl;
            break;
        }
        parent = parent->getParent();
    }

    // Insert symbol
    Symbol* sym = new Symbol();
    sym->name = name;
    sym->type = type;
    sym->scope_level = current_scope->getLevel();
    sym->line_number = line;

    return current_scope->insert(name, sym);
}
```

10.6 Handling Scope Entry dan Exit

Kompilator harus menangani masuk dan keluar scope dengan benar untuk berbagai konstruk bahasa. Ini dilakukan dengan memanggil `beginScope()` dan `endScope()` pada waktu yang tepat.

10.6.1 Function Declaration

Saat menemukan deklarasi fungsi, kita memasuki scope baru:

Listing 10.9: Handling function scope

```
// Dalam parser, saat menemukan function declaration:
void parseFunction() {
    // ... parse function signature ...

    symbolTable.beginScope();  // Masuk ke function scope

    // Parse parameter list (insert ke symbol table)
    for (auto param : parameters) {
        symbolTable.insert(param.name, param.type, param.line);
    }

    // Parse function body
    parseBlock();
```

```
14
15      symbolTable.endScope();  // Keluar dari function scope
16  }
```

10.6.2 Block Statement

Setiap blok { } menciptakan scope baru:

Listing 10.10: Handling block scope

```
1  void parseBlock() {
2      match('{');
3
4      symbolTable.beginScope();  // Masuk ke block scope
5
6      // Parse statements dalam block
7      while (currentToken != '}') {
8          parseStatement();
9      }
10
11     match('}');
12     symbolTable.endScope();  // Keluar dari block scope
13 }
```

10.6.3 Loop Statement

Beberapa bahasa (seperti C++ dengan for-loop) menciptakan scope untuk variabel loop:

Listing 10.11: Handling loop scope

```
1  void parseForLoop() {
2      match("for");
3      match('(');
4
5      symbolTable.beginScope();  // Masuk ke loop scope
6
7      // Parse loop variable declaration (jika ada)
8      if (isDeclaration()) {
9          parseDeclaration();
10     }
11
12     // Parse loop condition dan increment
13     parseExpression();  // condition
14     parseExpression();  // increment
15
16     match(')');
17
18     // Parse loop body
19     parseStatement();
20
21     symbolTable.endScope();  // Keluar dari loop scope
22 }
```

10.7 Implementasi Lengkap Symbol Table

Berikut adalah implementasi lengkap symbol table dengan semua fitur yang telah dibahas:

Listing 10.12: Implementasi lengkap SymbolTable

```
#include <string>
#include <unordered_map>
#include <iostream>
#include <vector>

struct Symbol {
    std::string name;
    std::string type;
    int scope_level;
    int line_number;
    void* memory_location;

    Symbol() : scope_level(-1), line_number(-1),
               memory_location(nullptr) {}
};

class Scope {
private:
    std::unordered_map<std::string, Symbol*> table;
    Scope* parent;
    int level;
    std::vector<std::string> declared_names;  // Untuk cleanup

public:
    Scope(Scope* p = nullptr, int l = 0)
        : parent(p), level(l) {}

    ~Scope() {
        // Cleanup semua symbols
        for (auto& pair : table) {
            delete pair.second;
        }
    }

    bool insert(const std::string& name, Symbol* sym) {
        if (table.find(name) != table.end()) {
            return false;  // Duplicate
        }
        table[name] = sym;
        declared_names.push_back(name);
        return true;
    }

    Symbol* lookupLocal(const std::string& name) {
        auto it = table.find(name);
        if (it != table.end()) {
            return it->second;
        }
        return nullptr;
```

```
    }

    Symbol* lookup(const std::string& name) {
        Symbol* sym = lookupLocal(name);
        if (sym != nullptr) {
            return sym;
        }
        if (parent != nullptr) {
            return parent->lookup(name);
        }
        return nullptr;
    }

    Scope* getParent() { return parent; }
    int getLevel() { return level; }
    const std::vector<std::string>& getDeclaredNames() const {
        return declared_names;
    }
};

class SymbolTable {
private:
    Scope* current_scope;
    int next_level;

public:
    SymbolTable() {
        current_scope = new Scope(nullptr, 0);
        next_level = 1;
    }

    ~SymbolTable() {
        // Cleanup semua scopes
        while (current_scope != nullptr) {
            Scope* parent = current_scope->getParent();
            delete current_scope;
            current_scope = parent;
        }
    }

    void beginScope() {
        Scope* new_scope = new Scope(current_scope, next_level++);
        current_scope = new_scope;
    }

    void endScope() {
        if (current_scope == nullptr ||
            current_scope->getLevel() == 0) {
            return;  // Tidak bisa keluar dari global scope
        }

        Scope* parent = current_scope->getParent();
        delete current_scope;
        current_scope = parent;
```

```
104         next_level--;
105     }
106
107     bool insert(const std::string& name,
108                 const std::string& type,
109                 int line) {
110         if (current_scope == nullptr) {
111             return false;
112         }
113
114         // Cek duplikasi
115         if (current_scope->lookupLocal(name) != nullptr) {
116             std::cerr << "Error: Duplicate declaration of '"
117                       << name << "' at line " << line << std::endl;
118             return false;
119         }
120
121         // Cek shadowing (optional warning)
122         Scope* parent = current_scope->getParent();
123         while (parent != nullptr) {
124             Symbol* shadowed = parent->lookupLocal(name);
125             if (shadowed != nullptr) {
126                 std::cout << "Warning: '" << name
127                           << "' at line " << line
128                           << " shadows declaration at line "
129                           << shadowed->line_number << std::endl;
130                 break;
131             }
132             parent = parent->getParent();
133         }
134
135         // Insert symbol
136         Symbol* sym = new Symbol();
137         sym->name = name;
138         sym->type = type;
139         sym->scope_level = current_scope->getLevel();
140         sym->line_number = line;
141
142         return current_scope->insert(name, sym);
143     }
144
145     Symbol* lookup(const std::string& name) {
146         if (current_scope == nullptr) {
147             return nullptr;
148         }
149         return current_scope->lookup(name);
150     }
151
152     Symbol* lookupCurrentScope(const std::string& name) {
153         if (current_scope == nullptr) {
154             return nullptr;
155         }
156         return current_scope->lookupLocal(name);
157     }
```

```
158
159     int getCurrentLevel() {
160         return current_scope ? current_scope->getLevel() : -1;
161     }
162
163     Scope* getCurrentScope() {
164         return current_scope;
165     }
166 };
```

10.8 Visualisasi Symbol Table

Visualisasi symbol table sangat berguna untuk debugging dan pembelajaran. Berikut adalah fungsi untuk mencetak isi symbol table:

Listing 10.13: Fungsi visualisasi symbol table

```
1  // Fungsi helper untuk visualisasi (perlu menambahkan getCurrentScope()
2  // ke class SymbolTable atau menggunakan pendekatan lain)
3  void printSymbolTableHelper(Scope* scope) {
4      if (scope == nullptr) return;
5
6      // Rekursif ke parent dulu (agar print dari global ke current)
7      printSymbolTableHelper(scope->getParent());
8
9      // Print scope ini
10     std::cout << "\nLevel " << scope->getLevel() << ":" << std::endl;
11     std::cout << "  Symbols:" << std::endl;
12
13     // Print semua symbols di scope ini
14     for (const auto& name : scope->getDeclaredNames()) {
15         Symbol* sym = scope->lookupLocal(name);
16         if (sym != nullptr) {
17             std::cout << "    " << sym->name
18                       << " : " << sym->type
19                       << " (line " << sym->line_number << ")"
20                       << std::endl;
21         }
22     }
23 }
24
25 void printSymbolTable(SymbolTable& st) {
26     std::cout << "\n=== Symbol Table ===" << std::endl;
27
28     // Asumsikan SymbolTable memiliki method getCurrentScope()
29     // Atau kita bisa mengakses current_scope jika public/protected
30     // Untuk contoh ini, kita asumsikan ada method helper
31     Scope* current = st.getCurrentScope();  // Perlu ditambahkan ke class
32
33     printSymbolTableHelper(current);
34
35     std::cout << "====================\n" << std::endl;
36 }
```

Contoh output visualisasi:

```
=== Symbol Table ===

Level 0:
  Symbols:
    x : int (line 1)

Level 1:
  Symbols:
    y : int (line 4)
    x : int (line 5)

Level 2:
  Symbols:
    z : int (line 8)
    y : int (line 9)
==================
```

10.9 Integrasi dengan Semantic Analyzer

Symbol table diintegrasikan dengan semantic analyzer untuk melakukan berbagai pemeriksaan:

1. **Declaration Check**: Memastikan setiap identifier dideklarasi sebelum digunakan
2. **Type Checking**: Memverifikasi tipe data dalam operasi dan assignment
3. **Scope Resolution**: Menyelesaikan referensi identifier ke deklarasi yang benar
4. **Duplicate Detection**: Mendeteksi deklarasi ganda dalam scope yang sama

Contoh integrasi dengan semantic analyzer:

Listing 10.14: Contoh penggunaan symbol table dalam semantic analysis

```
void semanticAnalyzeIdentifier(ASTNode* node) {
    std::string name = node->getName();

    // Lookup identifier
    Symbol* sym = symbolTable.lookup(name);

    if (sym == nullptr) {
        // Error: identifier tidak dideklarasi
        error("Undeclared identifier: " + name,
              node->getLineNumber());
        return;
    }

    // Annotate AST node dengan symbol info
```

```
    node->setSymbol(sym);
    node->setType(sym->type);
}

void semanticAnalyzeAssignment(ASTNode* node) {
    ASTNode* lhs = node->getLeft();
    ASTNode* rhs = node->getRight();

    // Analyze kedua sisi
    semanticAnalyzeExpression(lhs);
    semanticAnalyzeExpression(rhs);

    // Type checking
    if (lhs->getType() != rhs->getType()) {
        error("Type mismatch in assignment",
              node->getLineNumber());
    }
}
```

10.10 Kesimpulan

Dalam bab ini, kita telah mempelajari:

1. Symbol table adalah struktur data penting yang memetakan identifier ke informasi deklarasinya
2. Hash table adalah implementasi efisien untuk symbol table dengan waktu akses O(1) rata-rata
3. Nested scopes diimplementasikan menggunakan stack of hash tables, di mana setiap scope memiliki parent pointer
4. Name resolution mengikuti aturan static scoping: pencarian dimulai dari scope saat ini dan naik ke enclosing scopes
5. Shadowing terjadi ketika identifier dalam scope dalam memiliki nama yang sama dengan identifier di scope luar
6. Scope entry/exit harus ditangani dengan benar untuk berbagai konstruk bahasa (function, block, loop)
7. Visualisasi symbol table membantu dalam debugging dan pembelajaran

Pemahaman tentang symbol table dan scope management adalah dasar penting untuk implementasi semantic analysis yang akan dibahas dalam bab selanjutnya. Symbol table proyek subset C (`symtab.h`/`symtab.c`) dipakai oleh type checking (Bab 11) dan code generation (Bab 14) dalam pipeline compiler proyek.

10.11 Referensi dan Bahan Bacaan Lanjutan

Untuk memperdalam pemahaman tentang symbol table dan scope management, mahasiswa disarankan membaca:

- **Dragon Book**: Aho, Lam, Sethi, & Ullman (2006). *Compilers: Principles, Techniques, and Tools* [2] - Bab 2.7: Symbol Tables
- **Engineering a Compiler**: Cooper & Torczon (2011) [3] - Bab 5: Scoping
- **Nguyen Thanh Vu - Compiler Class Notes**: Semantic Analysis dan Symbol Tables [1]
- **University of Texas at Arlington**: Symbol Table Implementation [1]
- **University of Wisconsin**: Symbol Tables and Scoping [2]

[1] https://lambda.uta.edu/cse5317/spring18/long/index.html
[2] https://pages.cs.wisc.edu/~fischer/cs536.s08/course.hold/html/NOTES/6.SYMBOL-TABLES.html

Daftar Pustaka

[1] Nguyen Thanh Vu. *Compiler Class Notes: Semantic Analysis*. Class notes. 2024. URL: https://nguyenthanhvuh.github.io/class-compilers/notes/sem.html.

[2] Alfred V. Aho **andothers**. *Compilers: Principles, Techniques, and Tools*. 2nd. Dragon Book. Pearson Education, 2006.

[3] Keith D. Cooper **and** Linda Torczon. *Engineering a Compiler*. 2nd. Morgan Kaufmann, 2011.