Bab 1

Intermediate Code Generation

1.1 Tujuan Pembelajaran

Setelah mempelajari bab ini, mahasiswa diharapkan mampu:

1. Memahami konsep intermediate code generation dan perannya dalam kompilator
2. Menjelaskan berbagai format intermediate representation (IR)
3. Mengimplementasikan generator three-address code (TAC) dari AST
4. Menangani generasi kode untuk berbagai jenis statement (assignment, if, loop)
5. Menerapkan optimasi dasar seperti common subexpression elimination
6. Memahami representasi quadruples dan implementasinya

1.2 Pendahuluan

Setelah fase semantic analysis menghasilkan annotated AST dengan informasi tipe dan symbol table yang lengkap, kompilator perlu menghasilkan representasi intermediate yang lebih dekat ke machine code namun tetap machine-independent. Fase ini disebut **Intermediate Code Generation**.

Menurut sumber dari OpenGenus:

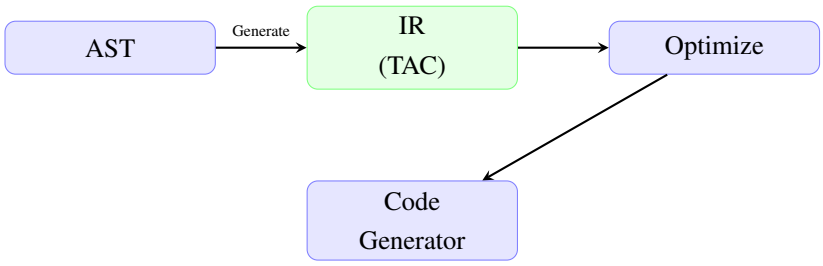
> "Intermediate code generation transforms AST to IR (three-address code, bytecode, etc.). Design and generate intermediate code representations (e.g., three-address code, DAGs)."**?**

Intermediate representation (IR) memiliki karakteristik penting:

- **Machine-Independent**: IR tidak bergantung pada arsitektur target tertentu, memungkinkan portabilitas

- **Simpler than AST**: Lebih sederhana dari AST, memudahkan optimasi dan code generation
- **Closer to Machine Code**: Lebih dekat ke machine code dibanding AST, memudahkan translasi ke target code
- **Optimization-Friendly**: Struktur yang memudahkan berbagai teknik optimasi

Gambar 1.1 menunjukkan posisi IR dalam pipeline kompilator.



Gambar 1.1: Posisi IR dalam pipeline kompilator

1.2.1 Alasan Menggunakan Intermediate Code

Penggunaan intermediate code memberikan beberapa keuntungan:

1. **Portabilitas**: Satu IR dapat digunakan untuk berbagai target platform. Kompilator hanya perlu mengubah back-end untuk target baru (tanpa mengubah front-end).
2. **Optimasi yang Lebih Baik**: IR yang lebih sederhana memudahkan analisis dan optimasi. Optimasi dapat dilakukan pada IR sebelum code generation.
3. **Pemisahan Front-end dan Back-end**: Front-end menghasilkan IR, back-end mengkonsumsi IR. Perubahan pada satu sisi tidak mempengaruhi sisi lain.
4. **Retargeting**: Untuk menambahkan dukungan target baru, cukup menambahkan code generator untuk IR tersebut.

1.3 Format Intermediate Representation

Terdapat berbagai format IR yang umum digunakan dalam kompilator modern:

1.3.1 Three-Address Code (TAC)

Three-address code adalah format IR di mana setiap instruksi memiliki paling banyak tiga operand (dua sumber dan satu tujuan). Format ini sangat populer karena:

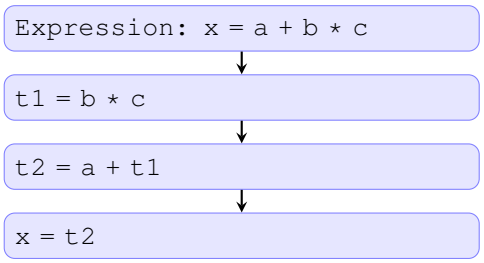
- Sederhana dan mudah dipahami
- Mirip dengan assembly code
- Memudahkan optimasi
- Mudah di-generate dari AST

Contoh three-address code untuk ekspresi `x = a + b * c`:

```
t1 = b * c
t2 = a + t1
x = t2
```

Setiap baris adalah satu instruksi dengan format: `result = operand1 operator operand2`

Gambar 1.2 menunjukkan contoh three-address code untuk ekspresi kompleks.



Gambar 1.2: Contoh three-address code

1.3.2 Quadruples

Quadruples adalah representasi struktural dari three-address code. Setiap instruksi direpresentasikan sebagai record dengan empat field:

- **op**: Operator (+, -, *, /, =, jmp, jmpf, dll.)
- **arg1**: Operand pertama
- **arg2**: Operand kedua (kosong untuk unary operations)
- **result**: Hasil operasi (temporary atau variabel)

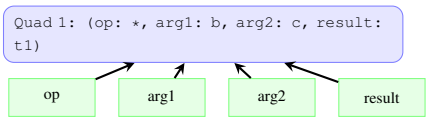
Contoh quadruple untuk `x = a + b * c`:

```
Quad 1: (op: *, arg1: b, arg2: c, result: t1)
Quad 2: (op: +, arg1: a, arg2: t1, result: t2)
Quad 3: (op: =, arg1: t2, arg2: _, result: x)
```

Keuntungan quadruples:

- Mudah untuk di-reorder (optimasi)
- Mudah untuk di-optimize (common subexpression elimination)
- Struktur data yang jelas untuk manipulasi

Gambar 1.3 menunjukkan struktur quadruple.



Gambar 1.3: Struktur quadruple

1.3.3 Format IR Lainnya

Selain TAC dan quadruples, terdapat format IR lainnya:

- **Static Single Assignment (SSA)**: Setiap variabel hanya di-assign sekali, memudahkan optimasi data-flow
- **Bytecode**: Untuk bahasa yang diinterpretasi (Java bytecode, Python bytecode)
- **DAG (Directed Acyclic Graph)**: Representasi graf untuk ekspresi, memudahkan common subexpression elimination
- **LLVM IR**: Format IR modern yang digunakan oleh LLVM compiler infrastructure

1.4 Implementasi TAC Generator dari AST

Implementasi generator TAC dari AST dilakukan dengan melakukan traversal pada AST secara recursive. Untuk setiap node AST, generator menghasilkan instruksi TAC yang sesuai.

1.4.1 Struktur Data untuk TAC

Sebelum mengimplementasikan generator, kita perlu mendefinisikan struktur data untuk menyimpan TAC:

Listing 1.1: Struktur data untuk Quadruple

```
struct Quad {
    std::string op;      // Operator
    std::string arg1;    // Operand pertama
    std::string arg2;    // Operand kedua (kosong jika unary)
```

```
    std::string result;  // Hasil (temporary atau variabel)

    Quad(const std::string& op, const std::string& arg1,
         const std::string& arg2, const std::string& result)
        : op(op), arg1(arg1), arg2(arg2), result(result) {}
};

class QuadList {
private:
    std::vector<Quad> quads;
    int tempCounter;
    int labelCounter;

public:
    QuadList() : tempCounter(0), labelCounter(0) {}

    void emit(const Quad& quad) {
        quads.push_back(quad);
    }

    std::string newTemp() {
        return "t" + std::to_string(tempCounter++);
    }

    std::string newLabel() {
        return "L" + std::to_string(labelCounter++);
    }

    void print() const {
        for (size_t i = 0; i < quads.size(); i++) {
            std::cout << i << ": (" << quads[i].op << ", "
                      << quads[i].arg1 << ", " << quads[i].arg2
                      << ", " << quads[i].result << ")\n";
        }
    }
};
```

1.4.2 Generator untuk Ekspresi

Generator untuk ekspresi aritmatika bekerja secara recursive:

Listing 1.2: Generator TAC untuk ekspresi

```
class ASTNode {
public:
    virtual std::string genCode(QuadList& quads, SymbolTable& symtab) =
↪ 0;
};

class ASTConst : public ASTNode {
    int value;
public:
    std::string genCode(QuadList& quads, SymbolTable& symtab) override {
        std::string temp = quads.newTemp();
```

```
        quads.emit(Quad("load_const", std::to_string(value), "", temp));
        return temp;
    }
};

class ASTVar : public ASTNode {
    std::string name;
public:
    std::string genCode(QuadList& quads, SymbolTable& symtab) override {
        return name;  // Langsung return nama variabel
    }
};

class ASTBinaryOp : public ASTNode {
    std::string op;
    ASTNode* left;
    ASTNode* right;
public:
    std::string genCode(QuadList& quads, SymbolTable& symtab) override {
        // Generate code untuk left dan right subtree
        std::string leftTemp = left->genCode(quads, symtab);
        std::string rightTemp = right->genCode(quads, symtab);

        // Generate temporary untuk hasil
        std::string resultTemp = quads.newTemp();

        // Emit quadruple
        quads.emit(Quad(op, leftTemp, rightTemp, resultTemp));

        return resultTemp;
    }
};
```

1.4.3 Generator untuk Assignment

Assignment statement menghasilkan instruksi assignment:

Listing 1.3: Generator TAC untuk assignment

```
class ASTAssign : public ASTNode {
    std::string varName;
    ASTNode* expr;
public:
    std::string genCode(QuadList& quads, SymbolTable& symtab) override {
        // Generate code untuk ekspresi
        std::string exprTemp = expr->genCode(quads, symtab);

        // Emit assignment
        quads.emit(Quad("=", exprTemp, "", varName));

        return varName;
    }
};
```

1.5 Handling Control Flow Statements

Generasi kode untuk control flow statements (if, while, for) memerlukan label dan jump instructions.

1.5.1 If-Then-Else Statement

Untuk if statement, kita perlu:

- Label untuk else branch (jika ada)
- Label untuk end of if statement
- Conditional jump berdasarkan kondisi
- Unconditional jump untuk skip else branch

Listing 1.4: Generator TAC untuk if statement

```
class ASTIf : public ASTNode {
    ASTNode* condition;
    ASTNode* thenBranch;
    ASTNode* elseBranch;  // bisa null
public:
    std::string genCode(QuadList& quads, SymbolTable& symtab) override {
        // Generate code untuk kondisi
        std::string condTemp = condition->genCode(quads, symtab);

        std::string elseLabel = quads.newLabel();
        std::string endLabel = quads.newLabel();

        // Jump to else jika kondisi false
        quads.emit(Quad("jmpf", condTemp, "", elseLabel));

        // Generate code untuk then branch
        thenBranch->genCode(quads, symtab);

        // Jump to end (skip else)
        quads.emit(Quad("jmp", "", "", endLabel));

        // Else label
        quads.emit(Quad("label", "", "", elseLabel));

        // Generate code untuk else branch (jika ada)
        if (elseBranch != nullptr) {
            elseBranch->genCode(quads, symtab);
        }

        // End label
        quads.emit(Quad("label", "", "", endLabel));

        return "";  // if statement tidak menghasilkan nilai
```

```
    }
};
```

Contoh output untuk `if (x > 0) y = 1; else y = 0;`:

```
t1 = x > 0
jmpf t1, L0
t2 = 1
y = t2
jmp L1
label L0
t3 = 0
y = t3
label L1
```

1.5.2 While Loop

While loop memerlukan:

- Label untuk start of loop
- Label untuk end of loop
- Conditional jump untuk exit loop
- Unconditional jump untuk kembali ke start

Listing 1.5: Generator TAC untuk while loop

```
class ASTWhile : public ASTNode {
    ASTNode* condition;
    ASTNode* body;
public:
    std::string genCode(QuadList& quads, SymbolTable& symtab) override {
        std::string startLabel = quads.newLabel();
        std::string endLabel = quads.newLabel();

        // Start label
        quads.emit(Quad("label", "", "", startLabel));

        // Generate code untuk kondisi
        std::string condTemp = condition->genCode(quads, symtab);

        // Jump to end jika kondisi false
        quads.emit(Quad("jmpf", condTemp, "", endLabel));

        // Generate code untuk body
        body->genCode(quads, symtab);

        // Jump back to start
        quads.emit(Quad("jmp", "", "", startLabel));

```

```
24          // End label
25          quads.emit(Quad("label", "", "", endLabel));
26
27          return "";
28      }
29  };
```

Contoh output untuk `while (i < 10) { i = i + 1; }`:

```
label L0
t1 = i < 10
jmpf t1, L1
t2 = i + 1
i = t2
jmp L0
label L1
```

1.5.3 For Loop

For loop dapat di-translate menjadi while loop atau di-generate langsung. Implementasi sebagai while loop:

```
for (init; condition; update) {
    body
}
```

Menjadi:

```
init
label L0
jmpf condition, L1
body
update
jmp L0
label L1
```

1.6 Handling Function Calls

Function calls memerlukan:

- Evaluasi arguments
- Parameter passing (param instructions)
- Call instruction
- Return value handling

Listing 1.6: Generator TAC untuk function call

```
class ASTFunctionCall : public ASTNode {
    std::string funcName;
    std::vector<ASTNode*> arguments;
public:
    std::string genCode(QuadList& quads, SymbolTable& symtab) override {
        // Generate code untuk setiap argument
        for (auto arg : arguments) {
            std::string argTemp = arg->genCode(quads, symtab);
            quads.emit(Quad("param", argTemp, "", ""));
        }

        // Generate temporary untuk return value
        std::string resultTemp = quads.newTemp();

        // Emit call instruction
        quads.emit(Quad("call", funcName,
                        std::to_string(arguments.size()),
                        resultTemp));

        return resultTemp;
    }
};
```

1.7 Optimasi Dasar: Common Subexpression Elimination

Common subexpression elimination (CSE) adalah optimasi yang mengidentifikasi dan menghilangkan komputasi ekspresi yang sama yang dilakukan berulang kali.

1.7.1 Konsep Common Subexpression Elimination

Contoh ekspresi yang dapat dioptimasi:

```
x = a + b * c
y = a + b * c  // b * c dihitung dua kali
```

Setelah optimasi:

```
t1 = b * c
x = a + t1
y = a + t1  // Menggunakan t1 yang sudah dihitung
```

1.7.2 Implementasi CSE Sederhana

CSE dapat diimplementasikan dengan:

1. Mencari ekspresi yang sama dalam basic block
2. Mengganti ekspresi kedua dan seterusnya dengan temporary yang sudah dihitung

3. Menghapus komputasi yang redundant

Algoritma sederhana untuk CSE:

Listing 1.7: Algoritma CSE sederhana

```
void eliminateCommonSubexpressions(QuadList& quads) {
    // Map untuk menyimpan ekspresi yang sudah dihitung
    std::map<std::pair<std::string, std::pair<std::string, std::string>>,
             std::string> exprMap;

    for (auto& quad : quads.quads) {
        // Skip non-computational operations
        if (quad.op == "=" || quad.op == "jmp" ||
            quad.op == "jmpf" || quad.op == "label") {
            continue;
        }

        // Buat key dari operator dan operands
        auto key = std::make_pair(quad.op,
                                  std::make_pair(quad.arg1, quad.arg2));

        // Cek apakah ekspresi ini sudah dihitung sebelumnya
        if (exprMap.find(key) != exprMap.end()) {
            // Ganti result dengan temporary yang sudah ada
            std::string existingTemp = exprMap[key];
            // Update semua penggunaan result dengan existingTemp
            replaceUses(quads, quad.result, existingTemp);
            // Hapus quad ini (atau mark sebagai redundant)
            markAsRedundant(quad);
        } else {
            // Simpan ekspresi ini
            exprMap[key] = quad.result;
        }
    }
}
```

1.7.3 Contoh Optimasi CSE

Sebelum optimasi:

```
t1 = b * c
t2 = a + t1
x = t2
t3 = b * c    // Common subexpression!
t4 = a + t3   // Common subexpression!
y = t4
```

Setelah optimasi:

```
t1 = b * c
t2 = a + t1
x = t2
y = t2        // Menggunakan t2 yang sudah dihitung
```

1.8 Integrasi dengan Fase Sebelumnya

Intermediate code generation terintegrasi dengan fase-fase sebelumnya:

1.8.1 Input dari Semantic Analysis

Generator IR menerima:

- **Annotated AST**: AST dengan informasi tipe pada setiap node
- **Symbol Table**: Informasi tentang variabel, fungsi, dan tipe
- **Type Information**: Informasi tipe untuk setiap ekspresi

1.8.2 Output untuk Code Generation

Generator IR menghasilkan:

- **Quadruple List**: Daftar instruksi IR
- **Label Information**: Informasi tentang label yang digunakan
- **Temporary Variables**: Daftar temporary yang digunakan

1.9 Contoh Lengkap: Ekspresi Kompleks

Mari kita lihat contoh lengkap generasi TAC untuk ekspresi kompleks:

1.9.1 Source Code

```
int a, b, c, x, y;
x = a + b * c;
y = (a + b) * c;
if (x > y) {
    x = x + 1;
} else {
    y = y + 1;
}
```

1.9.2 Generated TAC

```
// Assignment: x = a + b * c
t1 = b * c
t2 = a + t1
x = t2
```

```
// Assignment: y = (a + b) * c
t3 = a + b
t4 = t3 * c
y = t4

// If statement: if (x > y) ...
t5 = x > y
jmpf t5, L0
t6 = x + 1
x = t6
jmp L1
label L0
t7 = y + 1
y = t7
label L1
```

1.10 Kesimpulan

Dalam bab ini, kita telah mempelajari:

1. Intermediate code generation adalah fase yang mengubah AST menjadi IR yang lebih dekat ke machine code
2. Three-address code (TAC) dan quadruples adalah format IR yang populer
3. Generator TAC bekerja dengan recursive traversal pada AST
4. Control flow statements memerlukan label dan jump instructions
5. Common subexpression elimination adalah optimasi dasar yang penting
6. IR memungkinkan portabilitas dan optimasi yang lebih baik

Pemahaman tentang intermediate code generation menjadi dasar penting untuk fase code generation dan optimasi yang akan dipelajari dalam bab-bab selanjutnya.

1.11 Referensi dan Bahan Bacaan Lanjutan

Untuk memperdalam pemahaman tentang intermediate code generation, mahasiswa disarankan membaca:

- **Dragon Book**: Aho, Lam, Sethi, & Ullman (2006). *Compilers: Principles, Techniques, and Tools* **?** - Bab 6: Intermediate-Code Generation
- **Engineering a Compiler**: Cooper & Torczon (2011) **?** - Bab 6: The Procedure Abstraction dan Bab 7: Code Shape

- **SDSU CS 524**: Intermediate Code Generation [1]
- **GeeksforGeeks**: Three Address Code [2]
- **Linköping University**: Lab 6 - Intermediate Code Generation [3]
- **Shasank's Engineering Notes**: Module 7 - Intermediate Code Generation [4]
- **Wikipedia - Intermediate Representation**: [5]
- **LLVM Language Reference**: [6] - Untuk mempelajari format IR modern

[1] https://stewart.sdsu.edu/cs524/spr08/lects/ch6_IntermediateCodeGen.html

[2] https://www.geeksforgeeks.org/three-address-code-compiler/

[3] https://www.ida.liu.se/~TDDB44/laboratories/instructions/lab6.html

[4] https://shasankp000.github.io/CSE-Engineering-Notes/Compiler_Design/Module-7----Intermediate-Code-Generation

[5] https://en.wikipedia.org/wiki/Intermediate_representation

[6] https://llvm.org/docs/LangRef.html