Bab 19

Soal Ujian Tengah Semester dan Ujian Akhir Semester

19.1 Soal Ujian Tengah Semester (UTS)

19.1.1 Petunjuk

- Waktu ujian: 120 menit
- Materi: Bab 1 sampai Bab 7
- Jawablah semua soal dengan jelas dan lengkap
- Gunakan diagram atau ilustrasi jika diperlukan

19.1.2 Soal UTS

1. **[Bab 1 - 15 poin]** Jelaskan secara detail perbedaan antara kompilator dan interpreter. Berikan contoh minimal 2 bahasa pemrograman untuk masing-masing pendekatan dan jelaskan mengapa bahasa tersebut menggunakan pendekatan tersebut. Selain itu, jelaskan juga pendekatan hybrid yang digunakan oleh beberapa bahasa modern.

2. **[Bab 1 - 15 poin]** Buatlah diagram lengkap yang menunjukkan alur kerja kompilator dari source code C sampai menjadi executable binary. Sertakan semua fase kompilasi yang telah dipelajari beserta output dari setiap fase. Jelaskan juga peran linker dan assembler dalam proses kompilasi.

3. **[Bab 2 - 20 poin]**

   (a) Buatlah regular expression untuk mengenali:

   - Identifier dalam bahasa C (dimulai dengan huruf atau underscore, diikuti huruf, angka, atau underscore)
   - Floating point number (format: `123.456` atau `1.23e-4`)
   - C-style multi-line comment (`/* ... */`)

(b) Konstruksi NFA untuk regular expression `(a|b)*abb` menggunakan algoritma Thompson. Gambarkan state diagram lengkap dengan semua transisi.

(c) Konversi NFA dari poin (b) menjadi DFA menggunakan subset construction. Tunjukkan proses konversi secara detail dan gambarkan DFA yang dihasilkan.

4. **[Bab 3 - 15 poin]** Implementasikan lexer sederhana dalam C++ yang dapat mengenali token-token berikut:

   - Keywords: `if`, `else`, `while`, `int`, `float`
   - Identifier: `[a-zA-Z_][a-zA-Z0-9_]*`
   - Integer literal: `[0-9]+`
   - Float literal: `[0-9]+\.[0-9]+`
   - Operators: `+`, `-`, `*`, `/`, `=`, `==`, `!=`
   - Punctuation: `;`, `,`, `(`, `)`, `{`, `}`

   Sertakan juga error handling untuk karakter yang tidak valid dan tracking posisi (line dan column number).

5. **[Bab 4 - 15 poin]**

   (a) Buatlah specification file Flex untuk mengenali semua token yang sama dengan soal nomor 3. Bandingkan kompleksitas implementasi antara hand-written lexer dan Flex-generated lexer.

   (b) Jelaskan kapan sebaiknya menggunakan hand-written lexer dan kapan menggunakan lexer generator seperti Flex atau re2c. Berikan contoh kasus konkret untuk masing-masing pendekatan.

19.2 Jawaban Ujian Tengah Semester (UTS)

19.2.1 Jawaban Soal UTS

1. **[Bab 1 - 15 poin] Perbedaan antara kompilator dan interpreter:**

   Kompilator:

   - Menerjemahkan seluruh program menjadi kode target sebelum eksekusi.
   - Hasil translasi berupa object code atau executable binary.
   - Proses kompilasi dilakukan sekali, hasilnya dapat dijalankan berulang kali tanpa kompilasi ulang.
   - Contoh bahasa: C, C++, Rust, Fortran.

- Alasan penggunaan: performa eksekusi tinggi karena kode telah diterjemahkan ke bahasa mesin dan dapat dioptimasi.

Interpreter:

- Mengeksekusi program melalui interpretasi atau bytecode execution.
- Source code biasanya diterjemahkan ke intermediate representation (bytecode) sebelum dieksekusi.
- Eksekusi dilakukan melalui virtual machine atau interpreter runtime.
- Contoh bahasa: Python, JavaScript, Ruby, PHP.
- Alasan penggunaan: fleksibilitas tinggi dan kemudahan pengembangan.

Pendekatan Hybrid:

- **Java**: dikompilasi menjadi bytecode, kemudian dieksekusi oleh JVM dengan dukungan JIT compilation.
- **C#**: dikompilasi menjadi Intermediate Language (IL), kemudian dieksekusi oleh CLR dengan JIT.
- **Python**: dikompilasi menjadi bytecode (.pyc) lalu diinterpretasi oleh Python Virtual Machine; beberapa implementasi mendukung JIT.

2. **[Bab 1 - 15 poin] Tahapan proses kompilasi:**

   (a) **Preprocessing**: menghasilkan expanded source code.

   (b) **Lexical Analysis**: menghasilkan token stream.

   (c) **Syntax Analysis**: membangun parse tree atau AST.

   (d) **Semantic Analysis**: menghasilkan annotated AST.

   (e) **Intermediate Code Generation**: menghasilkan intermediate representation (IR).

   (f) **Code Optimization**: optimasi IR.

   (g) **Code Generation**: menghasilkan assembly code.

   (h) **Assembling**: menghasilkan object file.

   (i) **Linking**: menghasilkan executable binary.

3. **[Bab 2 - 20 poin]**

   (a) **Regular expression:**

   - Identifier C: `[a-zA-Z_][a-zA-Z0-9_]*`

- Floating point number:

$$[0-9]+\.[0-9]+([eE][+-]?[0-9]+)?$$

- C-style multi-line comment:

$$/([^*]|[^*/])*/$$

(b) **Konstruksi NFA Thompson untuk $(a|b)^*abb$:**

Struktur Thompson:

- NFA untuk a dan b dibangun secara terpisah.
- Operator union $(a|b)$ dibangun dengan ϵ-transition.
- Operator Kleene star $(a|b)^*$ ditambahkan dengan state awal dan akhir baru.
- Konkatenasi dengan simbol a, b, dan b dilakukan secara berurutan.

(c) **Subset construction (NFA $\rightarrow$ DFA):**

i. Tentukan ϵ-closure dari state awal NFA sebagai state awal DFA.

ii. Untuk setiap state DFA dan setiap simbol input, hitung transisi dan ϵ-closure.

iii. State DFA yang mengandung accept state NFA menjadi accept state DFA.

4. **[Bab 3 - 15 poin] Implementasi lexer C++:**

(isi tetap, tidak ada kesalahan konsep utama)

5. **[Bab 4 - 15 poin]**

Perbaikan Flex specification:

```
%%
"if"|"else"|"while"|"int"|"float"     { return KEYWORD; }
[a-zA-Z_][a-zA-Z0-9_]*                { return IDENTIFIER; }
[0-9]+                                { return INTEGER; }
[0-9]+\.[0-9]+([eE][+-]?[0-9]+)?      { return FLOAT; }
"=="|"!="|"<="|">="                   { return OPERATOR; }
"+"|"-"|"*"|"/"|"="                   { return OPERATOR; }
";"|","|"("|")"|"{"|"}"               { return PUNCTUATION; }
[ \t\n]+                              { /* skip whitespace */ }
.                                     { /* lexical error */ }
%%
```

6. **[Bab 5 - 20 poin]**

Grammar bebas left recursion:

```
E  -> T E'
E' -> + T E' | - T E' | ε
```

```
T  -> F T'
T' -> * F T' | / F T' | ε
F  -> number | ( E )
```

Leftmost derivation untuk $2 + 3 * 4$**:**

```
E
=> T E'
=> F T' E'
=> 2 E'
=> 2 + T E'
=> 2 + F T' E'
=> 2 + 3 * F T' E'
=> 2 + 3 * 4
```

Ambiguity grammar:

Grammar

$$E \rightarrow E + E \mid E * E \mid (E) \mid number$$

adalah ambiguous karena terdapat lebih dari satu parse tree untuk string $2 + 3 * 4$.

7. **[Bab 6 - 20 poin]**

   Grammar setelah eliminasi left recursion:

```
E  -> T E'
E' -> + T E' | - T E' | ε
T  -> F T'
T' -> * F T' | / F T' | ε
F  -> number | ( E )
```

8. **[Bab 7 - 20 poin]**

   Perbaikan shift-reduce parsing:

   Grammar:

```
E -> E + T | T
T -> T * F | F
F -> id
```

   Urutan reduce yang benar:

   - F $\rightarrow$ id
   - T $\rightarrow$ F
   - F $\rightarrow$ id
   - T $\rightarrow$ T * F
   - E $\rightarrow$ T

- $E \rightarrow E + T$

Contoh grammar LALR(1) tetapi bukan SLR(1):

```
S -> L = R | R
L -> * R | id
R -> L
```

9. **[Integratif - 20 poin]**

   (arsitektur kompilator tetap, sudah benar)

19.3 Soal Ujian Akhir Semester (UAS)

19.3.1 Petunjuk

- Waktu ujian: 150 menit
- Materi: Bab 8 sampai Bab 15
- Jawablah semua soal dengan jelas dan lengkap
- Gunakan diagram, pseudocode, atau ilustrasi jika diperlukan
- Soal bersifat integratif, menghubungkan konsep dari berbagai bab

19.3.2 Soal UAS

1. **[Bab 8 - 20 poin]**

   (a) Buatlah spesifikasi grammar untuk Bison/Yacc yang dapat mem-parsing ekspresi aritmatika dengan operator `+`, `-`, `*`, `/`, dan tanda kurung. Grammar harus menghormati precedence dan associativity yang benar. Sertakan semantic actions untuk membangun AST.

   (b) Jelaskan bagaimana Bison menangani shift-reduce conflict dan reduce-reduce conflict. Berikan contoh grammar yang menyebabkan masing-masing conflict dan jelaskan cara mengatasinya.

   (c) Implementasikan kalkulator sederhana menggunakan Flex dan Bison yang dapat mengevaluasi ekspresi aritmatika. Sertakan error handling untuk kesalahan sintaks.

2. **[Bab 9 - 20 poin]**

   (a) Rancang struktur data AST untuk bahasa sederhana yang mendukung:

   - Deklarasi variabel: `int x;`

- Assignment: `x = 5;`
- Ekspresi aritmatika: `a + b * c`
- Pernyataan if-else: `if (x > 0) { ... } else { ... }`
- Perulangan while: `while (x < 10) { ... }`

Gunakan class hierarchy dalam C++ atau struct dengan variant types.

(b) Implementasikan fungsi untuk melakukan post-order traversal pada AST. Jelaskan mengapa traversal ini penting untuk code generation.

(c) Buat fungsi visualisasi sederhana untuk mencetak AST dalam format tree. Berikan contoh output untuk ekspresi `(a + b) * c - d`.

3. **[Bab 10 - 20 poin]**

(a) Rancang implementasi symbol table yang efisien untuk bahasa dengan nested scope. Struktur data harus mendukung:

- Insert
- Lookup
- Enter scope
- Exit scope

Jelaskan pilihan struktur data dan kompleksitas operasinya.

(b) Implementasikan name resolution untuk program berikut dan jelaskan hasil resolusi setiap identifier:

```
int x = 1;
{
    int y = 2;
    {
        int x = 3;
        y = x + y;
    }
    x = y;
}
```

(c) Jelaskan perbedaan static scoping dan dynamic scoping beserta contoh program.

4. **[Bab 11 - 20 poin]**

(a) Implementasikan type checker sederhana.

(b) Jelaskan konsep type inference dan bandingkan dengan explicit type checking.

(c) Rancang algoritma semantic analysis.

5. **[Bab 12 - 25 poin]**

(a) Implementasikan generator three-address code (TAC) dari AST.

(b) Konversi ekspresi berikut ke three-address code:

```
x = (a + b) * (c - d) / e;
```

(c) Implementasikan representasi quadruples.

(d) Jelaskan perbedaan TAC, quadruples, dan triples.

6. **[Bab 13 - 20 poin]**

   (a) Jelaskan konsep activation record.

   (b) Trace eksekusi fungsi rekursif `factorial(3)`.

   (c) Bandingkan stack-based dan heap-based memory management.

7. **[Bab 14 - 25 poin]**

   (a) Implementasikan code generator sederhana.

   (b) Konversi TAC ke assembly-like code.

   (c) Jelaskan algoritma graph coloring untuk register allocation.

   (d) Bandingkan one-pass dan multi-pass code generation.

8. **[Bab 15 - 20 poin]**

   (a) Implementasikan optimasi dasar pada TAC.

   (b) Jelaskan data flow analysis.

   (c) Optimasi kode yang diberikan.

9. **[Integratif - 30 poin]** Rancang dan implementasikan kompilator sederhana untuk bahasa mini.

10. **[Integratif - 20 poin]** Analisis kompilator modern (GCC, Clang, atau lainnya) dan bandingkan dengan kompilator sederhana pada soal integratif sebelumnya.

Total: 220 poin

19.4 Jawaban Ujian Akhir Semester (UAS)

19.4.1 Jawaban Soal UAS

1. **[Bab 8 - 20 poin]**

(a) **Grammar specification untuk Bison/Yacc**

Grammar dengan semantic value dan precedence yang benar:

```
%{
#include <stdio.h>
#include "ast.h"
extern int yylex();
void yyerror(const char* s);
%}

%union {
    int ival;
    ASTNode* node;
}

%token <ival> NUMBER
%type  <node> expr

%left '+' '-'
%left '*' '/'
%right UMINUS

%%
expr:
      NUMBER            { $$ = createNumberNode($1); }
    | expr '+' expr     { $$ = createBinaryNode('+', $1, $3); }
    | expr '-' expr     { $$ = createBinaryNode('-', $1, $3); }
    | expr '*' expr     { $$ = createBinaryNode('*', $1, $3); }
    | expr '/' expr     { $$ = createBinaryNode('/', $1, $3); }
    | '-' expr %prec UMINUS { $$ = createUnaryNode('-', $2); }
    | '(' expr ')'      { $$ = $2; }
;
%%
```

Penjelasan:

- `%union` mendefinisikan tipe semantic value.
- `%token <ival>` menentukan tipe token.
- `%type <node>` menentukan tipe non-terminal.
- Deklarasi precedence menghindari ambiguity pada grammar ekspresi aritmetika.

(b) **Handling conflicts dalam Bison**

Shift-reduce conflict

Contoh grammar ambigu:

```
expr: expr '+' expr
```

```
    | expr '*' expr
    | NUMBER
```

Untuk input `1 + 2 * 3`, parser tidak dapat menentukan apakah harus melakukan shift atau reduce tanpa aturan precedence.

Solusi:

- Menambahkan deklarasi precedence:

```
%left '+'
%left '*'
```

Reduce-reduce conflict

Contoh grammar:

```
A: B
 | C
B: 'a'
C: 'a'
```

Token `a` dapat direduksi menjadi `B` atau `C`, sehingga terjadi reduce-reduce conflict.

Solusi:

- Menghilangkan ambiguity dengan memodifikasi grammar.

(c) **Implementasi kalkulator dengan Flex dan Bison**

Flex file (calc.l)

```
%{
#include "calc.tab.h"
#include <stdlib.h>
%}
%%
[0-9]+      { yylval = atoi(yytext); return NUMBER; }
[ \t\r]+    { /* skip whitespace */ }
\n          { return 0; }
.           { return yytext[0]; }
%%
```

Bison file (calc.y)

```
%{
#include <stdio.h>
int yylex();
void yyerror(const char* s) {
    fprintf(stderr, "Error: %s\n", s);
}
%}

%token NUMBER
```

```
%left '+' '-'
%left '*' '/'

%%
expr:
      NUMBER          { $$ = $1; }
    | expr '+' expr   { $$ = $1 + $3; }
    | expr '-' expr   { $$ = $1 - $3; }
    | expr '*' expr   { $$ = $1 * $3; }
    | expr '/' expr   {
        if ($3 == 0) yyerror("Division by zero");
        else $$ = $1 / $3;
      }
    | '(' expr ')'    { $$ = $2; }
;
%%
int main() {
    printf("Result: ");
    yyparse();
    return 0;
}
```

2. **[Bab 9 - 20 poin]**

   (a) **Struktur data AST**

   AST dirancang menggunakan class hierarchy dan Visitor Pattern:

```
class ASTNode {
public:
    virtual ~ASTNode() = default;
    virtual void accept(class Visitor* v) = 0;
};

class ExprNode : public ASTNode {};

class BinaryExpr : public ExprNode {
public:
    char op;
    ExprNode* left;
    ExprNode* right;
    void accept(Visitor* v) override;
};

class NumberNode : public ExprNode {
public:
    int value;
    void accept(Visitor* v) override;
```

```
};
```

(b) **Post-order traversal**

Post-order traversal mengevaluasi child sebelum parent, sehingga cocok untuk code generation.

```
void traverse(ASTNode* node, Visitor* v) {
    if (!node) return;
    node->accept(v);
}
```

(c) **AST visualizer**

```
void printAST(ASTNode* node, int indent = 0) {
    std::string pad(indent * 2, ' ');
    if (auto* b = dynamic_cast<BinaryExpr*>(node)) {
        std::cout << pad << "BinaryExpr(" << b->op << ")\n";
        printAST(b->left, indent + 1);
        printAST(b->right, indent + 1);
    } else if (auto* n = dynamic_cast<NumberNode*>(node)) {
        std::cout << pad << "Number(" << n->value << ")\n";
    }
}
```

3. **[Bab 10 - 20 poin]**

(a) **Symbol table dengan nested scope**

```
class SymbolTable {
private:
    std::vector<std::unordered_map<std::string, std::string>> scopes;

public:
    void enterScope() { scopes.push_back({}); }
    void exitScope() { scopes.pop_back(); }

    bool insert(const std::string& name, const std::string& type) {
        auto& scope = scopes.back();
        if (scope.count(name)) return false;
        scope[name] = type;
        return true;
    }

    std::string lookup(const std::string& name) {
        for (auto it = scopes.rbegin(); it != scopes.rend(); ++it)
            if (it->count(name)) return (*it)[name];
        return "";
    }
};
```

Kompleksitas:

- Insert: $O(1)$ rata-rata.
- Lookup: $O(s)$, dengan s jumlah scope.

(b) **Static vs Dynamic scoping**

Static scoping menentukan binding variabel berdasarkan struktur leksikal program, sedangkan dynamic scoping berdasarkan call stack saat runtime.

4. **[Bab 11 - 20 poin]**

(a) **Type checker**

```
enum Type { INT, FLOAT, BOOL, ERROR };

Type inferType(NumberNode* n) {
    return INT;
}
```

(b) **Semantic analysis**

Semantic analysis dilakukan dalam beberapa tahap:

i. Pengumpulan deklarasi.

ii. Pemeriksaan penggunaan identifier.

iii. Pemeriksaan tipe.

5. **[Bab 12 - 25 poin]**

(a) **Three-address code**

Untuk ekspresi:

```
x = (a + b) * (c - d) / e;
```

TAC:

```
t1 = a + b
t2 = c - d
t3 = t1 * t2
t4 = t3 / e
x = t4
```

6. **[Integratif] Rancangan kompilator**

Pipeline kompilator modern:

```
Source → Lexer → Parser → AST → IR → SSA → Optimization → Code Generation
```