# BUKU AJAR

# TEKNIK KOMPILASI

Berbasis Outcome-Based Education (OBE)

Praktis dengan C/C++


Oleh

**[Nama Dosen Pengampu]**


*Digunakan di lingkungan sendiri, sebagai buku ajar mata kuliah* ***Teknik Kompilasi*** *pada Program Studi S1 Teknik Informatika*


**Program Studi S1 Teknik Informatika**

Fakultas Teknik

Universitas



**2026**

# INFORMASI BUKU


| | |
|---|---|
| **Judul** | : Buku Ajar Teknik Kompilasi |
| **Subjudul** | : Berbasis Outcome-Based Education (OBE), Praktis dengan C/C++ |
| **Penulis** | : [Nama Dosen Pengampu] |
| **Program Studi** | : S1 Teknik Informatika |
| **Fakultas** | : Fakultas Teknik |
| **Universitas** | : Universitas |
| **Mata Kuliah** | : Teknik Kompilasi |
| **SKS** | : 3 SKS |
| **Pertemuan** | : 16 Pertemuan |
| **Tahun** | : 2026 |
| **ISBN** | : [ISBN jika ada] |


*Buku ajar ini disusun sebagai bahan pembelajaran untuk mata kuliah **Teknik Kompilasi** pada Program Studi S1 Teknik Informatika. Buku ini dirancang mengikuti pendekatan **Outcome-Based Education (OBE)** dengan fokus pada pembelajaran berbasis praktik.*

# Prakata

Buku ajar ini disusun sebagai bahan pembelajaran untuk mata kuliah **Teknik Kompilasi** pada Program Studi S1 Teknik Informatika. Buku ini dirancang mengikuti pendekatan *Outcome-Based Education (OBE)* dengan fokus pada pembelajaran berbasis praktik.

Buku ini bertujuan untuk memberikan pemahaman dan keterampilan praktis dalam merancang dan mengimplementasikan kompilator untuk bahasa pemrograman. Setiap bab dilengkapi dengan contoh praktis menggunakan bahasa pemrograman C atau C++, serta latihan yang mengarah pada pembangunan komponen-komponen kompilator secara bertahap.

Mata kuliah ini mencakup fase-fase kompilator dari Analisis Leksikal (Lexical Analysis), Analisis Sintaksis (Syntax Analysis), Analisis Semantik (Semantic Analysis), hingga Generasi Kode (Code Generation). Sesuai dengan pendekatan OBE, mahasiswa diharapkan tidak hanya memahami teori, tetapi juga mampu mengimplementasikan setiap fase kompilator secara praktis.

*Penyusun*
*28 Januari 2026*

# Daftar Isi

# Daftar Gambar

# Daftar Tabel

# Bab 1

## Pengenalan Kompilator dan Fase-Fase Kompilasi

### 1.1 Tujuan Pembelajaran

Setelah mempelajari bab ini, mahasiswa diharapkan mampu:

1. Menjelaskan definisi dan konsep dasar kompilator
2. Memahami perbedaan antara kompilator, interpreter, dan translator lainnya
3. Mengidentifikasi fase-fase utama dalam proses kompilasi
4. Menjelaskan arsitektur kompilator secara keseluruhan
5. Memahami alur kerja dari source code hingga executable code

### 1.2 Apa itu Kompilator?

Kompilator adalah program komputer yang menerjemahkan source code (kode sumber) yang ditulis dalam bahasa pemrograman tingkat tinggi menjadi target code (kode target) dalam bahasa yang dapat dieksekusi oleh komputer, seperti assembly language atau machine code.

#### 1.2.1 Definisi Kompilator

Secara formal, kompilator adalah program yang melakukan translasi dari bahasa sumber (source language) ke bahasa target (target language), dengan mempertahankan makna semantik dari program sumber. Proses translasi ini disebut **kompilasi**.

Menurut Aho, Lam, Sethi, dan Ullman dalam buku klasik "Compilers: Principles, Techniques, and Tools"[1]:

> “A compiler is a program that can read a program in one language (the source language) and translate it into an equivalent program in another language (the target language).”

### 1.2.2 Karakteristik Kompilator

Kompilator memiliki beberapa karakteristik penting:

- **Translasi Lengkap**: Kompilator membaca seluruh program sumber sebelum menghasilkan output, berbeda dengan interpreter yang mengeksekusi baris demi baris.
- **Analisis Mendalam**: Kompilator melakukan analisis mendalam terhadap struktur program, termasuk pengecekan syntax, semantic, dan optimasi.
- **Output Terpisah**: Hasil kompilasi adalah file terpisah (executable atau object file) yang dapat dieksekusi tanpa perlu source code.
- **Optimasi**: Kompilator dapat melakukan berbagai optimasi untuk meningkatkan efisiensi program yang dihasilkan.

### 1.2.3 Perbedaan Kompilator dengan Interpreter

Meskipun keduanya menerjemahkan kode, kompilator dan interpreter memiliki perbedaan fundamental:

- **Kompilator**: Menerjemahkan seluruh program sekaligus menjadi executable file sebelum eksekusi. Contoh: C, C++, Rust.
- **Interpreter**: Menerjemahkan dan mengeksekusi program baris demi baris secara langsung tanpa menghasilkan file terpisah. Contoh: Python, JavaScript, Ruby.
- **Hybrid**: Beberapa bahasa menggunakan kombinasi keduanya, seperti Java (compile ke bytecode, kemudian diinterpretasi oleh JVM).

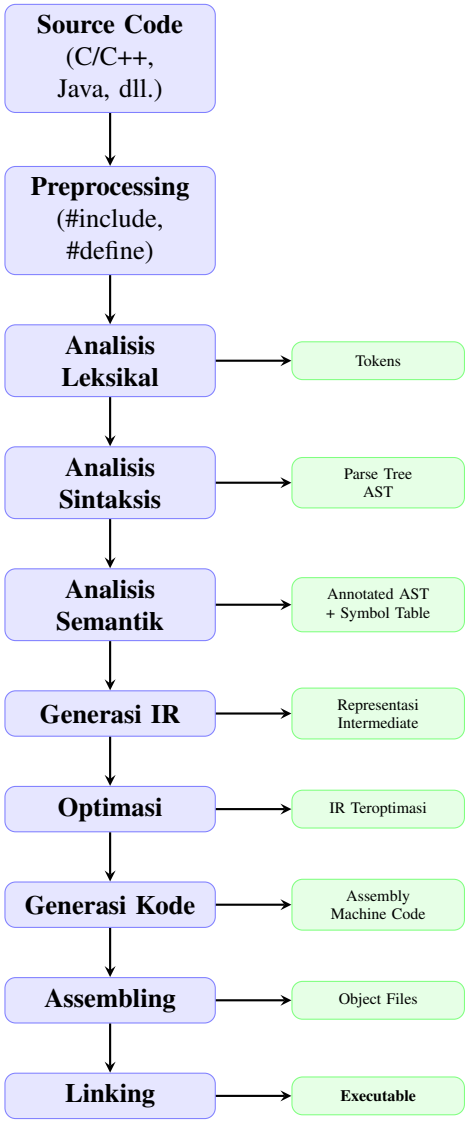
### 1.2.4 Peran Kompilator dalam Pengembangan Software

Kompilator memainkan peran penting dalam ekosistem pengembangan software modern:

1. **Bridge antara High-Level dan Low-Level**: Memungkinkan programmer menulis kode dalam bahasa yang mudah dipahami manusia, sementara komputer menjalankan instruksi tingkat rendah yang efisien.
2. **Error Detection**: Mendeteksi berbagai jenis error (syntax, semantic, type) sebelum program dieksekusi.
3. **Optimization**: Menerapkan berbagai teknik optimasi untuk menghasilkan kode yang lebih efisien.
4. **Portability**: Memungkinkan program yang sama dikompilasi untuk berbagai platform dan arsitektur.

## 1.3 Alur Kerja Kompilator: Gambaran Umum

Sebelum membahas detail arsitektur dan fase-fase kompilasi, mari kita lihat gambaran umum alur kerja kompilator dari source code hingga executable. Gambar 1.1 menunjukkan alur kerja kompilator secara keseluruhan:




Gambar 1.1: Alur kerja kompilator dari source code ke executable

Dari gambar di atas, dapat dilihat bahwa proses kompilasi melibatkan beberapa tahap utama:

1. **Preprocessing**: Memproses directive khusus sebelum kompilasi

2. **Analisis** (Front-end): Analisis Leksikal (Lexical Analysis), Analisis Sintaksis (Syntax Analysis), dan Analisis Semantik (Semantic Analysis)

3. **Sintesis** (Back-end): Generasi IR (IR Generation), Optimasi (Optimization), dan Generasi Kode (Code Generation)
4. **Assembling dan Linking**: Mengubah assembly menjadi executable

### 1.3.1 Preprocessing

Sebelum kompilasi dimulai, preprocessor memproses directive khusus seperti:

- `#include`: Menyisipkan konten file header
- `#define`: Makro definisi
- `#ifdef`, `#ifndef`: Conditional compilation

### 1.3.2 Assembling dan Linking

Setelah Generasi Kode (Code Generation), assembler mengubah assembly code menjadi object code (file .o atau .obj). Linker kemudian menyatukan:

- Object files dari source code yang dikompilasi
- Library files (static atau dynamic libraries)
- Startup code

Menjadi satu executable file yang siap dieksekusi.

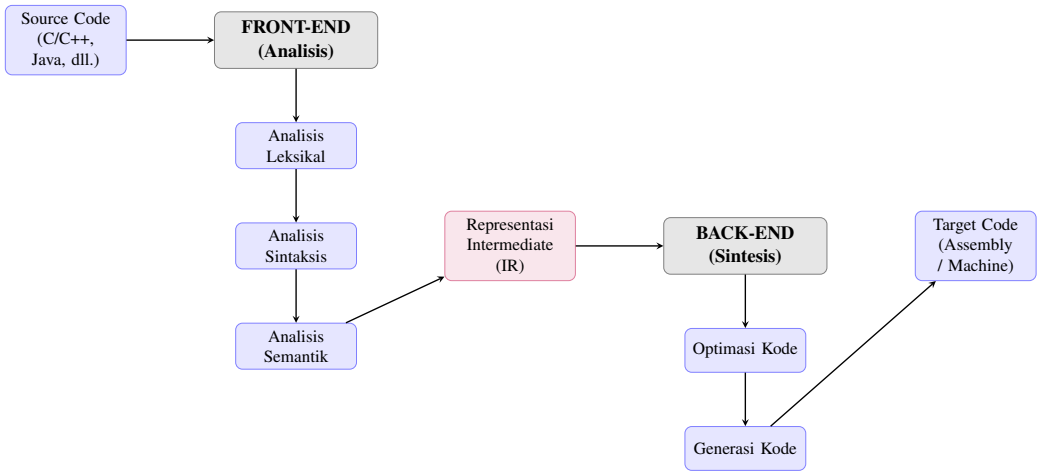
## 1.4 Arsitektur Kompilator

Kompilator modern umumnya dibagi menjadi dua bagian utama: **front-end** dan **back-end**. Gambar 20.1 menunjukkan struktur arsitektur kompilator secara keseluruhan.

### 1.4.1 Front-End (Analisis)

Front-end bertanggung jawab untuk menganalisis source code dan membangun representasi internal. Menurut sumber terbuka:

> "A typical compiler front end comprises several sequential phases: lexical analysis (scanning), syntax analysis (parsing), and semantic analysis. Lexical analysis breaks input text into lexemes which correspond to tokens, eliminating comments and whitespace. Syntax analysis checks grammar validity and builds a structural representation (parse tree or AST). Semantic analysis performs scope resolution, type checking, name resolution, and checks language-specific semantic rules."[2]




Gambar 1.2: Arsitektur kompilator: Front-end (analisis) dan Back-end (sintesis)

Fase-fase dalam front-end meliputi:

1. **Analisis Leksikal (Lexical Analysis)**: Memecah source code menjadi token-token (identifiers, keywords, operators, literals, dll.)

2. **Analisis Sintaksis (Syntax Analysis)**: Menganalisis struktur grammar dan membangun parse tree atau Abstract Syntax Tree (AST)

3. **Analisis Semantik (Semantic Analysis)**: Memeriksa aturan semantik bahasa, seperti type checking, scope resolution, dan name resolution

4. **Generasi Kode Intermediate (Intermediate Code Generation)**: Mengubah AST menjadi representasi intermediate (misalnya three-address code, quadruples, atau bytecode)

### 1.4.2 Back-End (Sintesis)

Setelah front-end selesai menganalisis source code dan menghasilkan representasi intermediate, back-end bertanggung jawab untuk menghasilkan target code dari representasi tersebut. Fase-fase dalam back-end meliputi:

1. **Optimasi Kode (Code Optimization)**: Mengoptimasi kode intermediate untuk meningkatkan efisiensi tanpa mengubah semantik

2. **Generasi Kode (Code Generation)**: Menghasilkan target code (assembly atau machine code) dari kode intermediate yang sudah dioptimasi

## 1.5 Fase-Fase Kompilasi Secara Detail

Setelah memahami gambaran umum arsitektur kompilator, sekarang kita akan mempelajari setiap fase kompilasi secara lebih mendalam. Setiap fase memiliki peran spesifik dalam transformasi source code menjadi executable code. Mari kita pelajari setiap fase berdasarkan sumber dari UC San Diego[3]:

### 1.5.1 Fase 1: Analisis Leksikal (Lexical Analysis)

Fase pertama dalam proses kompilasi adalah Analisis Leksikal (Lexical Analysis) atau scanning. Fase ini merupakan langkah awal yang mengubah source code (berupa string karakter) menjadi stream token yang bermakna. Tujuan fase ini adalah:

- Membaca source code karakter demi karakter
- Mengelompokkan karakter menjadi token-token bermakna
- Mengeliminasi whitespace, comments, dan karakter yang tidak relevan
- Melacak informasi posisi (baris, kolom) untuk error reporting

Contoh: Source code `int x = 42;` akan dipecah menjadi token-token:

- `int` (keyword)
- `x` (identifier)
- = (operator assignment)
- `42` (integer literal)
- `;` (punctuation/semicolon)

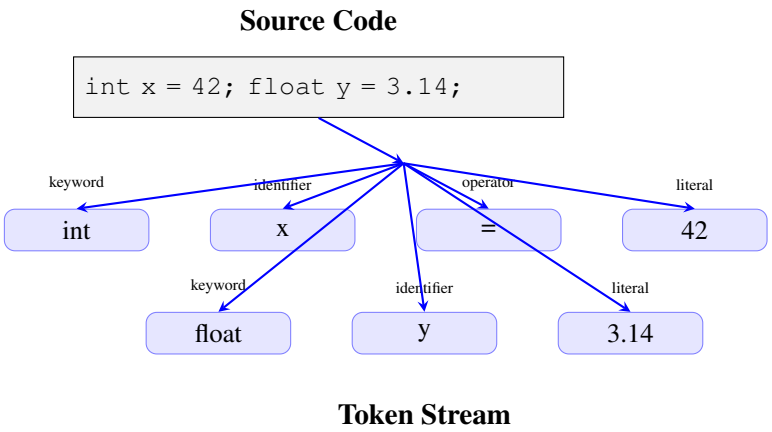
Gambar 1.3 menunjukkan contoh proses tokenization untuk source code yang lebih kompleks.

Analisis Leksikal biasanya diimplementasikan menggunakan finite automata dan regular expressions. Tools seperti Flex, re2c, atau implementasi manual dapat digunakan.

### 1.5.2 Fase 2: Analisis Sintaksis (Syntax Analysis)

Setelah Analisis Leksikal menghasilkan stream token, fase berikutnya adalah Analisis Sintaksis (Syntax Analysis) atau parsing. Fase ini mengambil stream token dari lexical analyzer dan memverifikasi bahwa token-token tersebut membentuk struktur yang valid menurut grammar bahasa tersebut.

Hasil dari parsing biasanya berupa:



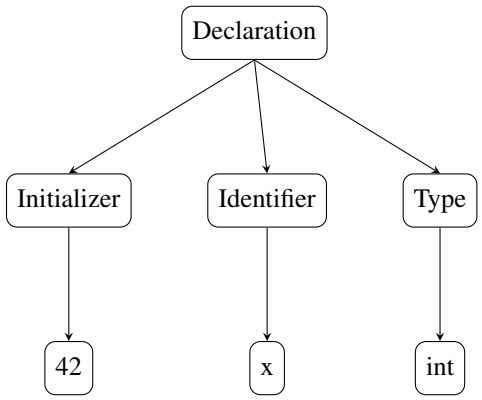

Gambar 1.3: Contoh proses lexical analysis: dari source code ke token stream

- **Parse Tree**: Representasi lengkap dari struktur grammar, termasuk semua non-terminal
- **Abstract Syntax Tree (AST)**: Representasi yang lebih abstrak, hanya menyertakan informasi yang relevan untuk fase selanjutnya

Contoh: token-token `int x = 42;` akan di-parse menjadi struktur berikut:

```
Declaration
||-- Type: int
||-- Identifier: x
`-- Initializer: 42
```

Gambar 1.4 menunjukkan contoh parse tree untuk deklarasi variabel sederhana.




Gambar 1.4: Contoh parse tree untuk deklarasi `int x = 42;`

Parsing dapat dilakukan dengan berbagai metode:

- Top-down parsing (recursive descent, LL parsers)

- Bottom-up parsing (LR, LALR, GLR parsers)
- Menggunakan parser generators seperti Bison, Yacc, atau ANTLR

### 1.5.3 Fase 3: Analisis Semantik (Semantic Analysis)

Setelah Analisis Sintaksis memastikan bahwa struktur program valid secara grammar, Analisis Semantik (Semantic Analysis) memastikan bahwa program juga memenuhi aturan semantik bahasa. Meskipun program sudah valid secara sintaks, belum tentu valid secara semantik. Tugas utama fase ini meliputi:

- **Type Checking**: Memastikan tipe data yang digunakan sesuai dengan aturan bahasa
- **Scope Resolution**: Menyelesaikan referensi variabel dan fungsi ke deklarasi yang sesuai
- **Name Resolution**: Memastikan setiap identifier merujuk ke deklarasi yang valid
- **Contextual Checks**: Memeriksa aturan spesifik bahasa (misalnya: break hanya dalam loop, return type match, dll.)

Misalnya, semantic analyzer akan memeriksa:

- Apakah variabel digunakan sebelum dideklarasi?
- Apakah operasi aritmatika dilakukan pada tipe yang kompatibel?
- Apakah fungsi dipanggil dengan jumlah dan tipe parameter yang benar?

### 1.5.4 Fase 4: Generasi Kode Intermediate (Intermediate Code Generation)

Setelah Analisis Semantik selesai dan memastikan program valid secara semantik, kompilator menghasilkan Representasi Intermediate (IR). IR adalah representasi program yang berada di antara AST (yang masih dekat dengan source code) dan target code (yang dekat dengan machine code). IR adalah representasi program yang:

- Lebih dekat ke machine code dibanding AST, tetapi tetap machine-independent
- Memudahkan optimasi karena lebih sederhana dari AST
- Memungkinkan portabilitas (IR yang sama dapat digunakan untuk berbagai target platform)

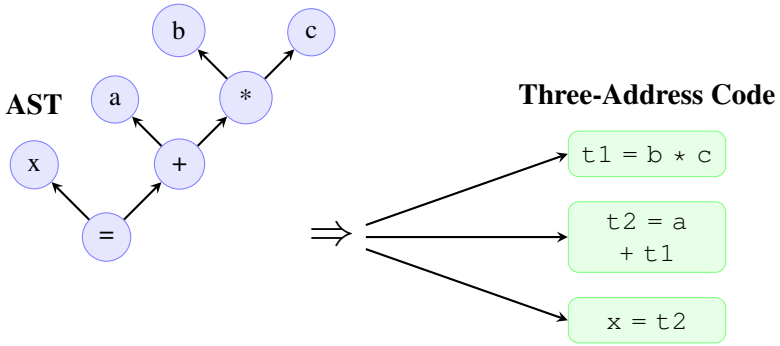
Bentuk IR yang umum digunakan:

- **Three-Address Code (TAC)**: Setiap instruksi memiliki paling banyak tiga operand
- **Quadruples**: Format IR dengan operator, dua operan, dan satu hasil
- **Static Single Assignment (SSA)**: Setiap variabel hanya di-assign sekali, memudahkan optimasi
- **Bytecode**: Untuk bahasa yang diinterpretasi (seperti Java, Python)

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

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

Gambar 1.5 menunjukkan visualisasi proses generasi TAC dari AST.




Gambar 1.5: Generasi Three-Address Code dari AST untuk ekspresi `x = a + b * c`

### 1.5.5 Fase 5: Optimasi Kode (Code Optimization)

Setelah IR dihasilkan, fase Optimasi Kode (Code Optimization) bertujuan untuk meningkatkan kualitas kode yang dihasilkan tanpa mengubah semantik program. Optimasi ini penting untuk menghasilkan kode yang lebih efisien dalam hal waktu eksekusi dan penggunaan memori. Optimasi dapat dilakukan pada berbagai level:

- **Local Optimization**: Optimasi dalam basic block (satu entry, satu exit)
  - Constant folding: `x = 3 + 5` $\rightarrow$ `x = 8`
  - Constant propagation: Mengganti variabel dengan nilai konstantanya
  - Dead code elimination: Menghapus kode yang tidak pernah dieksekusi
- **Global Optimization**: Optimasi lintas basic blocks
  - Common subexpression elimination

  - Loop optimization (loop unrolling, loop invariant code motion)
  - Data flow analysis

- **Machine-Specific Optimization**: Optimasi yang memanfaatkan fitur hardware tertentu

Menurut sumber dari UC San Diego, Northeastern University, dan sumber lainnya[4], optimasi harus menyeimbangkan antara:

- Waktu kompilasi
- Kualitas kode yang dihasilkan
- Konsumsi memory compiler

### 1.5.6 Fase 6: Generasi Kode (Code Generation)

Fase terakhir dalam proses kompilasi adalah Generasi Kode (Code Generation), yaitu menghasilkan target code dari IR yang telah dioptimasi. Fase ini mengubah representasi intermediate menjadi kode yang dapat dieksekusi oleh mesin target. Code generator bertanggung jawab untuk:

- **Instruction Selection**: Memilih instruksi machine yang tepat untuk setiap operasi IR
- **Register Allocation**: Mengalokasikan register untuk variabel (register terbatas, variabel banyak)
- **Instruction Scheduling**: Mengatur urutan instruksi untuk memaksimalkan penggunaan pipeline processor
- **Address Assignment**: Mengalokasikan memory untuk variabel dan data structures

Contoh: TAC `x = a + b` dapat di-generate menjadi assembly:

```
LOAD R1, a
LOAD R2, b
ADD R3, R1, R2
STORE R3, x
```

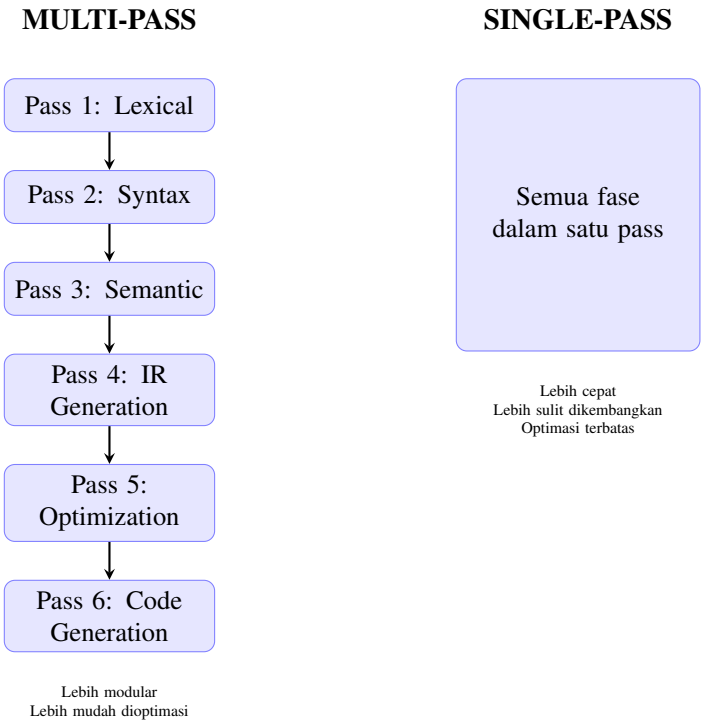
## 1.6 Kompilator Modern: Multi-Pass vs Single-Pass

Setelah memahami arsitektur dan fase-fase kompilasi, penting untuk mengetahui bahwa kompilator modern umumnya menggunakan pendekatan **multi-pass**, di mana setiap fase dijalankan dalam pass terpisah. Keuntungannya:

- Memisahkan concern (tiap fase fokus pada tugas spesifik)
- Memudahkan maintenance dan debugging
- Memungkinkan optimasi yang lebih kompleks

Sebaliknya, kompilator **single-pass** mencoba menyelesaikan semua fase dalam satu pass. Pendekatan ini lebih cepat tetapi lebih sulit diimplementasikan dan terbatas dalam optimasi.

Gambar 1.6 menunjukkan perbedaan antara kedua pendekatan.




Gambar 1.6: Perbandingan arsitektur Multi-Pass dan Single-Pass Compiler

## 1.7 Contoh Praktis: Alur Kompilasi Program C Sederhana

Untuk memperkuat pemahaman tentang fase-fase kompilasi yang telah dipelajari, mari kita lihat contoh konkret bagaimana sebuah program C sederhana diproses melalui setiap fase kompilasi. Contoh ini akan menunjukkan aplikasi praktis dari konsep-konsep yang telah dibahas.

Mari kita lihat contoh konkret bagaimana sebuah program C sederhana diproses melalui fase-fase kompilasi:

Listing 1.1: Program C sederhana: hello.c

```
1 #include <stdio.h>
2
3 int main() {
```

```
4     int x = 10;
5     int y = 20;
6     int sum = x + y;
7     printf("Sum = %d\n", sum);
8     return 0;
9 }
```

### Setelah Preprocessing

Preprocessor akan mengganti `#include <stdio.h>` dengan isi file header tersebut (biasanya ratusan baris deklarasi fungsi).

### Setelah Lexical Analysis

Source code dipecah menjadi token-token seperti yang ditunjukkan pada Tabel 1.1.

Tabel 1.1: Token stream hasil lexical analysis untuk program hello.c

| Token | Token Type |
|---|---|
| `int` | KEYWORD |
| `main` | IDENTIFIER |
| `(` | PUNCTUATION (LPAREN) |
| `)` | PUNCTUATION (RPAREN) |
| `{` | PUNCTUATION (LBRACE) |
| `int` | KEYWORD |
| `x` | IDENTIFIER |
| `=` | OPERATOR (ASSIGN) |
| `10` | INTEGER_LITERAL |
| `;` | PUNCTUATION (SEMICOLON) |
| `int` | KEYWORD |
| `y` | IDENTIFIER |
| `=` | OPERATOR (ASSIGN) |
| `20` | INTEGER_LITERAL |
| `;` | PUNCTUATION (SEMICOLON) |
| `int` | KEYWORD |
| `sum` | IDENTIFIER |
| `=` | OPERATOR (ASSIGN) |
| `x` | IDENTIFIER |
| `+` | OPERATOR (PLUS) |

Dilanjutkan pada halaman berikutnya

**Tabel 1.1 – lanjutan dari halaman sebelumnya**

| Token | Token Type |
|---|---|
| `y` | IDENTIFIER |
| `;` | PUNCTUATION (SEMICOLON) |
| `printf` | IDENTIFIER |
| `(` | PUNCTUATION (LPAREN) |
| `"Sum = %d\n"` | STRING_LITERAL |
| `,` | PUNCTUATION (COMMA) |
| `sum` | IDENTIFIER |
| `)` | PUNCTUATION (RPAREN) |
| `;` | PUNCTUATION (SEMICOLON) |
| `return` | KEYWORD |
| `0` | INTEGER_LITERAL |
| `;` | PUNCTUATION (SEMICOLON) |
| `}` | PUNCTUATION (RBRACE) |

### Setelah Syntax Analysis

Parser membangun AST yang menunjukkan struktur program. Bagian AST untuk statement `int sum = x + y;` ditunjukkan pada Gambar 1.7.

### Setelah Semantic Analysis

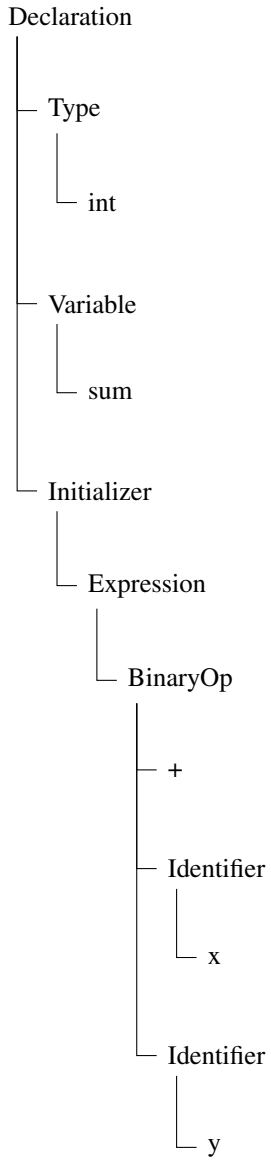
Semantic analyzer memeriksa:

- Variabel `x` dan `y` sudah dideklarasi sebelum digunakan
- Tipe data `int` kompatibel untuk operasi penjumlahan
- Fungsi `printf` dideklarasi di `stdio.h`
- Format string `"Sum = %d\n"` sesuai dengan parameter `sum` (tipe `int`)

### Setelah Intermediate Code Generation

TAC yang dihasilkan untuk bagian `sum = x + y;`:

```
t1 = x + y
sum = t1
```




Gambar 1.7: AST untuk deklarasi `int sum = x + y;`

**Setelah Code Generation**

Assembly code yang dihasilkan (contoh untuk x86-64):

```
mov eax, DWORD PTR [rbp-4]    ; Load x
add eax, DWORD PTR [rbp-8]    ; Add y
mov DWORD PTR [rbp-12], eax   ; Store to sum
```

## 1.8 Contoh Kompilator Sederhana

Sebagai referensi pembelajaran, terdapat beberapa kompilator sederhana yang dapat dipelajari untuk memahami implementasi praktis dari fase-fase kompilasi:

- **TinyCC (TCC)**: Kompilator C kecil dan cepat yang menunjukkan implementasi praktis dari fase-fase kompilasi. TCC dapat digunakan sebagai referensi pembelajaran karena ukurannya yang relatif kecil namun lengkap.
- **AnjaneyaTripathi's C Compiler**[1]: Implementasi compiler sederhana menggunakan Flex dan Bison, yang sangat berguna untuk pembelajaran karena menunjukkan penggunaan tools generator (lexer dan parser generator).
- **Project Open Source Lainnya**: Banyak project open source yang tersedia di GitHub yang mengimplementasikan compiler sederhana untuk berbagai bahasa, yang dapat digunakan sebagai bahan pembelajaran.

## 1.9 Proyek Buku: Compiler Subset C

Sepanjang Bab 2 hingga Bab 16, kita secara bertahap membangun **satu compiler untuk subset bahasa C**. Setiap bab menambah satu lapis ke proyek yang sama: spesifikasi token (Bab 2), lexer hand-written (Bab 3), lexer Flex (Bab 4), grammar (Bab 5), parser hand-written (Bab 6), teori bottom-up (Bab 7), parser Bison (Bab 8), AST (Bab 9), symbol table (Bab 10), type checking (Bab 11), IR (Bab 12), runtime (Bab 13), code generation (Bab 14), optimasi (Bab 15), dan integrasi (Bab 16). Spesifikasi berikut menjadi acuan tunggal agar semua contoh dan kode mengacu ke bahasa yang sama.

### 1.9.1 Spesifikasi Token Proyek Subset C

Token yang dikenali oleh compiler proyek (untuk Bab 2–4):

- **Identifier**: huruf atau underscore diikuti huruf, angka, atau underscore. Pola: `[a-zA-Z_][a-zA-Z0-9_]*`

[1] https://github.com/AnjaneyaTripathi/c-compiler

- **Kata kunci**: `int`, `float`, `print`. (Nanti dapat diperluas: `if`, `else`, `while`.)
- **Literal**: integer `[0-9]+`, float `[0-9]+.[0-9]+`, string `"..."` dalam tanda kutip ganda.
- **Operator**: `+`, `-`, `*`, `/`, `=`, `==`, `!=`, `<`, `>`, `<=`, `>=`.
- **Punctuator**: `;`, `,`, `()`, kurung kurawal `{}`.
- **Komentar**: satu baris `//` dan banyak baris `/* */`; serta whitespace (spasi, tab, newline) diabaikan.

### 1.9.2 Spesifikasi Grammar Proyek Subset C

Grammar dalam BNF untuk Bab 5–8 (dan parser proyek):

- **Program**: barisan statement.
- **Statement**: deklarasi `;` | assignment `;` | print-statement `;`
- **Deklarasi**: `int` identifier | `float` identifier
- **Assignment**: identifier = ekspresi
- **Print-statement**: `print (` string-literal `)` | `print (` ekspresi `)`
- **Ekspresi**: term | ekspresi + term | ekspresi − term
- **Term**: factor | term `*` factor | term `/` factor
- **Factor**: literal | identifier | `(` ekspresi `)`

Precedence: `*` dan `/` lebih tinggi dari + dan −; associativity kiri untuk semuanya.

### 1.9.3 Peta Bab ke Lapis Proyek

| Bab | Lapis proyek |
|---|---|
| 2 | Spesifikasi token + teori RE/FA |
| 3 | Lexer hand-written (mengikuti spec token) |
| 4 | Lexer proyek (Flex, file `simplec.l`) |
| 5 | Grammar proyek (BNF/EBNF di atas) |
| 6 | Parser hand-written (mengikuti grammar proyek) |
| 7 | Teori LR; grammar proyek termasuk kelas LR |
| 8 | Parser proyek (Bison, file `simplec.y`) |
| 9 | AST proyek (`ast.h`/`ast.c`) |
| 10 | Symbol table proyek (`symtab.h`/`symtab.c`) |
| 11 | Type checking proyek |
| 12 | IR proyek (TAC/quadruples dari AST) |
| 13 | Runtime; asumsi proyek untuk stack/activation record |
| 14 | Code generation proyek (IR $\rightarrow$ assembly) |
| 15 | Optimasi proyek (basic block, constant folding, dll.) |
| 16 | Integrasi dan presentasi compiler subset C lengkap |

Semua bab dari Bab 2 sampai Bab 16 merujuk ke spesifikasi ini. Kode dan contoh dalam bab tersebut mengacu ke token set dan grammar di atas, serta ke file proyek (`simplec.l`, `simplec.y`, dan seterusnya) yang tumbuh di folder `proyek-compiler-subset-c/`.

## 1.10 Kesimpulan

Dalam bab ini, kita telah mempelajari konsep dasar kompilator dan fase-fase kompilasi. Ringkasan materi yang telah dibahas:

1. **Definisi Kompilator**: Kompilator adalah program yang menerjemahkan source code ke target code melalui beberapa fase yang terstruktur.

2. **Perbedaan dengan Translator Lainnya**: Kompilator berbeda dengan interpreter dan translator lainnya (assembler, linker, loader, preprocessor, decompiler) dalam cara dan tujuan translasinya.

3. **Arsitektur Kompilator**: Kompilator modern terdiri dari dua bagian utama:

    - **Front-end (Analisis)**: Analisis Leksikal (Lexical Analysis), Analisis Sintaksis (Syntax Analysis), dan Analisis Semantik (Semantic Analysis)
    - **Back-end (Sintesis)**: Generasi Kode Intermediate (Intermediate Code Generation), Optimasi Kode (Code Optimization), dan Generasi Kode (Code Generation)

4. **Enam Fase Utama Kompilasi**: Setiap fase memiliki peran spesifik dalam transformasi source code menjadi executable:
    - Analisis Leksikal (Lexical Analysis): Tokenization
    - Analisis Sintaksis (Syntax Analysis): Parsing dan pembangunan AST
    - Analisis Semantik (Semantic Analysis): Type checking dan scope resolution
    - Generasi Kode Intermediate (Intermediate Code Generation): Pembuatan IR
    - Optimasi Kode (Code Optimization): Optimasi kode
    - Generasi Kode (Code Generation): Generasi target code
5. **Alur Kerja Lengkap**: Proses dari source code hingga executable melibatkan preprocessing, enam fase kompilasi utama, assembling, dan linking.
6. **Pendekatan Modern**: Kompilator modern menggunakan pendekatan multi-pass yang lebih modular dan memungkinkan optimasi yang lebih kompleks dibanding single-pass.

Pemahaman terhadap arsitektur dan fase-fase kompilasi ini menjadi dasar penting untuk mempelajari implementasi praktis setiap fase dalam bab-bab selanjutnya. Dari Bab 2 hingga Bab 16, kita secara bertahap membangun satu compiler untuk subset bahasa C; spesifikasi token dan grammar proyek dapat dilihat pada Bagian 1.9. Setiap fase akan dibahas secara lebih mendalam dengan contoh yang mengacu ke proyek compiler subset C tersebut.

## 1.11 Referensi dan Bahan Bacaan Lanjutan

Untuk memperdalam pemahaman tentang pengenalan kompilator, mahasiswa disarankan membaca:

- **Dragon Book**: Aho, Lam, Sethi, & Ullman (2006). *Compilers: Principles, Techniques, and Tools* [1] - Bab 1: Introduction
- **Engineering a Compiler**: Cooper & Torczon (2011) [5] - Bab 1: Overview of Compilation
- **UC San Diego CSE 231**: Course materials tentang compiler construction [3]
- **Northeastern University CS 4410**: Comprehensive compiler design course [4]
- **Johns Hopkins University EN.601.428**: Course tentang compilers dan interpreters [6]

## 1.12 Evaluasi

Untuk menguji pemahaman Anda terhadap materi Bab 1, silakan jawab pertanyaan-pertanyaan berikut:

1. Jelaskan perbedaan mendasar antara kompilator dan interpreter dalam hal eksekusi program dan output yang dihasilkan!
2. Sebutkan dan jelaskan secara singkat tiga fase utama dalam Front-End kompilator!
3. Apa fungsi utama dari Analisis Semantik (Semantic Analysis)? Berikan satu contoh kesalahan yang dapat dideteksi oleh fase ini!
4. Mengapa Generasi Kode Intermediate (Intermediate Code Generation / IR) penting dalam arsitektur kompilator modern? Jelaskan hubungannya dengan portabilitas!
5. Apa tujuan utama dari Optimasi Kode (Code Optimization)? Mengapa optimasi perlu memperhatikan waktu kompilasi?

# Bab 2

## Regular Expression dan Finite Automata untuk Lexical Analysis

### 2.1 Tujuan Pembelajaran

Setelah mempelajari bab ini, mahasiswa diharapkan mampu:

1. Memahami konsep regular expression dan regular language
2. Menjelaskan perbedaan antara NFA (Nondeterministic Finite Automata) dan DFA (Deterministic Finite Automata)
3. Mengkonversi regular expression ke NFA menggunakan algoritma Thompson
4. Mengkonversi NFA ke DFA menggunakan subset construction
5. Mengimplementasikan NFA dan DFA sederhana dalam C/C++
6. Membuat recognizer untuk pattern token sederhana menggunakan finite automata
7. Memahami hubungan antara regular expression, finite automata, dan lexical analysis

### 2.2 Pendahuluan

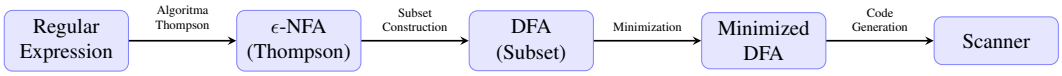
Spesifikasi token untuk proyek compiler subset C telah didefinisikan di Bab 1 (Bagian 1.9). Dalam bab ini kita mempelajari teori formal yang mendasari lexical analysis—regular expression dan finite automata—dan mengaitkan contoh ke token-token proyek (identifier, number, keyword, operator, punctuator) bila relevan.

Sebagai landasan untuk memahami lexical analysis, kita perlu mempelajari teori formal yang mendasarinya. Menurut sumber dari Aoyama Gakuin University:

> "Lexical analysis breaks input text into lexemes which correspond to tokens. Usually implemented using regular languages $\rightarrow$ regex $\rightarrow$ NFA $\rightarrow$ DFA $\rightarrow$ (minimized) DFA for efficiency."[7]

Alur ini menunjukkan bahwa lexical analysis dalam kompilator modern menggunakan teori formal language, khususnya regular languages, yang direpresentasikan sebagai regular expressions dan kemudian diimplementasikan sebagai finite automata untuk efisiensi.

Gambar 2.1 menunjukkan alur lengkap dari regular expression hingga implementasi praktis dalam lexical analyzer.




Gambar 2.1: Alur konversi dari regular expression ke implementasi scanner

## 2.3 Regular Expression dan Regular Language

### 2.3.1 Definisi Regular Expression

Regular expression (regex) adalah notasi formal untuk mendeskripsikan pola string dalam suatu bahasa. Regular expression menggunakan operasi-operasi dasar untuk membangun pattern yang lebih kompleks.

Operasi-operasi dasar dalam regular expression meliputi:

1. **Literal**: Karakter tunggal, misalnya `a` mencocokkan string “a”
2. **Concatenation**: Penggabungan, misalnya `ab` mencocokkan string “ab”
3. **Union/Alternation**: Pilihan, misalnya `a|b` mencocokkan “a” atau “b”
4. **Kleene Star**: Nol atau lebih pengulangan, misalnya `a*` mencocokkan “”, “a”, “aa”, “aaa”, dll.
5. **Kleene Plus**: Satu atau lebih pengulangan, misalnya `a+` mencocokkan “a”, “aa”, “aaa”, dll.
6. **Optional**: Nol atau satu, misalnya `a?` mencocokkan “” atau “a”
7. **Character Class**: Set karakter, misalnya `[0-9]` mencocokkan digit 0-9

### 2.3.2 Contoh Regular Expression untuk Token

Dalam lexical analysis, setiap jenis token didefinisikan menggunakan regular expression. Berikut beberapa contoh:

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

  - Dimulai dengan huruf atau underscore
  - Diikuti oleh nol atau lebih huruf, digit, atau underscore

- **Integer Literal**: `[0-9]+`
  - Satu atau lebih digit
- **Floating Point**: `[0-9]+\.[0-9]+`
  - Digit, titik desimal, digit
- **String Literal**: `"([^"]|\\.)*"`
  - Dimulai dan diakhiri dengan tanda kutip
  - Berisi karakter apapun kecuali tanda kutip (atau escape sequence)
- **Whitespace**: `[ \t\n]+`
  - Satu atau lebih spasi, tab, atau newline
- **Operator**: `+|-|*|/|=|==|!=`
  - Operator aritmatika dan perbandingan

### 2.3.3 Regular Language

Bahasa yang dapat dinyatakan dengan regular expression disebut **regular language**. Regular language memiliki sifat-sifat penting:

- Dapat dikenali oleh finite automata (NFA atau DFA)
- Tertutup terhadap operasi union, concatenation, dan Kleene star
- Tidak dapat mengekspresikan struktur nested (seperti matching parentheses)
- Cukup untuk mendeskripsikan sebagian besar token dalam bahasa pemrograman

## 2.4 Finite Automata

Finite automata adalah model matematika yang digunakan untuk mengenali string dalam suatu bahasa. Terdapat dua jenis utama: NFA (Nondeterministic Finite Automata) dan DFA (Deterministic Finite Automata).

### 2.4.1 Definisi Formal

**Finite Automaton** didefinisikan sebagai tuple $(Q, \Sigma, \delta, q_0, F)$ dimana:

- $Q$: Himpunan state (keadaan) yang terbatas
- $\Sigma$: Alphabet (himpunan simbol input)
- $\delta$: Fungsi transisi (transition function)
- $q_0$: Start state (state awal)
- $F$: Himpunan accept states (final states)

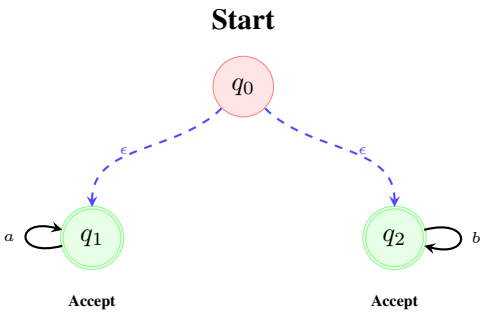
### 2.4.2 NFA (Nondeterministic Finite Automata)

NFA memiliki karakteristik:

- Untuk suatu state dan input symbol, dapat memiliki **nol, satu, atau lebih** transisi
- Dapat memiliki $\epsilon$**-transitions** (epsilon transitions) yang tidak mengonsumsi input
- Lebih mudah dikonstruksi dari regular expression
- Simulasi memerlukan backtracking atau multiple states tracking

Contoh NFA untuk pattern `a | b`:

Gambar 2.2 menunjukkan NFA untuk pattern `a | b` yang menggunakan $\epsilon$-transitions untuk branching.




Gambar 2.2: NFA untuk pattern `a | b` dengan $\epsilon$-transitions

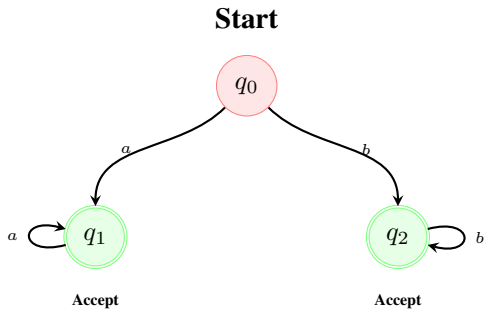
State $q_0$ adalah start state, $q_1$ dan $q_2$ adalah accept states. Dari $q_0$, dengan $\epsilon$-transition dapat menuju $q_1$ atau $q_2$, kemudian dari $q_1$ dapat menerima 'a', dan dari $q_2$ dapat menerima 'b'.

### 2.4.3 DFA (Deterministic Finite Automata)

DFA memiliki karakteristik:

- Untuk setiap state dan input symbol, terdapat **tepat satu** transisi
- Tidak memiliki $\epsilon$-transitions
- Lebih efisien untuk simulasi (deterministic)
- Setiap NFA dapat dikonversi menjadi DFA yang ekuivalen

Contoh DFA untuk pattern `a|b`:
Gambar 2.3 menunjukkan DFA yang ekuivalen dengan NFA di atas, tetapi deterministik.




Gambar 2.3: DFA untuk pattern `a|b` (deterministik)

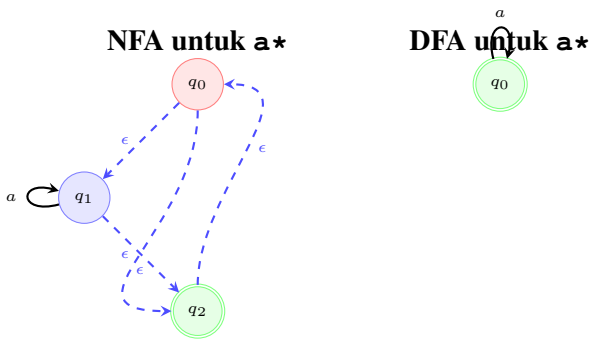
DFA ini deterministik: dari $q_0$, input 'a' selalu menuju $q_1$, input 'b' selalu menuju $q_2$. Tidak ada $\epsilon$-transitions dan setiap state memiliki tepat satu transisi untuk setiap input symbol.

### 2.4.4 Perbedaan NFA dan DFA

Perbedaan utama antara NFA dan DFA dapat dilihat pada Tabel 2.1 dan perbandingan visual pada Gambar 2.4.

| **Aspek** | **NFA** | **DFA** |
|---|---|---|
| Transisi per state | Bisa 0, 1, atau lebih | Tepat 1 |
| $\epsilon$-transitions | Diizinkan | Tidak diizinkan |
| Efisiensi simulasi | Perlu backtracking | Linear time |
| Jumlah states | Biasanya lebih sedikit | Bisa lebih banyak |
| Konstruksi dari regex | Lebih mudah | Lebih kompleks |

Tabel 2.1: Perbandingan NFA dan DFA




Gambar 2.4: Perbandingan visual NFA dan DFA untuk pattern `a*` (keduanya ekuivalen)

## 2.5 Konversi Regular Expression ke NFA: Algoritma Thompson

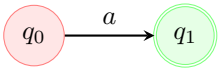
Algoritma Thompson adalah metode sistematis untuk mengkonversi regular expression menjadi $\epsilon$-NFA. Algoritma ini menggunakan pendekatan rekursif dengan template untuk setiap operasi regex.

### 2.5.1 Template Dasar

Algoritma Thompson menggunakan template untuk setiap operasi regex. Template-template berikut menunjukkan konstruksi dasar yang digunakan.

#### Literal (Karakter Tunggal)

Untuk regex `a`, NFA-nya adalah:




Gambar 2.5: Template NFA untuk literal `a`

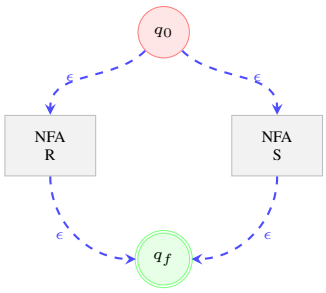
#### Concatenation (RS)

Untuk regex `RS`, NFA-nya dibangun dengan menghubungkan NFA untuk R dan S menggunakan $\epsilon$-transition:


Gambar 2.6: Template NFA untuk concatenation `RS`

**Union (R|S)**

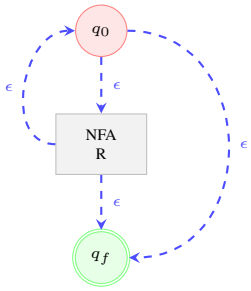
Untuk regex `R|S`, NFA-nya menggunakan $\epsilon$-transitions untuk branching:




Gambar 2.7: Template NFA untuk union `R|S`

**Kleene Star (R*)**

Untuk regex `R*`, NFA-nya memiliki loop dengan $\epsilon$-transitions:

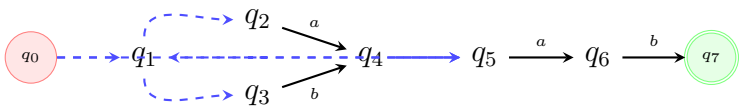



Gambar 2.8: Template NFA untuk Kleene star `R*`

### 2.5.2 Contoh: Konversi `(a|b)*abb`

Mari kita konstruksi NFA untuk regex `(a|b)*abb` menggunakan algoritma Thompson. Gambar 2.9 menunjukkan proses konstruksi langkah demi langkah.

1. **Literal 'a' dan 'b'**: Buat NFA untuk masing-masing
2. **Union (a|b)**: Gabungkan dengan $\epsilon$-transitions
3. **Kleene Star ((a|b)*)**: Tambahkan loop dengan $\epsilon$-transitions
4. **Concatenation dengan 'a'**: Tambahkan NFA untuk 'a'
5. **Concatenation dengan 'b'**: Tambahkan NFA untuk 'b' (dua kali)




Gambar 2.9: NFA untuk regex `(a|b)*abb` menggunakan algoritma Thompson

Hasil akhirnya adalah NFA yang dapat mengenali string seperti “abb”, “aabb”, “babb”, “ababb”, dll. NFA ini memiliki $\epsilon$-transitions yang memungkinkan multiple paths untuk input yang sama.

## 2.6 Konversi NFA ke DFA: Subset Construction

Karena NFA tidak deterministik dan simulasi NFA bisa tidak efisien, kita perlu mengkonversi NFA menjadi DFA yang ekuivalen menggunakan algoritma **subset construction**.

### 2.6.1 Konsep $\epsilon$-Closure

Sebelum subset construction, kita perlu memahami konsep **$\epsilon$-closure**:

- **$\epsilon$-closure** dari suatu state adalah himpunan semua state yang dapat dicapai dari state tersebut melalui $\epsilon$-transitions saja
- **$\epsilon$-closure** dari suatu set states adalah union dari $\epsilon$-closure setiap state dalam set tersebut

### 2.6.2 Algoritma Subset Construction

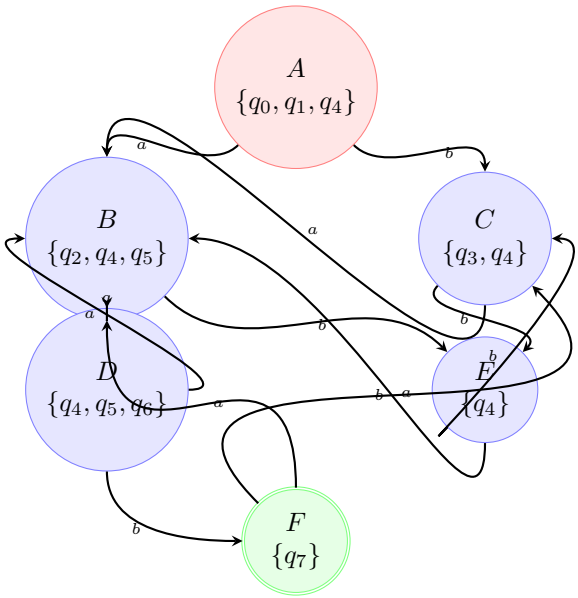
Algoritma subset construction bekerja sebagai berikut:

1. **Start State DFA**: $\epsilon$-closure dari start state NFA
2. **Untuk setiap state DFA dan setiap input symbol**:
   (a) Hitung semua NFA states yang dapat dicapai dengan input symbol tersebut
   (b) Ambil $\epsilon$-closure dari set states tersebut
   (c) Jika hasilnya belum ada sebagai state DFA, buat state baru
   (d) Tambahkan transisi dari state DFA saat ini ke state hasil
3. **Accept States DFA**: Setiap state DFA yang mengandung accept state NFA

### 2.6.3 Contoh: Konversi NFA `(a|b)*abb` ke DFA

Mari kita ikuti langkah-langkah subset construction. Gambar 2.10 menunjukkan proses konversi dan hasil DFA yang dihasilkan.

1. **Start State DFA**:
   - Mulai dari start state NFA, ambil $\epsilon$-closure
   - Misalkan hasilnya adalah set $\{q_0, q_1, q_4\} \rightarrow$ ini menjadi state DFA $A$
2. **Transisi dari State A dengan input 'a'**:
   - Dari semua NFA states dalam A, cari yang dapat menerima 'a'
   - Ambil $\epsilon$-closure dari hasilnya $\rightarrow$ misalkan $\{q_2, q_4, q_5\} \rightarrow$ state DFA $B$
3. **Transisi dari State A dengan input 'b'**:
   - Dari semua NFA states dalam A, cari yang dapat menerima 'b'
   - Ambil $\epsilon$-closure dari hasilnya $\rightarrow$ misalkan $\{q_3, q_4\} \rightarrow$ state DFA $C$
4. **Lanjutkan untuk state B dan C** dengan cara yang sama
5. **Accept States**: State DFA yang mengandung accept state NFA (misalnya state yang mengandung $q_7$)




Gambar 2.10: DFA hasil subset construction dari NFA `(a|b)*abb`

Hasilnya adalah DFA yang ekuivalen dengan NFA asli, tetapi deterministik dan lebih efisien untuk simulasi. Setiap state DFA merepresentasikan set states NFA yang dapat dicapai dengan input tertentu.

## 2.7 Implementasi NFA dan DFA dalam C/C++

Untuk memahami konsep secara praktis, kita akan melihat struktur data dan algoritma dasar untuk mengimplementasikan NFA dan DFA.

### 2.7.1 Struktur Data NFA

Listing 2.1: Struktur Data untuk NFA

```
#include <vector>
#include <set>
#include <map>

struct NFATransition {
    int from_state;
    char symbol;  // '\0' untuk epsilon transition
    int to_state;
};

class NFA {
private:
    int num_states;
    int start_state;
    std::set<int> accept_states;
    std::vector<NFATransition> transitions;

public:
    // Konstruktor
    NFA(int states, int start);

    // Menambahkan transisi
    void addTransition(int from, char symbol, int to);

    // Menghitung epsilon closure
    std::set<int> epsilonClosure(const std::set<int>& states);

    // Simulasi NFA
    bool simulate(const std::string& input);
};
```

### 2.7.2 Struktur Data DFA

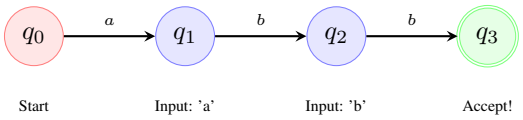
Listing 2.2: Struktur Data untuk DFA

```
class DFA {
private:
    int num_states;
    int start_state;
    std::set<int> accept_states;
    std::map<std::pair<int, char>, int> transition_table;

public:
```

```
    // Konstruktor
    DFA(int states, int start);

    // Menambahkan transisi (deterministic)
    void addTransition(int from, char symbol, int to);

    // Simulasi DFA (lebih sederhana dari NFA)
    bool simulate(const std::string& input);
};
```

### 2.7.3 Implementasi Simulasi DFA

Simulasi DFA lebih sederhana karena deterministik. Gambar 2.11 menunjukkan proses simulasi DFA untuk input string.




Gambar 2.11: Contoh simulasi DFA untuk input "abb"

Listing 2.3: Simulasi DFA

```
bool DFA::simulate(const std::string& input) {
    int current_state = start_state;

    for (char c : input) {
        auto key = std::make_pair(current_state, c);
        if (transition_table.find(key) == transition_table.end()) {
            return false;  // Tidak ada transisi
        }
        current_state = transition_table[key];
    }

    return accept_states.find(current_state) != accept_states.end();
}
```

### 2.7.4 Implementasi Subset Construction

Berikut adalah pseudocode untuk subset construction:

Listing 2.4: Subset Construction Algorithm

```
DFA NFA::toDFA() {
    DFA dfa(0, 0);
    std::map<std::set<int>, int> state_mapping;
    std::queue<std::set<int>> work_queue;

    // Start state DFA = epsilon closure dari start state NFA
    std::set<int> start_set = epsilonClosure({start_state});
```

```
    int dfa_start = dfa.addState();
    state_mapping[start_set] = dfa_start;
    work_queue.push(start_set);

    while (!work_queue.empty()) {
        std::set<int> nfa_states = work_queue.front();
        work_queue.pop();
        int dfa_state = state_mapping[nfa_states];

        // Untuk setiap input symbol
        for (char symbol : alphabet) {
            if (symbol == '\0') continue;  // Skip epsilon

            // Hitung move dengan symbol
            std::set<int> next_nfa_states;
            for (int state : nfa_states) {
                // Cari semua transisi dengan symbol ini
                for (auto& trans : transitions) {
                    if (trans.from_state == state &&
                        trans.symbol == symbol) {
                        next_nfa_states.insert(trans.to_state);
                    }
                }
            }

            // Ambil epsilon closure
            std::set<int> closure = epsilonClosure(next_nfa_states);

            if (!closure.empty()) {
                int next_dfa_state;
                if (state_mapping.find(closure) == state_mapping.end()) {
                    // State baru
                    next_dfa_state = dfa.addState();
                    state_mapping[closure] = next_dfa_state;
                    work_queue.push(closure);
                } else {
                    next_dfa_state = state_mapping[closure];
                }

                dfa.addTransition(dfa_state, symbol, next_dfa_state);
            }
        }
    }

    // Set accept states
    for (auto& pair : state_mapping) {
        for (int nfa_accept : accept_states) {
            if (pair.first.find(nfa_accept) != pair.first.end()) {
                dfa.setAcceptState(pair.second);
                break;
            }
        }
    }

```

```
62      return dfa;
63  }
```

## 2.8 Aplikasi dalam Lexical Analysis

Gambar 2.12 menunjukkan alur lengkap dari regular expression hingga token recognition dalam lexical analysis.


Gambar 2.12: Alur proses dari regular expression ke token scanner

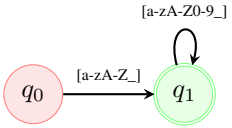
### 2.8.1 Token Recognition dengan DFA

Dalam lexical analysis, kita menggunakan DFA untuk mengenali token. Prosesnya:

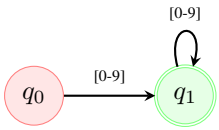
1. **Definisi Token**: Setiap jenis token didefinisikan dengan regular expression
2. **Kombinasi Regex**: Semua regex untuk token digabungkan dengan union
3. **Konversi ke DFA**: Regex gabungan dikonversi menjadi satu DFA
4. **Scanning**: Input dibaca karakter demi karakter, DFA dijalankan
5. **Longest Match**: Ambil token terpanjang yang cocok
6. **Token Classification**: Tentukan jenis token berdasarkan accept state yang dicapai

### 2.8.2 Contoh: Recognizer untuk Identifier dan Number

Mari kita buat recognizer sederhana untuk identifier dan number. Gambar 2.13 dan 2.14 menunjukkan DFA untuk masing-masing pattern.




Gambar 2.13: DFA untuk identifier: `[a-zA-Z_][a-zA-Z0-9_]*`




Gambar 2.14: DFA untuk number: `[0-9]+`

Listing 2.5: Token Recognizer menggunakan DFA

```
enum TokenType {
    IDENTIFIER,
    NUMBER,
    UNKNOWN
};

class TokenRecognizer {
private:
    DFA identifier_dfa;  // DFA untuk [a-zA-Z_][a-zA-Z0-9_]*
    DFA number_dfa;      // DFA untuk [0-9]+

public:
    TokenRecognizer() {
        // Konstruksi DFA untuk identifier dan number
        // (dari regex menggunakan Thompson + subset construction)
        buildIdentifierDFA();
        buildNumberDFA();
    }

    TokenType recognize(const std::string& lexeme) {
        if (identifier_dfa.simulate(lexeme)) {
            return IDENTIFIER;
        } else if (number_dfa.simulate(lexeme)) {
            return NUMBER;
        } else {
            return UNKNOWN;
        }
    }

private:
    void buildIdentifierDFA() {
        // Implementasi konstruksi DFA untuk identifier
        // Regex: [a-zA-Z_][a-zA-Z0-9_]*
    }

    void buildNumberDFA() {
        // Implementasi konstruksi DFA untuk number
        // Regex: [0-9]+
    }
};
```

Contoh penggunaan:

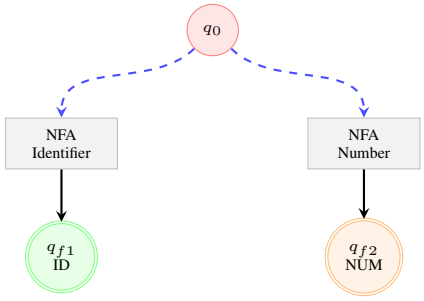
Listing 2.6: Contoh penggunaan TokenRecognizer

```
int main() {
```

```
    TokenRecognizer recognizer;

    std::vector<std::string> test_inputs = {
        "variable123",  // IDENTIFIER
        "42",           // NUMBER
        "_private",     // IDENTIFIER
        "3.14",         // UNKNOWN (belum support float)
        "123abc"        // UNKNOWN (mixed)
    };

    for (const auto& input : test_inputs) {
        TokenType type = recognizer.recognize(input);
        std::cout << input << " -> ";
        switch(type) {
            case IDENTIFIER: std::cout << "IDENTIFIER\n"; break;
            case NUMBER: std::cout << "NUMBER\n"; break;
            case UNKNOWN: std::cout << "UNKNOWN\n"; break;
        }
    }

    return 0;
}
```

### 2.8.3 Handling Multiple Tokens

Ketika kita memiliki multiple token types, kita perlu menggabungkan semua pattern menjadi satu DFA. Gambar 2.15 menunjukkan proses penggabungan NFA untuk multiple token types.




Gambar 2.15: Penggabungan NFA untuk multiple token types dengan labeling accept states

Proses handling multiple tokens:

1. Membuat NFA terpisah untuk setiap token type
2. Menggabungkan semua NFA dengan union, tetapi **label setiap accept state** dengan token type-nya
3. Konversi ke DFA (setiap DFA state mungkin mengandung multiple NFA accept states dengan label berbeda)

4. Saat scanning, jika mencapai accept state dengan multiple labels, gunakan **priority** atau **longest match**

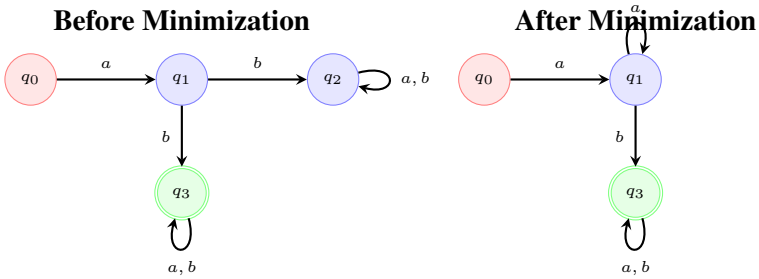
Contoh implementasi untuk multiple tokens:

Listing 2.7: Handling Multiple Token Types

```
class MultiTokenRecognizer {
private:
    DFA combined_dfa;
    std::map<int, TokenType> state_to_token;

public:
    TokenType recognize(const std::string& lexeme) {
        int final_state = combined_dfa.simulate(lexeme);
        if (final_state == -1) return UNKNOWN;

        // Jika state memiliki multiple labels, gunakan priority
        if (state_to_token.find(final_state) != state_to_token.end()) {
            return state_to_token[final_state];
        }
        return UNKNOWN;
    }

    // Longest match: scan sampai tidak bisa lanjut
    Token scanLongestMatch(std::istream& input) {
        std::string lexeme;
        int last_accept_state = -1;
        int last_accept_pos = -1;
        int current_state = start_state;
        int pos = 0;

        char c;
        while (input.get(c)) {
            lexeme += c;
            // Update state dengan input c
            // ...

            if (isAcceptState(current_state)) {
                last_accept_state = current_state;
                last_accept_pos = pos;
            }
            pos++;
        }

        // Kembalikan ke posisi terakhir yang accept
        input.seekg(last_accept_pos);
        return Token(state_to_token[last_accept_state],
                     lexeme.substr(0, last_accept_pos + 1));
    }
};
```

## 2.9 Optimasi: DFA Minimization

Setelah subset construction, DFA yang dihasilkan mungkin memiliki states yang redundan. Kita dapat meminimalkan DFA menggunakan algoritma seperti **Hopcroft's algorithm** atau **Moore's algorithm**. Gambar 2.16 menunjukkan contoh DFA sebelum dan sesudah minimisasi.




Gambar 2.16: Contoh DFA sebelum dan sesudah minimisasi (state $q_2$ dihapus karena equivalent dengan $q_1$)

### 2.9.1 Konsep State Equivalence

Dua states dalam DFA dikatakan **equivalent** jika:

- Keduanya accept states ATAU keduanya bukan accept states
- Untuk setiap input symbol, transisi dari kedua states menuju ke states yang equivalent

### 2.9.2 Algoritma Minimization

Algoritma minimisasi bekerja dengan:

1. Partisi states menjadi dua grup: accept states dan non-accept states
2. Untuk setiap partisi, periksa apakah states dalam partisi tersebut equivalent
3. Jika tidak equivalent, pisahkan menjadi partisi baru
4. Ulangi sampai tidak ada partisi yang dapat dipisah lagi
5. Merge states dalam partisi yang sama

DFA yang sudah diminimalkan memiliki jumlah states minimum yang masih ekuivalen dengan DFA asli.

## 2.10 Contoh Praktis: Implementasi Lengkap

Sebagai contoh praktis, berikut adalah implementasi lengkap DFA sederhana untuk mengenali identifier dan number:

Listing 2.8: Implementasi Lengkap DFA untuk Identifier dan Number

```
#include <iostream>
#include <string>
#include <map>
#include <set>

class SimpleDFA {
private:
    int start_state;
    std::set<int> accept_states;
    std::map<std::pair<int, char>, int> transitions;

    bool isLetter(char c) {
        return (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || c == '
↪ _';
    }

    bool isDigit(char c) {
        return c >= '0' && c <= '9';
    }

    bool isAlphanumeric(char c) {
        return isLetter(c) || isDigit(c);
    }

public:
    SimpleDFA() {
        // DFA untuk identifier: [a-zA-Z_][a-zA-Z0-9_]*
        start_state = 0;
        accept_states.insert(1);

        // State 0 -> State 1 dengan letter atau underscore
        for (char c = 'a'; c <= 'z'; c++) {
            transitions[{0, c}] = 1;
            transitions[{1, c}] = 1;  // Loop di state 1
        }
        for (char c = 'A'; c <= 'Z'; c++) {
            transitions[{0, c}] = 1;
            transitions[{1, c}] = 1;
        }
        transitions[{0, '_'}] = 1;
        transitions[{1, '_'}] = 1;

        // State 1 -> State 1 dengan digit
        for (char c = '0'; c <= '9'; c++) {
            transitions[{1, c}] = 1;
        }
    }

```

```
48      bool simulate(const std::string& input) {
49          int current_state = start_state;
50
51          for (char c : input) {
52              auto key = std::make_pair(current_state, c);
53              if (transitions.find(key) == transitions.end()) {
54                  return false;
55              }
56              current_state = transitions[key];
57          }
58
59          return accept_states.find(current_state) != accept_states.end();
60      }
61  };
62
63  int main() {
64      SimpleDFA dfa;
65
66      std::vector<std::string> test_cases = {
67          "variable",
68          "var123",
69          "_private",
70          "123var",       // Invalid (dimulai dengan digit)
71          "var_name",
72          "VarName123"
73      };
74
75      std::cout << "Testing DFA untuk Identifier:\n";
76      std::cout << "============================\n";
77      for (const auto& test : test_cases) {
78          bool result = dfa.simulate(test);
79          std::cout << "\"" << test << "\" -> "
80                    << (result ? "ACCEPT" : "REJECT") << "\n";
81      }
82
83      return 0;
84  }
```

Output program di atas:

```
Testing DFA untuk Identifier:
============================
"variable" -> ACCEPT
"var123" -> ACCEPT
"_private" -> ACCEPT
"123var" -> REJECT
"var_name" -> ACCEPT
"VarName123" -> ACCEPT
```

## 2.11 Kesimpulan

Dalam bab ini, kita telah mempelajari:

1. Regular expression adalah notasi formal untuk mendeskripsikan pola token
2. Finite automata (NFA dan DFA) adalah model matematika untuk mengenali regular language
3. Algoritma Thompson mengkonversi regular expression menjadi $\epsilon$-NFA
4. Subset construction mengkonversi NFA menjadi DFA yang ekuivalen
5. DFA lebih efisien untuk simulasi dan digunakan dalam lexical analysis
6. Implementasi praktis memerlukan struktur data yang tepat dan algoritma yang efisien

Pemahaman tentang regular expression dan finite automata ini menjadi dasar penting untuk implementasi lexical analyzer yang akan dipelajari dalam bab-bab selanjutnya.

## 2.12 Referensi dan Bahan Bacaan Lanjutan

Untuk memperdalam pemahaman tentang regular expression dan finite automata, mahasiswa disarankan membaca:

- **Dragon Book**: Aho, Lam, Sethi, & Ullman (2006). *Compilers: Principles, Techniques, and Tools* [1] - Bab 3: Lexical Analysis
- **Engineering a Compiler**: Cooper & Torczon (2011) [5] - Bab 2: Scanners
- **Aoyama Gakuin University**: Lecture notes tentang lexical analysis dan finite automata [7]
- **OpenGenus**: Tutorial tentang membangun lexer [8]
- **GeeksforGeeks**: Artikel tentang regular expression to NFA dan NFA to DFA conversion

# Bab 3

## Implementasi Lexer Sederhana (Hand-Written)

### 3.1 Tujuan Pembelajaran

Setelah mempelajari bab ini, mahasiswa diharapkan mampu:

1. Memahami konsep dan struktur hand-written lexer
2. Merancang state machine untuk token recognition
3. Mengimplementasikan lexer hand-written dalam C/C++ untuk subset bahasa C sesuai spesifikasi proyek (Bab 1, Bagian 1.9)
4. Menangani whitespace, komentar (single-line dan multi-line), dan escape sequences
5. Mengimplementasikan error handling untuk token tidak valid
6. Membuat unit test untuk berbagai kasus input

### 3.2 Pendahuluan

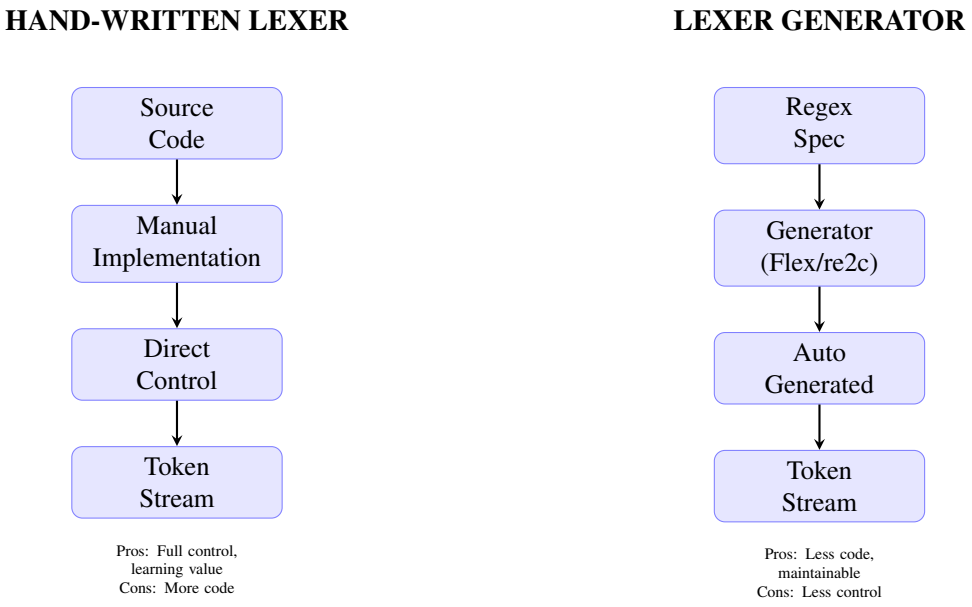
Pada bab ini kita mengimplementasikan **lexer hand-written** yang mengikuti **spesifikasi token proyek subset C** (Bab 1, Bagian 1.9). Token set (identifier, kata kunci `int`/`float`/`print`, literal, operator, punctuator) diseragamkan dengan spec agar contoh di bab ini konsisten dengan lexer proyek di Bab 4 (`simplec.l`) dan parser proyek di Bab 8.

Setelah memahami konsep lexical analysis secara teori pada bab sebelumnya, pada bab ini kita akan mengimplementasikan lexer secara praktis menggunakan pendekatan **hand-written** (ditulis manual). Menurut sumber terbuka:

> "Hand-written lexers are possible: directly code a state machine, or use manual scanning logic. Requires careful handling of edge cases (e.g. unclosed strings/comments)."[8]

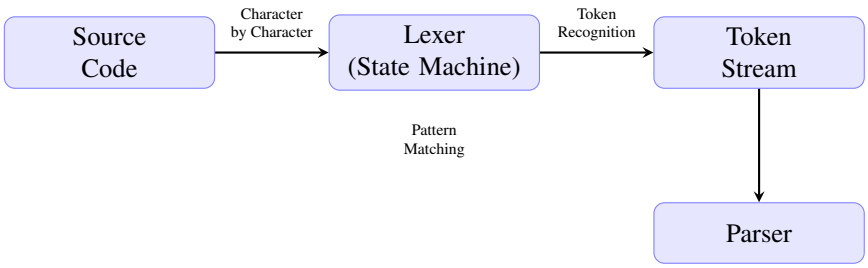
Pendekatan hand-written memberikan kontrol penuh terhadap implementasi dan sangat berguna untuk pembelajaran karena mahasiswa dapat memahami setiap detail proses tokenization. Meskipun lebih kompleks dibanding menggunakan generator seperti Flex atau re2c, hand-written lexer memberikan fleksibilitas dan pemahaman yang lebih dalam.

Gambar 3.1 menunjukkan perbandingan antara hand-written lexer dan lexer generator.




Gambar 3.1: Perbandingan hand-written lexer vs lexer generator

Gambar 3.2 menunjukkan alur umum proses tokenization dalam hand-written lexer.




Gambar 3.2: Alur umum proses tokenization dalam hand-written lexer

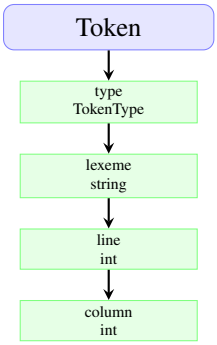
## 3.3 Struktur Token

Sebelum mengimplementasikan lexer, kita perlu mendefinisikan struktur data untuk merepresentasikan token. Token minimal harus menyimpan:

1. **Token Type**: Jenis token (identifier, keyword, number, operator, dll.)

2. **Lexeme**: String aktual yang di-match dari source code
3. **Position Information**: Baris dan kolom untuk error reporting
4. **Value** (opsional): Nilai numerik untuk number literals

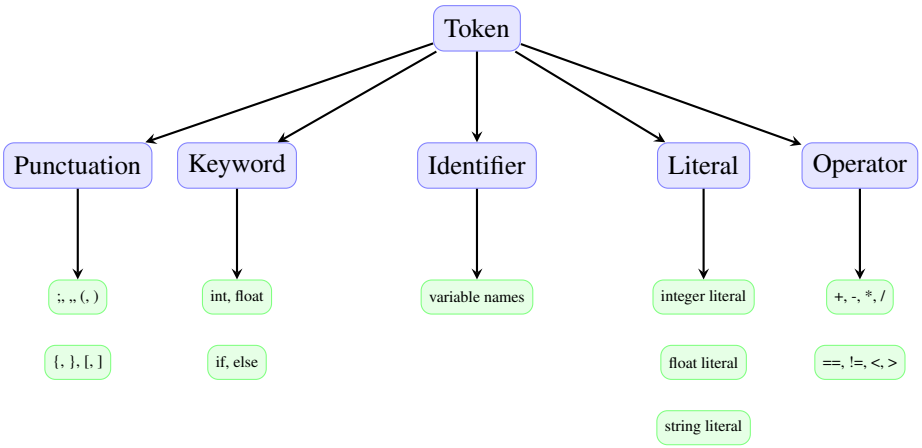
Gambar 3.3 menunjukkan struktur data token secara visual.




Gambar 3.3: Struktur data Token

### 3.3.1 Token Types

Token types dapat didefinisikan menggunakan enum. Berikut contoh untuk subset bahasa C:
Gambar 3.4 menunjukkan hierarki token types yang digunakan dalam lexer.




Gambar 3.4: Hierarki tipe token dalam lexer

Listing 3.1: Definisi Token Types

```
enum class TokenType {
    // Identifiers and Keywords
    IDENTIFIER,
    KEYWORD_INT, KEYWORD_FLOAT, KEYWORD_IF, KEYWORD_ELSE,
    KEYWORD_WHILE, KEYWORD_FOR, KEYWORD_RETURN,
```

```
6
7       // Literals
8       INTEGER_LITERAL,
9       FLOAT_LITERAL,
10      STRING_LITERAL,
11      CHAR_LITERAL,
12
13      // Operators
14      OP_PLUS, OP_MINUS, OP_MULTIPLY, OP_DIVIDE,
15      OP_ASSIGN, OP_EQUAL, OP_NOT_EQUAL,
16      OP_LESS, OP_LESS_EQUAL, OP_GREATER, OP_GREATER_EQUAL,
17      OP_AND, OP_OR, OP_NOT,
18
19      // Punctuation
20      SEMICOLON, COMMA, DOT,
21      LPAREN, RPAREN,     // ( )
22      LBRACE, RBRACE,     // { }
23      LBRACKET, RBRACKET, // [ ]
24
25      // Special
26      END_OF_FILE,
27      INVALID
28  };
```

### 3.3.2 Token Structure

Struktur token dalam C++ dapat didefinisikan sebagai berikut:

Listing 3.2: Struktur Token

```
1   struct Token {
2       TokenType type;
3       std::string lexeme;
4       int line;
5       int column;
6       union {
7           int intValue;       // Untuk INTEGER_LITERAL
8           double floatValue; // Untuk FLOAT_LITERAL
9       };
10
11      Token(TokenType t, const std::string& lex, int l, int c)
12          : type(t), lexeme(lex), line(l), column(c) {}
13  };
```

## 3.4 Finite State Machine untuk Lexer

Lexical analysis secara fundamental adalah proses pattern matching yang dapat dimodelkan menggunakan **Finite State Machine (FSM)** atau **Finite Automata**. Menurut sumber dari Aoyama Gakuin University:

> "Lexical analysis breaks input text into lexemes which correspond to tokens.

> Usually implemented using regular languages → regex → NFA → DFA → (minimized) DFA for efficiency."[7]

Dalam implementasi hand-written, kita tidak perlu membuat DFA secara eksplisit, tetapi kita menggunakan logika state machine dalam kode.

### 3.4.1 State Machine Design

State machine untuk lexer sederhana dapat memiliki state-state berikut:

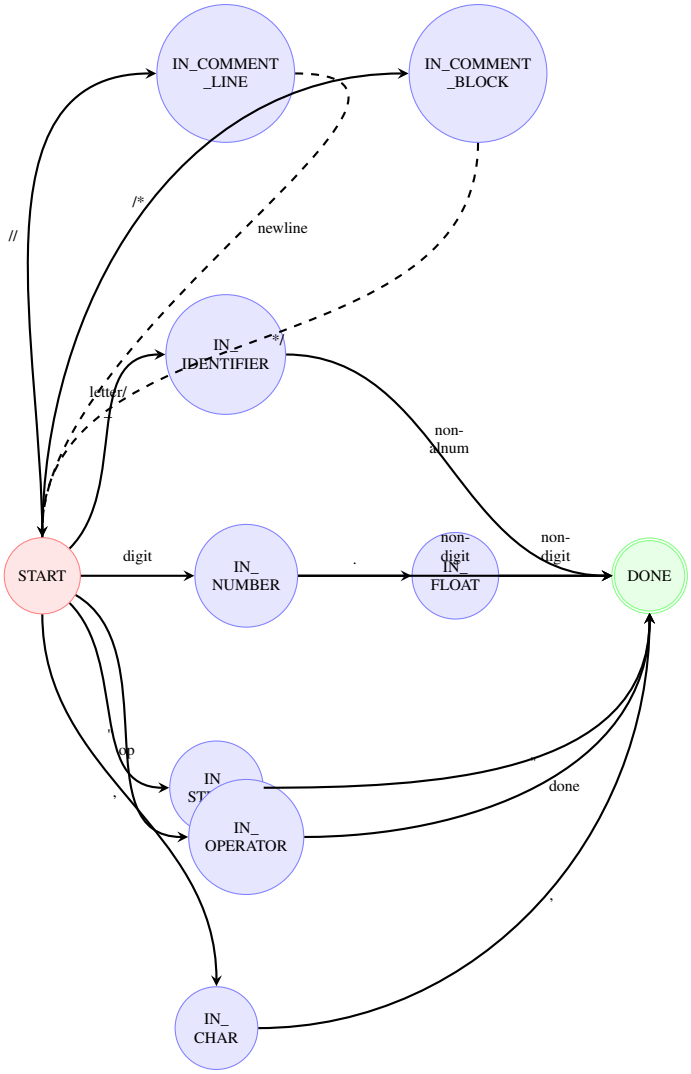
- **START**: State awal, menunggu karakter pertama dari token
- **IN_IDENTIFIER**: Sedang membaca identifier atau keyword
- **IN_NUMBER**: Sedang membaca angka (integer atau float)
- **IN_FLOAT**: Setelah menemukan titik desimal
- **IN_STRING**: Sedang membaca string literal
- **IN_CHAR**: Sedang membaca character literal
- **IN_COMMENT_LINE**: Sedang membaca single-line comment
- **IN_COMMENT_BLOCK**: Sedang membaca multi-line comment
- **IN_OPERATOR**: Sedang membaca operator (mungkin multi-character)
- **DONE**: Token selesai dibaca

Gambar 3.5 menunjukkan state machine untuk lexer sederhana.

### 3.4.2 State Transitions

Transisi state terjadi berdasarkan karakter yang dibaca:

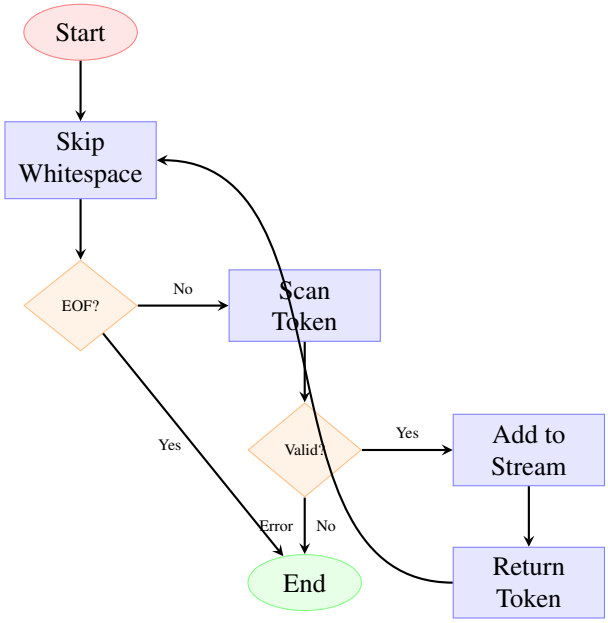
1. **START** → **IN_IDENTIFIER**: Jika karakter adalah huruf atau underscore
2. **START** → **IN_NUMBER**: Jika karakter adalah digit
3. **START** → **IN_STRING**: Jika karakter adalah double quote (`"`)
4. **START** → **IN_CHAR**: Jika karakter adalah single quote (`'`)
5. **START** → **IN_COMMENT_LINE**: Jika menemukan `//`
6. **START** → **IN_COMMENT_BLOCK**: Jika menemukan `/*`




Gambar 3.5: State machine untuk hand-written lexer

7. **START** → **IN_OPERATOR**: Jika karakter adalah operator
8. **IN_NUMBER** → **IN_FLOAT**: Jika menemukan titik desimal
9. **IN_IDENTIFIER** → **DONE**: Jika karakter bukan alphanumeric atau underscore
10. **IN_NUMBER** → **DONE**: Jika karakter bukan digit atau titik
11. **IN_STRING** → **DONE**: Jika menemukan closing quote (dengan handling escape)

Gambar 3.6 menunjukkan flowchart proses tokenization.




Gambar 3.6: Flowchart proses tokenization

## 3.5 Implementasi Lexer dalam C++

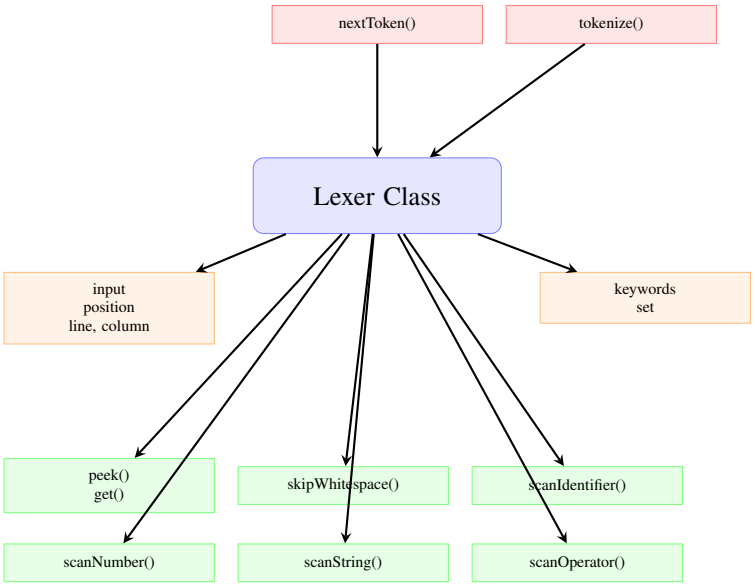
Berikut adalah implementasi lengkap lexer sederhana untuk subset bahasa C:

### 3.5.1 Kelas Lexer

Gambar 3.7 menunjukkan arsitektur kelas Lexer dan komponen-komponennya.

Listing 3.3: Header File: lexer.h

```
#ifndef LEXER_H
#define LEXER_H

#include <string>
#include <unordered_set>
#include <vector>
```




Gambar 3.7: Arsitektur kelas Lexer

```
7
8  enum class TokenType {
9      IDENTIFIER,
10     KEYWORD_INT, KEYWORD_FLOAT, KEYWORD_IF, KEYWORD_ELSE,
11     KEYWORD_WHILE, KEYWORD_FOR, KEYWORD_RETURN,
12     INTEGER_LITERAL, FLOAT_LITERAL,
13     STRING_LITERAL, CHAR_LITERAL,
14     OP_PLUS, OP_MINUS, OP_MULTIPLY, OP_DIVIDE,
15     OP_ASSIGN, OP_EQUAL, OP_NOT_EQUAL,
16     OP_LESS, OP_LESS_EQUAL, OP_GREATER, OP_GREATER_EQUAL,
17     OP_AND, OP_OR, OP_NOT,
18     SEMICOLON, COMMA, DOT,
19     LPAREN, RPAREN, LBRACE, RBRACE,
20     LBRACKET, RBRACKET,
21     END_OF_FILE, INVALID
22 };
23
24 struct Token {
25     TokenType type;
26     std::string lexeme;
27     int line;
28     int column;
29
30     Token(TokenType t, const std::string& lex, int l, int c)
31         : type(t), lexeme(lex), line(l), column(c) {}
32 };
33
34 class Lexer {
35 private:
36     std::string input;
37     size_t position;
38     int line;
39     int column;
```

```
40      std::unordered_set<std::string> keywords;
41
42      char peek() const;
43      char get();
44      void skipWhitespace();
45      void skipLineComment();
46      void skipBlockComment();
47      Token scanIdentifier();
48      Token scanNumber();
49      Token scanString();
50      Token scanChar();
51      Token scanOperator();
52      TokenType getKeywordType(const std::string& lexeme) const;
53
54  public:
55      Lexer(const std::string& source);
56      Token nextToken();
57      std::vector<Token> tokenize();
58  };
59
60  #endif
```

### 3.5.2 Implementasi Lexer

Listing 3.4: Implementasi: lexer.cpp (Bagian 1)

```
1   #include "lexer.h"
2   #include <cctype>
3   #include <stdexcept>
4
5   Lexer::Lexer(const std::string& source)
6       : input(source), position(0), line(1), column(1) {
7       // Initialize keywords
8       keywords = {"int", "float", "if", "else",
9                   "while", "for", "return"};
10  }
11
12  char Lexer::peek() const {
13      if (position >= input.length()) {
14          return '\0';
15      }
16      return input[position];
17  }
18
19  char Lexer::get() {
20      if (position >= input.length()) {
21          return '\0';
22      }
23      char c = input[position++];
24      if (c == '\n') {
25          line++;
26          column = 1;
27      } else {
```

```
28          column++;
29      }
30      return c;
31  }
```

### 3.5.3 Handling Whitespace dan Komentar

Listing 3.5: Implementasi: lexer.cpp (Bagian 2 - Whitespace dan Comments)

```
1  void Lexer::skipWhitespace() {
2      while (position < input.length()) {
3          char c = peek();
4          if (std::isspace(c)) {
5              get();
6          } else if (c == '/' && position + 1 < input.length()
7                      && input[position + 1] == '/') {
8              skipLineComment();
9          } else if (c == '/' && position + 1 < input.length()
10                     && input[position + 1] == '*') {
11             skipBlockComment();
12         } else {
13             break;
14         }
15     }
16 }
17
18 void Lexer::skipLineComment() {
19     // Skip "//"
20     get(); get();
21     // Skip until newline or EOF
22     while (peek() != '\n' && peek() != '\0') {
23         get();
24     }
25 }
26
27 void Lexer::skipBlockComment() {
28     // Skip "/*"
29     get(); get();
30     while (position < input.length()) {
31         if (peek() == '*' && position + 1 < input.length()
32             && input[position + 1] == '/') {
33             get(); get(); // Skip "*/"
34             return;
35         }
36         get();
37     }
38     // Error: unclosed comment
39     throw std::runtime_error("Unclosed block comment at line "
40                              + std::to_string(line));
41 }
```

### 3.5.4 Scanning Identifier dan Keyword

Listing 3.6: Implementasi: lexer.cpp (Bagian 3 - Identifier)

```
Token Lexer::scanIdentifier() {
    int startLine = line;
    int startCol = column;
    std::string lexeme;

    // First character must be letter or underscore
    if (std::isalpha(peek()) || peek() == '_') {
        lexeme += get();
    }

    // Subsequent characters can be alphanumeric or underscore
    while (std::isalnum(peek()) || peek() == '_') {
        lexeme += get();
    }

    // Check if it's a keyword
    TokenType type = getKeywordType(lexeme);
    if (type != TokenType::IDENTIFIER) {
        return Token(type, lexeme, startLine, startCol);
    }

    return Token(TokenType::IDENTIFIER, lexeme, startLine, startCol);
}

TokenType Lexer::getKeywordType(const std::string& lexeme) const {
    if (lexeme == "int") return TokenType::KEYWORD_INT;
    if (lexeme == "float") return TokenType::KEYWORD_FLOAT;
    if (lexeme == "if") return TokenType::KEYWORD_IF;
    if (lexeme == "else") return TokenType::KEYWORD_ELSE;
    if (lexeme == "while") return TokenType::KEYWORD_WHILE;
    if (lexeme == "for") return TokenType::KEYWORD_FOR;
    if (lexeme == "return") return TokenType::KEYWORD_RETURN;
    return TokenType::IDENTIFIER;
}
```

### 3.5.5 Scanning Number Literals

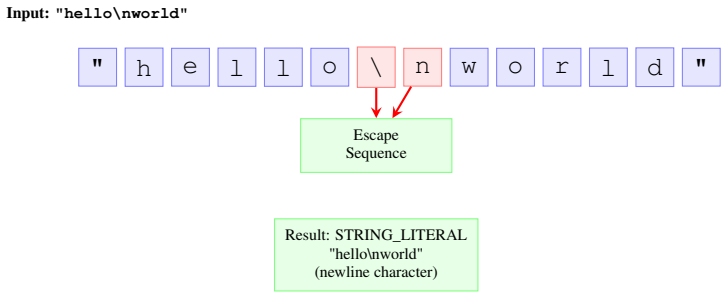
Listing 3.7: Implementasi: lexer.cpp (Bagian 4 - Numbers)

```
Token Lexer::scanNumber() {
    int startLine = line;
    int startCol = column;
    std::string lexeme;
    bool isFloat = false;

    // Read integer part
    while (std::isdigit(peek())) {
        lexeme += get();
    }

```

```
    // Check for decimal point
    if (peek() == '.') {
        lexeme += get();
        isFloat = true;

        // Read fractional part
        while (std::isdigit(peek())) {
            lexeme += get();
        }
    }

    // Check for exponent (optional, for future enhancement)
    if (peek() == 'e' || peek() == 'E') {
        lexeme += get();
        if (peek() == '+' || peek() == '-') {
            lexeme += get();
        }
        while (std::isdigit(peek())) {
            lexeme += get();
        }
        isFloat = true;
    }

    TokenType type = isFloat ? TokenType::FLOAT_LITERAL
                             : TokenType::INTEGER_LITERAL;
    return Token(type, lexeme, startLine, startCol);
}
```

### 3.5.6 Scanning String dan Character Literals

Gambar 3.8 menunjukkan contoh handling escape sequences dalam string literal.




Gambar 3.8: Handling escape sequences dalam string literal

Listing 3.8: Implementasi: lexer.cpp (Bagian 5 - Strings)

```
Token Lexer::scanString() {
    int startLine = line;
    int startCol = column;
    std::string lexeme;

    // Consume opening quote
```

```
    get(); // Skip opening "

    while (peek() != '"' && peek() != '\0') {
        if (peek() == '\\') {
            // Handle escape sequences
            get(); // Skip backslash
            char escaped = get();
            switch (escaped) {
                case 'n': lexeme += '\n'; break;
                case 't': lexeme += '\t'; break;
                case 'r': lexeme += '\r'; break;
                case '\\': lexeme += '\\'; break;
                case '"': lexeme += '"'; break;
                default: lexeme += '\\'; lexeme += escaped; break;
            }
        } else {
            lexeme += get();
        }
    }

    if (peek() == '\0') {
        // Unclosed string
        return Token(TokenType::INVALID, lexeme, startLine, startCol);
    }

    get(); // Consume closing "
    return Token(TokenType::STRING_LITERAL, lexeme, startLine, startCol);
}

Token Lexer::scanChar() {
    int startLine = line;
    int startCol = column;
    std::string lexeme;

    get(); // Skip opening '

    if (peek() == '\\') {
        // Escape sequence
        get(); // Skip backslash
        lexeme += get();
    } else {
        lexeme += get();
    }

    if (peek() != '\'') {
        return Token(TokenType::INVALID, lexeme, startLine, startCol);
    }

    get(); // Consume closing '
    return Token(TokenType::CHAR_LITERAL, lexeme, startLine, startCol);
}
```

### 3.5.7 Scanning Operators

Listing 3.9: Implementasi: lexer.cpp (Bagian 6 - Operators)

```
Token Lexer::scanOperator() {
    int startLine = line;
    int startCol = column;
    char first = get();
    std::string lexeme(1, first);

    // Check for multi-character operators
    char next = peek();

    switch (first) {
        case '=':
            if (next == '=') {
                lexeme += get();
                return Token(TokenType::OP_EQUAL, lexeme, startLine,
↪ startCol);
            }
            return Token(TokenType::OP_ASSIGN, lexeme, startLine,
↪ startCol);

        case '!':
            if (next == '=') {
                lexeme += get();
                return Token(TokenType::OP_NOT_EQUAL, lexeme, startLine,
↪ startCol);
            }
            return Token(TokenType::OP_NOT, lexeme, startLine, startCol);

        case '<':
            if (next == '=') {
                lexeme += get();
                return Token(TokenType::OP_LESS_EQUAL, lexeme, startLine,
↪  startCol);
            }
            return Token(TokenType::OP_LESS, lexeme, startLine, startCol)
↪ ;

        case '>':
            if (next == '=') {
                lexeme += get();
                return Token(TokenType::OP_GREATER_EQUAL, lexeme,
↪ startLine, startCol);
            }
            return Token(TokenType::OP_GREATER, lexeme, startLine,
↪ startCol);

        case '&':
            if (next == '&') {
                lexeme += get();
                return Token(TokenType::OP_AND, lexeme, startLine,
↪ startCol);
            }
```

```
            return Token(TokenType::INVALID, lexeme, startLine, startCol)
↪ ;

        case '|':
            if (next == '|') {
                lexeme += get();
                return Token(TokenType::OP_OR, lexeme, startLine,
↪ startCol);
            }
            return Token(TokenType::INVALID, lexeme, startLine, startCol)
↪ ;

        case '+':
            return Token(TokenType::OP_PLUS, lexeme, startLine, startCol)
↪ ;
        case '-':
            return Token(TokenType::OP_MINUS, lexeme, startLine, startCol
↪ );
        case '*':
            return Token(TokenType::OP_MULTIPLY, lexeme, startLine,
↪ startCol);
        case '/':
            return Token(TokenType::OP_DIVIDE, lexeme, startLine,
↪ startCol);

        default:
            return Token(TokenType::INVALID, lexeme, startLine, startCol)
↪ ;
    }
}
```

### 3.5.8 Main Tokenization Function

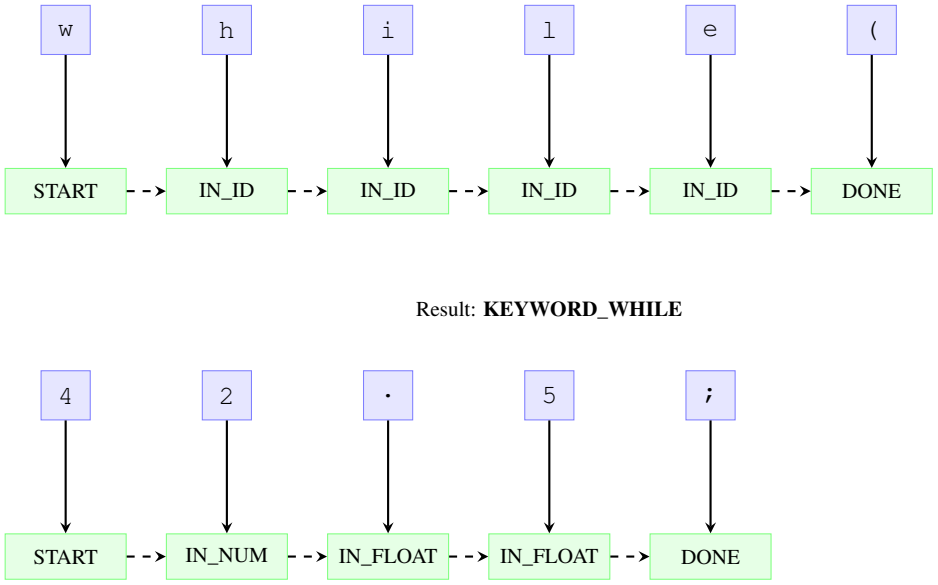
Listing 3.10: Implementasi: lexer.cpp (Bagian 7 - Main Function)

```
Token Lexer::nextToken() {
    skipWhitespace();

    if (position >= input.length()) {
        return Token(TokenType::END_OF_FILE, "", line, column);
    }

    char c = peek();

    // Identifier or keyword
    if (std::isalpha(c) || c == '_') {
        return scanIdentifier();
    }

    // Number
    if (std::isdigit(c)) {
        return scanNumber();
    }
```

```
19
20      // String literal
21      if (c == '"') {
22          return scanString();
23      }
24
25      // Character literal
26      if (c == '\'') {
27          return scanChar();
28      }
29
30      // Operators and punctuation
31      if (c == '+' || c == '-' || c == '*' || c == '/' ||
32          c == '=' || c == '!' || c == '<' || c == '>' ||
33          c == '&' || c == '|') {
34          return scanOperator();
35      }
36
37      // Punctuation
38      if (c == ';') {
39          get();
40          return Token(TokenType::SEMICOLON, ";", line, column - 1);
41      }
42      if (c == ',') {
43          get();
44          return Token(TokenType::COMMA, ",", line, column - 1);
45      }
46      if (c == '.') {
47          get();
48          return Token(TokenType::DOT, ".", line, column - 1);
49      }
50      if (c == '(') {
51          get();
52          return Token(TokenType::LPAREN, "(", line, column - 1);
53      }
54      if (c == ')') {
55          get();
56          return Token(TokenType::RPAREN, ")", line, column - 1);
57      }
58      if (c == '{') {
59          get();
60          return Token(TokenType::LBRACE, "{", line, column - 1);
61      }
62      if (c == '}') {
63          get();
64          return Token(TokenType::RBRACE, "}", line, column - 1);
65      }
66      if (c == '[') {
67          get();
68          return Token(TokenType::LBRACKET, "[", line, column - 1);
69      }
70      if (c == ']') {
71          get();
72          return Token(TokenType::RBRACKET, "]", line, column - 1);
```

```
    }

    // Unknown character
    get();
    return Token(TokenType::INVALID, std::string(1, c), line, column - 1)
↪ ;
}

std::vector<Token> Lexer::tokenize() {
    std::vector<Token> tokens;
    Token token = nextToken();
    while (token.type != TokenType::END_OF_FILE) {
        tokens.push_back(token);
        token = nextToken();
    }
    tokens.push_back(token); // Add EOF token
    return tokens;
}
```

Gambar 3.9 menunjukkan proses scanning untuk identifier dan keyword.

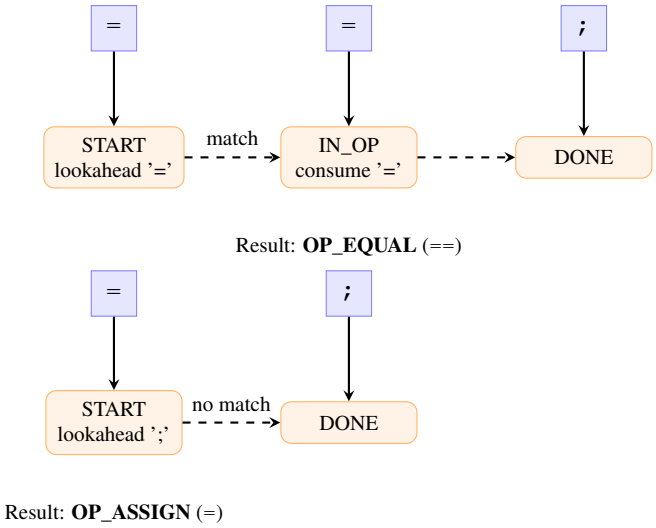



Gambar 3.9: Proses scanning keyword/identifier dan literal numerik pada lexer

Gambar 3.10 menunjukkan proses scanning untuk operator multi-character.

## 3.6 Error Handling

Error handling dalam lexer harus menangani berbagai kasus edge case:




Gambar 3.10: Proses scanning operator: perbandingan == dan =

### 3.6.1 Unclosed Strings dan Comments

- **Unclosed String**: Jika string literal tidak ditutup sebelum EOF, lexer harus mengembalikan token INVALID dengan informasi posisi yang tepat.
- **Unclosed Block Comment**: Jika komentar blok tidak ditutup, dapat di-handle dengan exception atau mengembalikan error token.

### 3.6.2 Invalid Characters

Karakter yang tidak valid (tidak termasuk dalam kategori token manapun) harus dikembalikan sebagai token INVALID dengan informasi posisi untuk error reporting yang baik.

Tabel 3.1 menunjukkan contoh-contoh token yang valid dan tidak valid.

| Input | Token Type | Keterangan |
|---|---|---|
| `int` | KEYWORD_INT | Keyword valid |
| `hello` | IDENTIFIER | Identifier valid |
| `42` | INTEGER_LITERAL | Integer valid |
| `3.14` | FLOAT_LITERAL | Float valid |
| `"hello"` | STRING_LITERAL | String valid |
| `==` | OP_EQUAL | Operator multi-character |
| `=` | OP_ASSIGN | Operator single-character |
| | INVALID | Karakter tidak valid |
| `"unclosed` | INVALID | String tidak tertutup |
| `/* comment` | Error | Comment tidak tertutup |

Tabel 3.1: Contoh token valid dan tidak valid

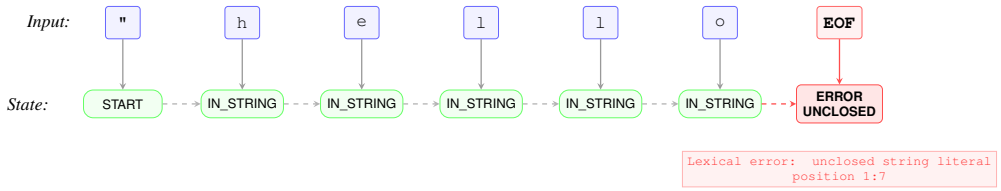
### 3.6.3 Malformed Numbers

Contoh kasus malformed:

- `123.` (titik tanpa digit setelahnya)
- `.456` (titik tanpa digit sebelumnya) - dapat di-handle sebagai valid float
- `12.34.56` (multiple decimal points)

Implementasi dapat memilih untuk menerima atau menolak format tertentu sesuai kebutuhan.

Gambar 3.11 menunjukkan contoh error handling untuk unclosed string.




Gambar 3.11: Transisi state lexer dan penanganan kesalahan pada string literal yang tidak tertutup

## 3.7 Testing Lexer

Unit testing sangat penting untuk memastikan lexer bekerja dengan benar. Berikut contoh test cases:

### 3.7.1 Test Cases untuk Identifier dan Keyword

Listing 3.11: Test Cases: Identifiers dan Keywords

```
void testIdentifiers() {
    Lexer lexer("int x = 42;");
    Token t1 = lexer.nextToken(); // Should be KEYWORD_INT
    Token t2 = lexer.nextToken(); // Should be IDENTIFIER "x"
    Token t3 = lexer.nextToken(); // Should be OP_ASSIGN
    // ...
}
```

### 3.7.2 Test Cases untuk Numbers

- `42` → INTEGER_LITERAL
- `3.14` → FLOAT_LITERAL

- `123.456` → FLOAT_LITERAL
- `0` → INTEGER_LITERAL

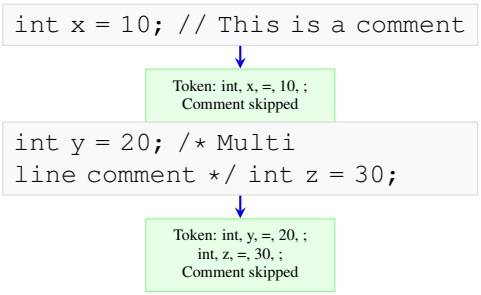
### 3.7.3 Test Cases untuk Strings

- `"hello"` → STRING_LITERAL dengan value "hello"
- `"hello\nworld"` → STRING_LITERAL dengan escape sequence
- `"unclosed` → INVALID (unclosed string)

### 3.7.4 Test Cases untuk Comments

- `// single line comment` → Di-skip, tidak menghasilkan token
- `/* multi-line comment */` → Di-skip
- `/* unclosed comment` → Error atau exception

Gambar 3.12 menunjukkan proses handling komentar dalam lexer.




Gambar 3.12: Handling komentar dalam lexer

## 3.8 Contoh Penggunaan

Berikut contoh penggunaan lexer untuk tokenize source code sederhana:

Listing 3.12: Contoh Penggunaan Lexer

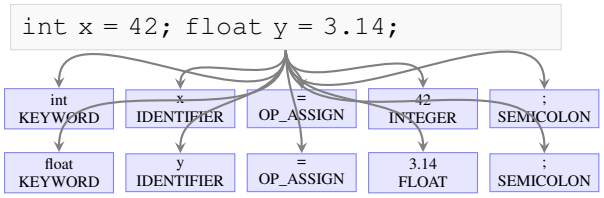
```
#include "lexer.h"
#include <iostream>

int main() {
    std::string source = R"(
        int x = 42;
        float y = 3.14;
        if (x > 10) {
            return y;
```

```
        }
    )";

    Lexer lexer(source);
    std::vector<Token> tokens = lexer.tokenize();

    for (const auto& token : tokens) {
        std::cout << "Token: " << token.lexeme
                  << " Type: " << static_cast<int>(token.type)
                  << " Line: " << token.line
                  << " Column: " << token.column << std::endl;
    }

    return 0;
}
```

Output yang diharapkan:

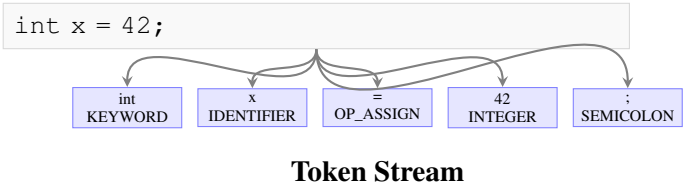
```
Token: int Type: 1 Line: 2 Column: 9
Token: x Type: 0 Line: 2 Column: 13
Token: = Type: 13 Line: 2 Column: 15
Token: 42 Type: 8 Line: 2 Column: 17
Token: ; Type: 20 Line: 2 Column: 19
...
```

Gambar 3.13 menunjukkan contoh lengkap tokenization untuk program sederhana.



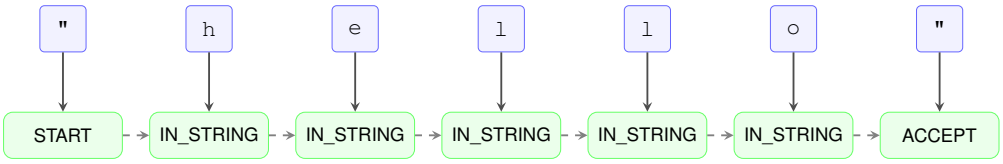

Gambar 3.13: Contoh lengkap tokenization untuk program sederhana

Gambar 3.14 menunjukkan visualisasi token stream untuk contoh kode sederhana.

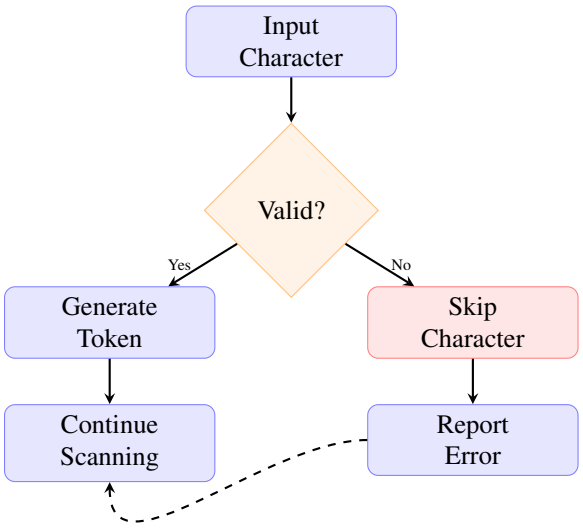



Gambar 3.14: Contoh tokenization: `int x = 42;` menjadi token stream

Gambar 3.15 menunjukkan proses scanning secara detail untuk string `"hello"`.
Gambar 3.16 menunjukkan strategi error recovery dalam lexer.




Gambar 3.15: Representasi formal proses analisis leksikal pada string literal `"hello"`




Gambar 3.16: Strategi error recovery dalam lexer

## 3.9 Best Practices

Beberapa best practices dalam implementasi hand-written lexer:

1. **Separation of Concerns**: Pisahkan logika untuk setiap jenis token ke fungsi terpisah
2. **Position Tracking**: Selalu track line dan column untuk error reporting yang baik
3. **Lookahead**: Gunakan `peek()` untuk lookahead tanpa mengkonsumsi karakter
4. **Error Recovery**: Rancang strategi error recovery (misalnya skip invalid character dan lanjut)
5. **Testing**: Buat comprehensive test suite untuk semua edge cases
6. **Documentation**: Dokumentasikan token types dan format yang didukung

## 3.10 Kesimpulan

Dalam bab ini, kita telah mempelajari:

1. Struktur token dan token types untuk subset bahasa C
2. Konsep finite state machine dalam konteks lexical analysis
3. Implementasi hand-written lexer dalam C++ dengan handling:
    - Identifier dan keyword recognition
    - Number literals (integer dan float)
    - String dan character literals dengan escape sequences
    - Operators (single dan multi-character)
    - Whitespace dan komentar (single-line dan multi-line)
4. Error handling untuk edge cases
5. Testing strategies untuk lexer

Implementasi hand-written lexer memberikan pemahaman mendalam tentang proses tokenization dan menjadi dasar untuk memahami bagaimana lexer generator seperti Flex bekerja di belakang layar. Lexer yang dibahas di bab ini mengimplementasikan spesifikasi token proyek subset C (Bab 1); lexer proyek dengan Flex dibangun di Bab 4 (`simplec.l`).

## 3.11 Referensi dan Bahan Bacaan Lanjutan

Untuk memperdalam pemahaman tentang implementasi lexer, mahasiswa disarankan membaca:

- **Dragon Book**: Aho, Lam, Sethi, & Ullman (2006). *Compilers: Principles, Techniques, and Tools* [1] - Bab 3: Lexical Analysis
- **Engineering a Compiler**: Cooper & Torczon (2011) [5] - Bab 2: Scanning
- **OpenGenus - Build Lexer**: Tutorial tentang hand-written lexer [8]
- **Aoyama Gakuin University**: Lecture notes tentang lexical analysis [7]
- **GeeksforGeeks**: Contoh implementasi lexical analyzer dalam C++ [1]
- **Programming Notes**: Tutorial tentang simple lexer menggunakan finite state machine [2]

[1] https://www.geeksforgeeks.org/cpp/lexical-analyzer-in-cpp/
[2] https://www.programmingnotes.org/4699/cpp-simple-lexer-using-a-finite-state-machine/

# Bab 4

## Lexer Generator (Flex/re2c) dan Praktikum Lexer

### 4.1 Tujuan Pembelajaran

Setelah mempelajari bab ini, mahasiswa diharapkan mampu:

1. Memahami konsep dan keuntungan menggunakan lexer generator
2. Menggunakan Flex untuk membuat specification file (.l) dan generate lexer code
3. Menggunakan re2c untuk membuat lexer dengan embedded specification
4. Membuat specification file untuk lexer bahasa sederhana
5. Mengintegrasikan generated lexer dengan program utama
6. Membandingkan hand-written lexer dengan generator-based lexer
7. Mengevaluasi trade-off antara performa, kemudahan maintenance, dan fleksibilitas

### 4.2 Pendahuluan

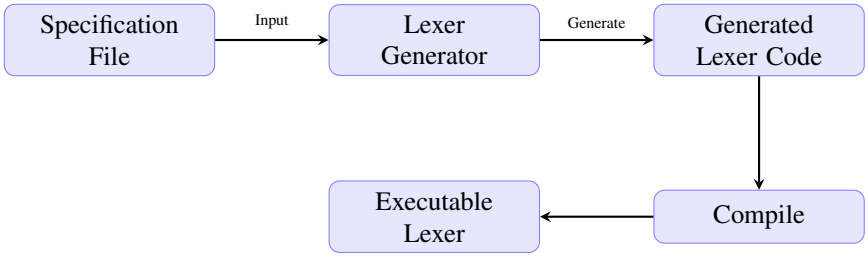
Pada bab ini kita menambah **lexer proyek compiler subset C** menggunakan Flex. Spesifikasi token proyek telah didefinisikan di Bab 1 (Bagian 1.9); file `simplec.l` yang dibangun di bab ini (dan dipakai bersama parser di Bab 8) mengimplementasikan token set tersebut. Contoh lengkap `simplec.l` terintegrasi dengan parser `simplec.y` di Bab 8; token yang dihasilkan mengikuti spesifikasi Bab 1. Kode hasil generate menjadi bagian dari codebase proyek di folder `proyek-compiler-subset-c/`.

Pada bab sebelumnya, kita telah mempelajari implementasi hand-written lexer. Meskipun pendekatan tersebut memberikan kontrol penuh dan pemahaman mendalam, dalam praktik industri, penggunaan **lexer generator** lebih umum karena efisiensi dan kemudahan maintenance. Menurut sumber terbuka:

> "re2c is a high-performance lexer generator for C/C++ that takes regex specifications and builds deterministic finite automata. It's used in real projects."[9]

Lexer generator adalah tools yang menerima specification file (berisi pattern dan action) dan menghasilkan kode lexer yang siap digunakan. Dua generator populer untuk C/C++ adalah **Flex** (Fast Lexical Analyzer) dan **re2c** (Regular Expressions to Code).

Gambar 4.1 menunjukkan alur kerja lexer generator secara umum.




Gambar 4.1: Alur kerja lexer generator

Keuntungan menggunakan lexer generator:

- **Produktivitas**: Lebih cepat dalam development karena tidak perlu menulis state machine manual
- **Maintainability**: Specification file lebih mudah dibaca dan dimodifikasi dibanding kode state machine
- **Optimasi Otomatis**: Generator menghasilkan kode yang sudah dioptimasi (DFA minimization, dll.)
- **Konsistensi**: Mengurangi bug karena generator sudah teruji

## 4.3 Flex (Fast Lexical Analyzer)

Flex adalah lexer generator yang paling banyak digunakan, terutama dalam kombinasi dengan Bison (parser generator). Flex membaca specification file dengan ekstensi `.l` dan menghasilkan kode C untuk lexer.

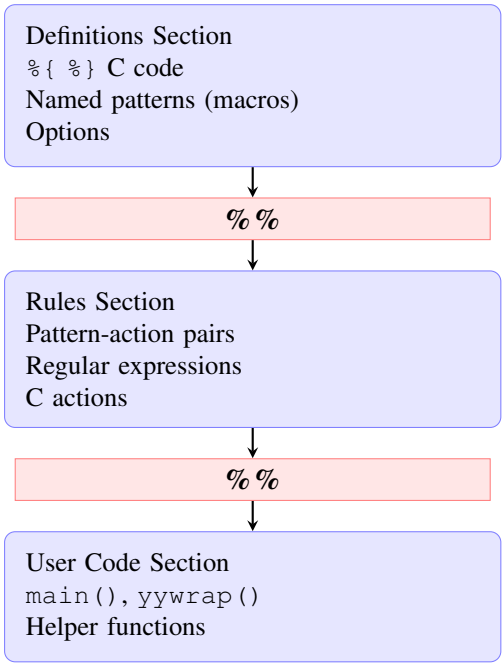
### 4.3.1 Struktur Flex Specification File

File specification Flex ('.l') terdiri dari tiga bagian yang dipisahkan oleh '%%':

```
Definitions
%%
```

```
Rules
%%
User Code
```

Gambar 4.2 menunjukkan struktur file specification Flex secara visual.




Gambar 4.2: Struktur file specification Flex

### Definitions Section

Bagian ini berisi:

- **Named patterns (macros)**: Definisi pattern yang dapat digunakan kembali
- **C code**: Kode C yang akan disalin langsung ke output (dalam ‘%{ %}‘)
- **Options**: Konfigurasi Flex (misalnya ‘%option noyywrap‘)

Contoh:

```
%{
#include <stdio.h>
#include "tokens.h"  // Definisi token constants
%}

DIGIT    [0-9]
LETTER   [a-zA-Z]
ID       {LETTER}({LETTER}|{DIGIT})*
NUMBER   {DIGIT}+
```

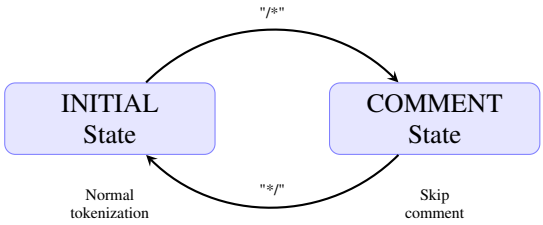
### Rules Section

Bagian ini berisi pattern-action pairs. Pattern menggunakan regular expression, dan action adalah kode C yang dieksekusi ketika pattern match.

Contoh:

```
%%
"if"           { return IF; }
"else"         { return ELSE; }
"while"        { return WHILE; }
{ID}           { return IDENTIFIER; }
{NUMBER}       { yylval.intval = atoi(yytext); return NUMBER; }
"=="           { return EQ; }
"!="           { return NE; }
[ \t\n]+       { /* skip whitespace */ }
"//".*         { /* skip single-line comment */ }
"/*"           { BEGIN(COMMENT); }
<COMMENT>"*/"  { BEGIN(INITIAL); }
<COMMENT>.     { /* skip comment content */ }
.              { return yytext[0]; }  /* default: return character */
%%
```

Gambar 4.3 menunjukkan penggunaan start conditions dalam Flex untuk menangani komentar multi-line.




Gambar 4.3: Start conditions dalam Flex untuk handling komentar

### User Code Section

Bagian ini berisi fungsi-fungsi pendukung seperti ‘main()‘, ‘yywrap()‘, dan helper functions.

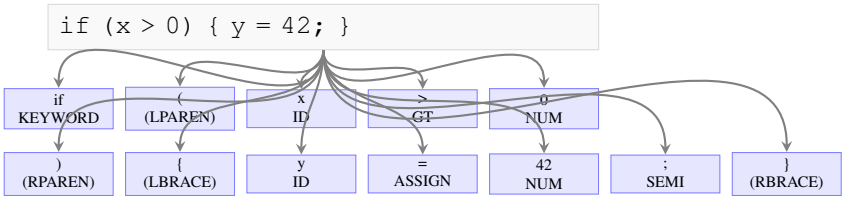
Gambar 4.4 menunjukkan contoh lengkap penggunaan Flex dari input hingga output token, mengikuti format yang konsisten dengan diagram tokenization di bab sebelumnya.

Gambar 4.5 menunjukkan proses pattern matching dalam Flex untuk keyword `"if"`.

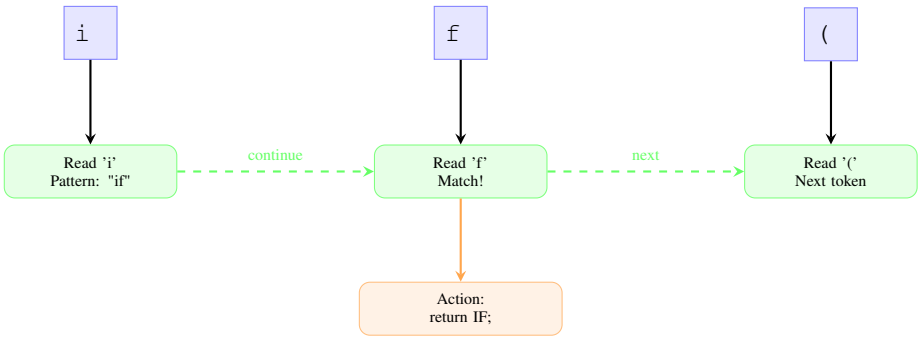
Gambar 4.6 menunjukkan berbagai jenis token yang dapat dikenali oleh Flex.

## 4.3.2 Contoh Lengkap: Flex Lexer untuk Bahasa Sederhana

Berikut adalah contoh specification file Flex untuk bahasa sederhana dengan token: identifier, number, keyword, dan operator:

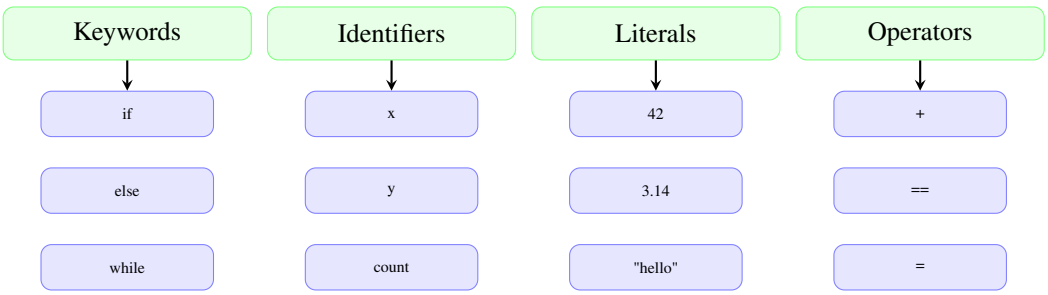



Gambar 4.4: Contoh lengkap tokenization dengan Flex: `if (x > 0) { y = 42; }`




Flex mencocokkan pola `"if"` dan mengeksekusi aksi leksikal

Gambar 4.5: Proses pattern matching dalam Flex: pencocokan keyword `"if"`




Gambar 4.6: Jenis-jenis token yang dikenali Flex

Listing 4.1: Contoh Flex specification file (calc.l)

```
%{
#include <stdio.h>
#include <stdlib.h>
#include "parser.tab.h"  // Header dari Bison

int yylineno = 1;
%}

%option noyywrap
%option yylineno

DIGIT    [0-9]
LETTER   [a-zA-Z_]
ID       {LETTER}({LETTER}|{DIGIT})*
NUMBER   {DIGIT}+
FLOAT    {DIGIT}+\.{DIGIT}+

%%

"int"        { return INT; }
"float"      { return FLOAT_TYPE; }
"if"         { return IF; }
"else"       { return ELSE; }
"while"      { return WHILE; }
"return"   { return RETURN; }

{ID}         {
                yylval.string = strdup(yytext);
                return IDENTIFIER;
             }

{NUMBER}     {
                yylval.intval = atoi(yytext);
                return NUMBER;
             }

{FLOAT}      {
                yylval.floatval = atof(yytext);
                return FLOAT_LITERAL;
             }

"+"          { return PLUS; }
"-"          { return MINUS; }
"*"          { return MULTIPLY; }
"/"          { return DIVIDE; }
"="          { return ASSIGN; }
"=="         { return EQ; }
"!="         { return NE; }
"<"          { return LT; }
">"          { return GT; }
"<="         { return LE; }
">="         { return GE; }

```

```
"("             { return LPAREN; }
")"             { return RPAREN; }
"{"             { return LBRACE; }
"}"             { return RBRACE; }
";"             { return SEMICOLON; }
","             { return COMMA; }

[ \t]+          { /* skip whitespace */ }
\n              { yylineno++; }
"//".*          { /* skip single-line comment */ }
"/*"            {
                    int c;
                    while ((c = input()) != EOF) {
                        if (c == '\n') yylineno++;
                        if (c == '*' && (c = input()) == '/') break;
                        if (c != EOF) unput(c);
                    }
                }

.               {
                    fprintf(stderr, "Error: unexpected character '%c' at line
↪  %d\n",
                            yytext[0], yylineno);
                    return ERROR;
                }

%%

int yywrap(void) {
    return 1;
}
```

Gambar 4.7 menunjukkan workflow kompilasi dan penggunaan Flex dan re2c secara perbandingan.

Gambar 4.8 menunjukkan bagaimana Flex menangani rule priority dan longest match.

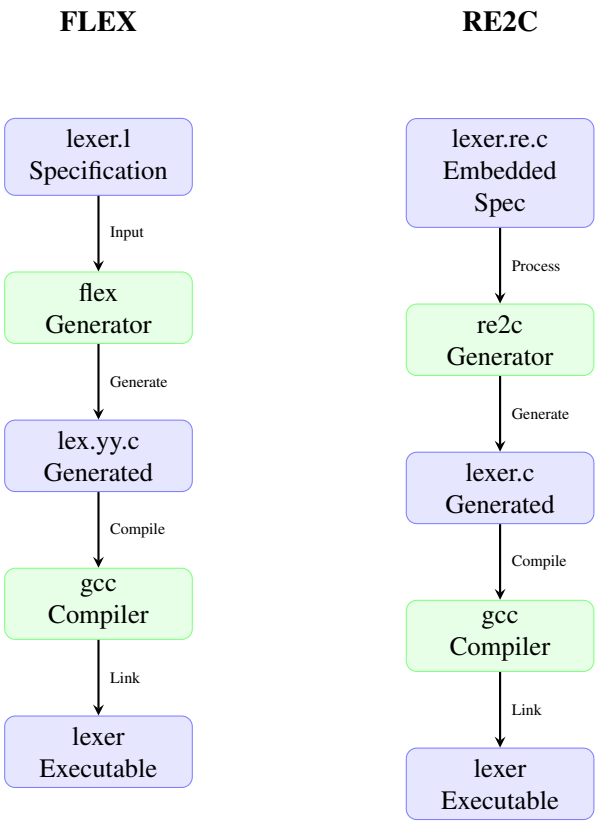
### 4.3.3 Kompilasi dan Penggunaan Flex

Untuk menggunakan Flex:

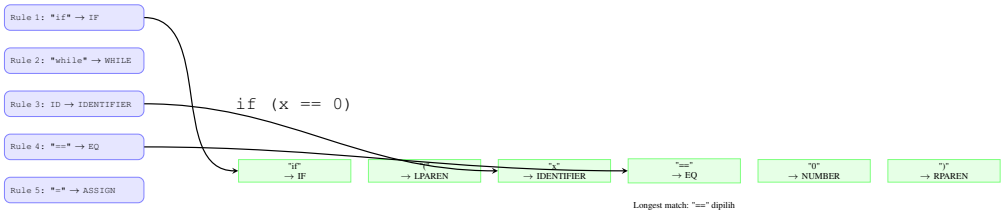
1. Buat file specification (misalnya `lexer.l`)
2. Generate lexer code: `flex lexer.l` (menghasilkan `lex.yy.c`)
3. Compile dengan compiler C: `gcc lex.yy.c -o lexer -lfl`
4. Atau link dengan program utama: `gcc main.c lex.yy.c -o program -lfl`

Fungsi utama yang digunakan:

- `yylex()`: Fungsi yang dipanggil untuk mendapatkan token berikutnya



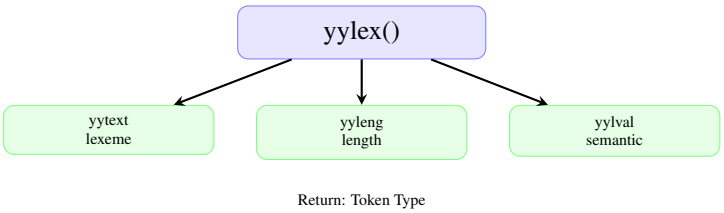

Gambar 4.7: Workflow kompilasi dan penggunaan Flex dan re2c




Gambar 4.8: Rule priority dan longest match dalam Flex

- `yytext`: String yang berisi lexeme yang baru saja di-match
- `yyleng`: Panjang dari `yytext`
- `yylval`: Union untuk menyimpan nilai semantic (untuk integrasi dengan parser)

Gambar 4.9 menunjukkan penggunaan fungsi-fungsi utama Flex.




Gambar 4.9: Fungsi dan variabel utama dalam Flex

Perbedaan utama antara Flex dan re2c adalah Flex menggunakan **separate file** (lexer.l) sedangkan re2c menggunakan **embedded specification** dalam kode C/C++ (lexer.re.c). Detail workflow kompilasi keduanya dapat dilihat pada Gambar 4.7.

## 4.4 re2c (Regular Expressions to Code)

re2c adalah lexer generator modern yang menghasilkan kode C/C++ dengan performa tinggi. Berbeda dengan Flex yang menggunakan file terpisah, re2c menggunakan **embedded specification** dalam kode C/C++.

Menurut dokumentasi resmi re2c:

> "re2c is a tool that generates fast lexers for C, C++ and Go. It compiles regular expressions to deterministic finite automata and encodes them as conditional jumps and comparisons. The generated code is highly optimized and does not use tables."[1]

### 4.4.1 Struktur re2c Specification

re2c specification ditulis dalam komentar khusus '/*!re2c ... */' yang disisipkan dalam kode C/C++:

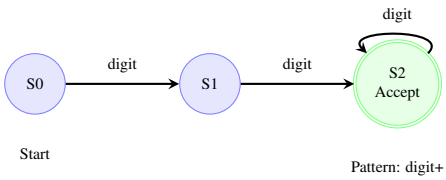
Listing 4.2: Struktur dasar re2c

```
1 #include <stdio.h>
2
3 static int lex(const char *YYCURSOR) {
4     const char *YYMARKER;
```

[1] https://re2c.org/

```
    /*!re2c
        re2c:define:YYCTYPE = "char";
        re2c:yyfill:enable = 0;

        // Named patterns
        digit    = [0-9];
        letter   = [a-zA-Z_];
        id       = letter (letter | digit)*;
        number   = digit+;

        // Rules
        *        { return 0; }  // error
        number   { printf("Number: %.*s\n", (int)(YYCURSOR - YYMARKER),
↪ YYMARKER); return 1; }
        id       { printf("ID: %.*s\n", (int)(YYCURSOR - YYMARKER),
↪ YYMARKER); return 1; }
        [ \t\n]+ { continue; }  // skip whitespace
    */
}

int main(int argc, char *argv[]) {
    for (int i = 1; i < argc; i++) {
        lex(argv[i]);
    }
    return 0;
}
```

Gambar 4.10 menunjukkan bagaimana re2c menghasilkan state machine untuk pattern matching.




Gambar 4.10: State machine yang dihasilkan re2c untuk pattern `digit+`

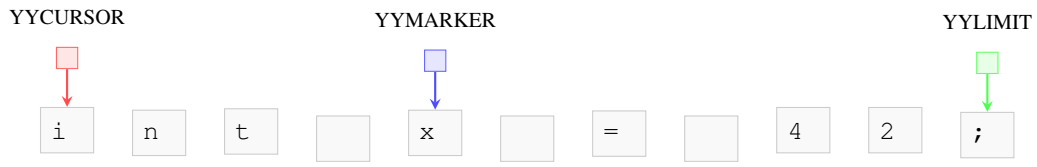
### 4.4.2 Key Concepts dalam re2c

Gambar 4.11 menunjukkan penggunaan variabel khusus dalam re2c.

#### Variables Khusus

re2c menggunakan variabel khusus untuk tracking posisi input:

- `YYCURSOR`: Pointer ke posisi saat ini dalam input
- `YYMARKER`: Pointer untuk backtracking




Gambar 4.11: Ilustrasi pointer internal `re2c` dalam buffer input

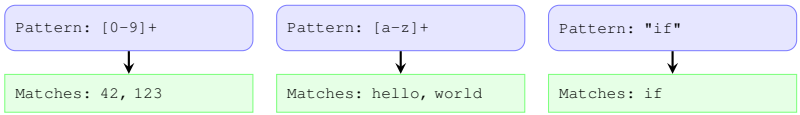
- `YYLIMIT`: Pointer ke akhir buffer
- `YYCTYPE`: Tipe data untuk karakter (default: `unsigned char`)

**Directives**

Directives mengkonfigurasi behavior re2c:

- `re2c:define:YYCTYPE`: Mendefinisikan tipe karakter
- `re2c:yyfill:enable`: Enable/disable buffer filling
- `re2c:input`: Mendefinisikan cara membaca input
- `re2c:conditions`: Enable start conditions (seperti Flex)

Gambar 4.12 menunjukkan berbagai pattern yang dapat digunakan dalam re2c.




Gambar 4.12: Contoh pattern dalam re2c

### 4.4.3 Contoh Lengkap: re2c Lexer untuk Identifier dan Number

Listing 4.3: Contoh re2c lexer (lexer.re.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef enum {
    TOKEN_EOF,
    TOKEN_IDENTIFIER,
    TOKEN_NUMBER,
    TOKEN_PLUS,
    TOKEN_MINUS,
    TOKEN_MULTIPLY,
    TOKEN_DIVIDE,
```

```
    TOKEN_ERROR
} TokenType;

typedef struct {
    TokenType type;
    char *value;
    int line;
} Token;

static Token tokenize(const char *input, int *line) {
    const char *YYCURSOR = input;
    const char *YYMARKER;
    const char *start;
    Token token = {TOKEN_EOF, NULL, *line};

    /*!re2c
        re2c:define:YYCTYPE = "char";
        re2c:yyfill:enable = 0;
        re2c:define:YYCURSOR = "YYCURSOR";

        digit    = [0-9];
        letter   = [a-zA-Z_];
        id       = letter (letter | digit)*;
        number   = digit+;
        ws       = [ \t]+;
        newline  = "\n";

        * {
            token.type = TOKEN_ERROR;
            return token;
        }

        "\x00" {
            token.type = TOKEN_EOF;
            return token;
        }

        ws {
            continue;
        }

        newline {
            (*line)++;
            continue;
        }

        number {
            start = YYMARKER;
            int len = YYCURSOR - start;
            token.value = (char*)malloc(len + 1);
            strncpy(token.value, start, len);
            token.value[len] = '\0';
            token.type = TOKEN_NUMBER;
            token.line = *line;
```

```
67              return token;
68          }
69
70          id {
71              start = YYMARKER;
72              int len = YYCURSOR - start;
73              token.value = (char*)malloc(len + 1);
74              strncpy(token.value, start, len);
75              token.value[len] = '\0';
76              token.type = TOKEN_IDENTIFIER;
77              token.line = *line;
78              return token;
79          }
80
81          "+" {
82              token.type = TOKEN_PLUS;
83              token.line = *line;
84              return token;
85          }
86
87          "-" {
88              token.type = TOKEN_MINUS;
89              token.line = *line;
90              return token;
91          }
92
93          "*" {
94              token.type = TOKEN_MULTIPLY;
95              token.line = *line;
96              return token;
97          }
98
99          "/" {
100             token.type = TOKEN_DIVIDE;
101             token.line = *line;
102             return token;
103         }
104     */
105 }
106
107 int main(int argc, char *argv[]) {
108     if (argc < 2) {
109         fprintf(stderr, "Usage: %s <input>\n", argv[0]);
110         return 1;
111     }
112
113     int line = 1;
114     Token token;
115
116     do {
117         token = tokenize(argv[1], &line);
118         printf("Token: %d, Value: %s, Line: %d\n",
119                token.type, token.value ? token.value : "NULL", token.line
    ↪ );
```

```
        if (token.value) free(token.value);
    } while (token.type != TOKEN_EOF && token.type != TOKEN_ERROR);

    return 0;
}
```

Untuk mengkompilasi:

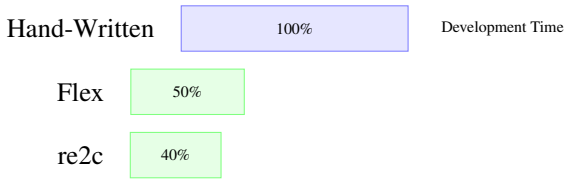
```
re2c -o lexer.c lexer.re.c
gcc lexer.c -o lexer
```

Gambar 4.13 menunjukkan perbandingan visual antara hand-written, Flex, dan re2c.

| Aspek | Hand-Written | Flex | re2c |
|---|---|---|---|
| Produktivitas | Rendah | Tinggi | Tinggi |
| Maintainability | Sedang | Tinggi | Tinggi |
| Performa | Tinggi | Sedang | Sangat Tinggi |
| Fleksibilitas | Sangat Tinggi | Sedang | Sedang |

Gambar 4.13: Perbandingan hand-written, Flex, dan re2c

Gambar 4.14 menunjukkan perbandingan waktu development antara berbagai pendekatan.




Gambar 4.14: Perbandingan waktu development (hand-written = baseline)

## 4.5 Perbandingan Hand-Written vs Generator-Based Lexer

Setelah mempelajari kedua pendekatan, mari kita bandingkan:

### 4.5.1 Hand-Written Lexer

**Keuntungan:**

- Kontrol penuh terhadap implementasi
- Tidak ada dependency eksternal

- Dapat dioptimasi secara spesifik untuk kebutuhan
- Pemahaman mendalam tentang proses tokenization

**Kekurangan:**

- Lebih banyak kode yang harus ditulis dan maintain
- Lebih mudah terjadi bug (edge cases)
- Perlu implementasi ulang untuk setiap bahasa
- Lebih sulit untuk modifikasi pattern

### 4.5.2 Generator-Based Lexer

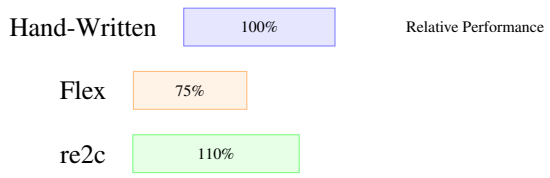
**Keuntungan:**

- Specification lebih ringkas dan mudah dibaca
- Generator menghasilkan kode yang sudah teroptimasi
- Lebih cepat dalam development
- Pattern mudah dimodifikasi tanpa mengubah banyak kode
- Sudah teruji dan digunakan di banyak project

**Kekurangan:**

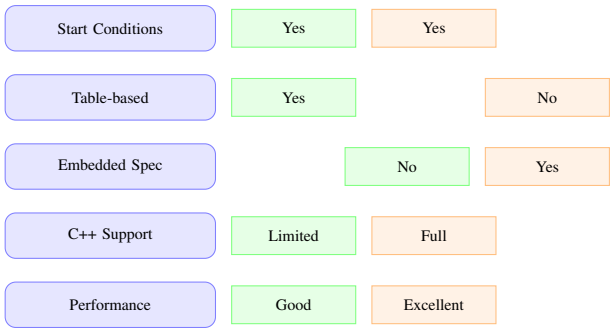
- Perlu mempelajari syntax generator
- Dependency pada tool eksternal
- Kurang fleksibel untuk kasus yang sangat spesifik
- Generated code mungkin lebih sulit di-debug

Gambar 4.15 menunjukkan perbandingan performa secara visual.




Gambar 4.15: Perbandingan performa relatif (hand-written = baseline)

Gambar 4.16 menunjukkan perbandingan fitur antara Flex dan re2c.




Gambar 4.16: Perbandingan fitur Flex vs re2c

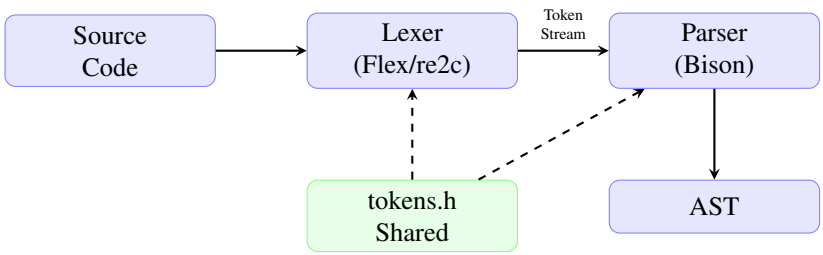
### 4.5.3 Perbandingan Performa

Secara umum, generator-based lexer (terutama re2c) memiliki performa yang sangat baik karena:

- Kode dihasilkan dengan optimasi DFA
- Tidak ada overhead dari table lookup (untuk re2c)
- Compiler dapat mengoptimasi generated code lebih baik

Hand-written lexer dapat lebih cepat hanya jika dioptimasi secara khusus untuk kasus tertentu, tetapi memerlukan effort yang lebih besar.

Gambar 4.17 menunjukkan alur integrasi lexer dengan parser.




Gambar 4.17: Integrasi lexer dengan parser

## 4.6 Integrasi dengan Parser

Lexer biasanya digunakan bersama dengan parser. Integrasi dilakukan melalui:

### 4.6.1 Token Definitions

Token constants didefinisikan dalam header file yang dibagi antara lexer dan parser:

Listing 4.4: File tokens.h

```
#ifndef TOKENS_H
#define TOKENS_H

typedef enum {
    // Keywords
    TOKEN_IF = 256,
    TOKEN_ELSE,
    TOKEN_WHILE,
    TOKEN_RETURN,
    TOKEN_INT,
    TOKEN_FLOAT,

    // Identifiers and literals
    TOKEN_IDENTIFIER,
    TOKEN_NUMBER,
    TOKEN_FLOAT_LITERAL,

    // Operators
    TOKEN_PLUS,
    TOKEN_MINUS,
    TOKEN_MULTIPLY,
    TOKEN_DIVIDE,
    TOKEN_ASSIGN,
    TOKEN_EQ,
    TOKEN_NE,

    // Punctuation
    TOKEN_LPAREN,
    TOKEN_RPAREN,
    TOKEN_LBRACE,
    TOKEN_RBRACE,
    TOKEN_SEMICOLON,
    TOKEN_COMMA,

    TOKEN_EOF,
    TOKEN_ERROR
} TokenType;

#endif
```

### 4.6.2 Semantic Values

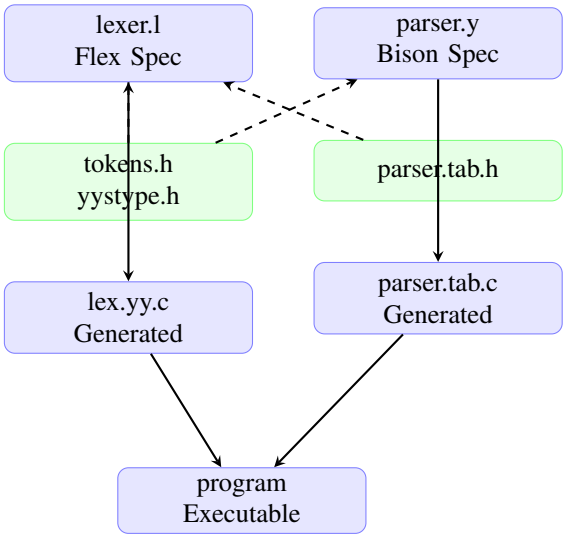
Untuk mengirim nilai dari lexer ke parser, digunakan union `yylval`:

Listing 4.5: File yystype.h

```
#ifndef YYSTYPE_H
#define YYSTYPE_H

#include "tokens.h"

typedef union {
    int intval;
```

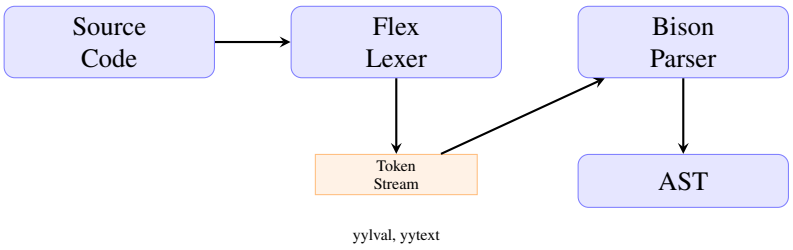
```
    double floatval;
    char *string;
} YYSTYPE;

extern YYSTYPE yylval;

#endif
```

Gambar 4.18 menunjukkan contoh integrasi Flex dengan Bison secara detail.




Gambar 4.18: Integrasi Flex dengan Bison: file dan dependencies

Gambar 4.19 menunjukkan alur data dalam integrasi Flex-Bison.




Gambar 4.19: Alur data dalam integrasi Flex-Bison

### 4.6.3 Contoh Integrasi Flex dengan Bison

File Flex (`lexer.l`):

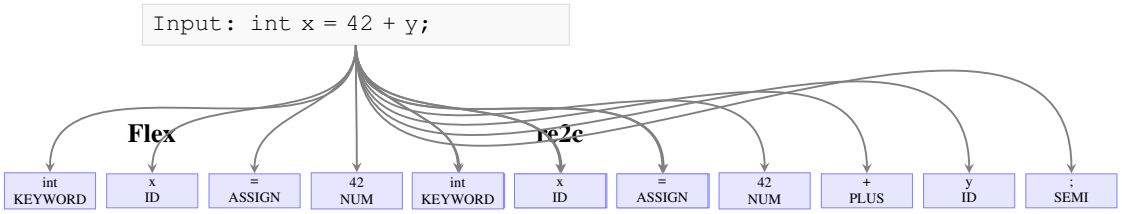
```
%{
#include "parser.tab.h"
#include "yystype.h"
%}

```

```
%%
{NUMBER} { yylval.intval = atoi(yytext); return NUMBER; }
{ID}     { yylval.string = strdup(yytext); return IDENTIFIER; }
%%
```

File Bison (`parser.y`):

```
%{
#include "yystype.h"
%}

%union {
    int intval;
    char *string;
}

%token <intval> NUMBER
%token <string> IDENTIFIER

%%
expression: NUMBER { printf("Number: %d\n", $1); }
          | IDENTIFIER { printf("ID: %s\n", $1); }
          ;
%%
```

Gambar 4.20 menunjukkan contoh tokenization menggunakan Flex dan re2c untuk input yang sama, menunjukkan bahwa kedua tool menghasilkan hasil yang konsisten.




Gambar 4.20: Perbandingan tokenization: Flex dan re2c menghasilkan token stream yang konsisten untuk input `int x = 42 + y;`

## 4.7 Praktikum: Membuat Lexer dengan Flex

### 4.7.1 Tugas Praktikum

Buatlah lexer menggunakan Flex untuk bahasa mini dengan minimal 10 token types:

1. **Keywords**: `if`, `else`, `while`, `int`, `float`, `return`
2. **Identifiers**: Nama variabel dan fungsi
3. **Literals**: Integer dan float numbers
4. **Operators**: +, −, ⋆, /, =, ==, !=, <, >
5. **Punctuation**: `(`, `)`, `{`, `}`, `;`, `,`
6. **Comments**: Single-line (`//`) dan multi-line (`/* */`)

### 4.7.2 Langkah-langkah

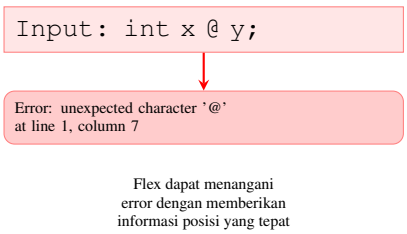
1. Buat file `lexer.l` dengan specification sesuai requirement
2. Generate lexer: `flex lexer.l`
3. Buat program test sederhana yang menggunakan `yylex()`
4. Test dengan berbagai input (valid dan invalid)
5. Dokumentasikan token types dan behavior lexer

### 4.7.3 Expected Output

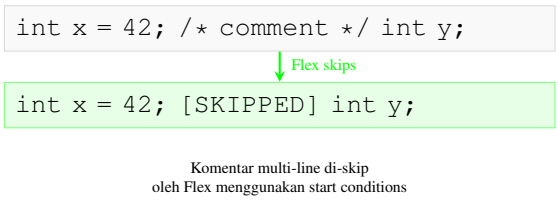
Lexer harus dapat:

- Mengenali semua token types yang didefinisikan
- Menangani whitespace dan comments dengan benar
- Memberikan error message yang informatif untuk invalid input
- Melacak line number untuk error reporting

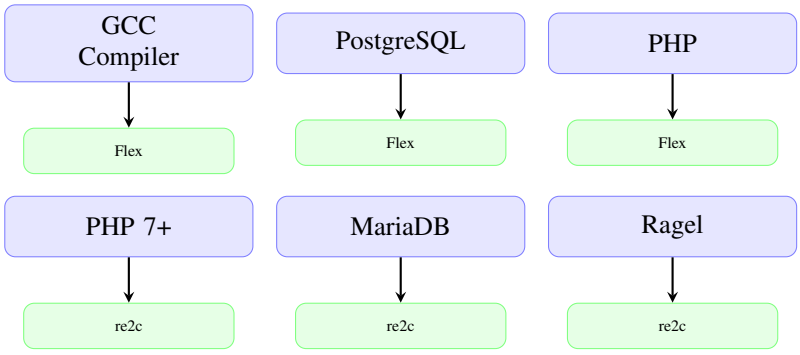
Gambar 4.21 menunjukkan contoh error handling dalam Flex.
Gambar 4.22 menunjukkan bagaimana Flex menangani komentar multi-line.
Gambar 4.23 menunjukkan penggunaan Flex dan re2c dalam project nyata.




Gambar 4.21: Error handling dalam Flex




Gambar 4.22: Handling komentar multi-line dalam Flex




Gambar 4.23: Contoh penggunaan Flex dan re2c dalam project nyata

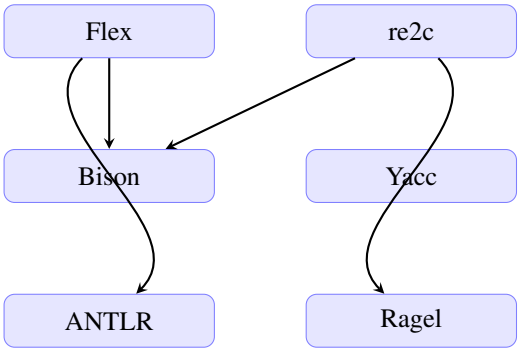
## 4.8 Kesimpulan

Dalam bab ini, kita telah mempelajari:

1. Keuntungan menggunakan lexer generator dibanding hand-written lexer
2. Cara menggunakan Flex untuk membuat specification file dan generate lexer
3. Cara menggunakan re2c dengan embedded specification
4. Perbandingan antara hand-written dan generator-based lexer
5. Integrasi lexer dengan parser menggunakan token definitions dan semantic values

Generator-based lexer adalah pilihan yang tepat untuk sebagian besar kasus karena memberikan keseimbangan yang baik antara produktivitas, maintainability, dan performa. Namun, pemahaman tentang hand-written lexer (seperti yang dipelajari di bab sebelumnya) tetap penting untuk memahami proses tokenization secara mendalam.

Gambar 4.24 menunjukkan ekosistem tools yang terkait dengan lexer generator.




Gambar 4.24: Ekosistem tools lexer dan parser generator

## 4.9 Referensi dan Bahan Bacaan Lanjutan

Untuk memperdalam pemahaman tentang lexer generator, mahasiswa disarankan membaca:

- **Flex Manual**: Dokumentasi resmi Flex [2]
- **re2c Documentation**: Dokumentasi dan tutorial re2c [3]
- **flex & bison**: Levine, J. R. (2009). *flex & bison: Text Processing Tools* [10] - Bab 2: Using Flex

[2] https://www.gnu.org/software/flex/manual/
[3] https://re2c.org/

- **Engineering a Compiler**: Cooper & Torczon (2011) [5] - Bab 2: Lexical Analysis (bagian tentang lexer generators)
- **IT Trip - C Parser Flex Bison**: Tutorial tentang integrasi Flex dan Bison [11]
- **Wikipedia - re2c**: Artikel tentang re2c [9]
- **Wikipedia - RE/flex**: Artikel tentang RE/flex (modern C++ lexer generator) [4]

[4] https://en.wikipedia.org/wiki/Draft:RE/flex

# Bab 5

## Context-Free Grammar dan Pengenalan Parsing

### 5.1 Tujuan Pembelajaran

Setelah mempelajari bab ini, mahasiswa diharapkan mampu:

1. Memahami konsep context-free grammar (CFG) dan perannya dalam syntax analysis
2. Menjelaskan notasi BNF (Backus-Naur Form) dan EBNF (Extended BNF)
3. Menulis grammar untuk ekspresi aritmatika dan konstruksi bahasa sederhana
4. Memahami konsep derivation (leftmost dan rightmost)
5. Membuat parse tree untuk kalimat yang diberikan
6. Mengidentifikasi dan menjelaskan ambiguity dalam grammar
7. Memahami hubungan antara grammar, parsing, dan syntax analysis

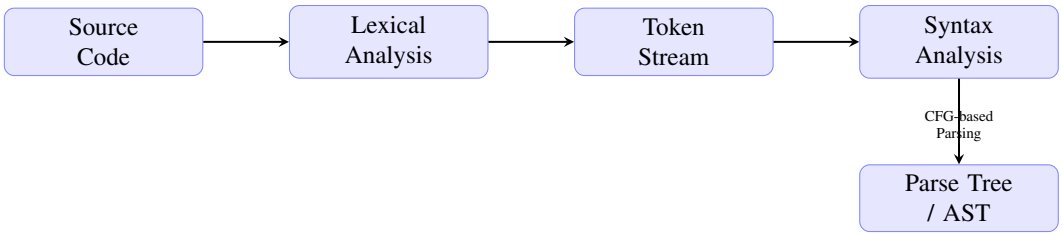
### 5.2 Pendahuluan

Setelah lexical analysis menghasilkan stream token, fase berikutnya dalam kompilator adalah syntax analysis atau parsing. Menurut sumber terbuka:

> "Given the stream of tokens from the lexer, syntax analysis checks whether they form a valid sequence under the language grammar. Builds a parse tree or AST that represents nested structure of language constructs."[2]

Syntax analysis membutuhkan formal grammar untuk mendefinisikan struktur yang valid dalam bahasa pemrograman. Context-free grammar (CFG) adalah alat formal yang paling umum digunakan untuk tujuan ini karena kemampuannya dalam menangani struktur nested dan recursive yang umum ditemui dalam bahasa pemrograman.

Gambar 5.1 menunjukkan posisi syntax analysis dalam pipeline kompilator.




Gambar 5.1: Posisi syntax analysis dalam pipeline kompilator

## 5.3 Grammar Proyek Subset C

Grammar lengkap proyek compiler subset C telah didefinisikan di Bab 1 (Bagian 1.9). Berikut ringkasan dalam notasi BNF yang akan dipakai di Bab 6 (parser hand-written) dan Bab 8 (parser Bison):

- **program**: satu atau lebih statement.
- **statement**: declaration `;` atau assignment `;` atau print-statement `;`
- **declaration**: `int` identifier atau `float` identifier
- **assignment**: identifier = expr
- **print-statement**: `print (` string-literal `)` atau `print (` expr `)`
- **expr**: term, atau expr + term, atau expr − term (associativity kiri)
- **term**: factor, atau term `*` factor, atau term `/` factor (associativity kiri)
- **factor**: literal, atau identifier, atau `(` expr `)`

Precedence: `*` dan `/` lebih tinggi dari `+` dan `-`. Parser proyek di Bab 8 mengimplementasikan grammar ini dalam file `simplec.y`; lexer proyek di Bab 4 (`simplec.l`) menghasilkan token set yang sesuai dengan spesifikasi Bab 1.

## 5.4 Context-Free Grammar (CFG)

### 5.4.1 Definisi Formal

Context-free grammar adalah tipe formal grammar yang didefinisikan sebagai tuple $G = (V, \Sigma, R, S)$ dimana:

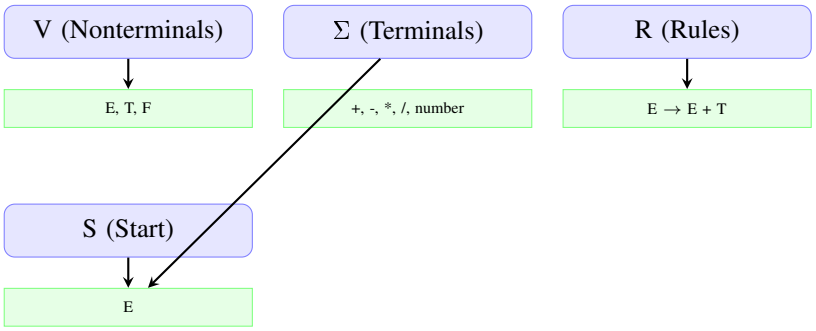
- $V$ adalah himpunan **nonterminal symbols** (variabel yang dapat di-expand)
- $\Sigma$ adalah himpunan **terminal symbols** (token yang tidak dapat di-expand, disjoint dari $V$)

- $R$ adalah himpunan **productions** (rules) dengan bentuk $A \to \alpha$, dimana $A \in V$ dan $\alpha \in (V \cup \Sigma)^*$

- $S \in V$ adalah **start symbol**

CFG disebut "context-free" karena aturan produksi dapat diterapkan tanpa mempertimbangkan konteks di sekitar nonterminal. Artinya, jika ada produksi $A \to \alpha$, maka $A$ dapat diganti dengan $\alpha$ di manapun $A$ muncul, terlepas dari simbol di sekitarnya.

Gambar 5.2 menunjukkan komponen-komponen CFG secara visual.




Gambar 5.2: Komponen-komponen CFG: $G = (V, \Sigma, R, S)$

### 5.4.2 Contoh Grammar Sederhana

Mari kita lihat contoh grammar untuk ekspresi aritmatika sederhana:

```
E → E + T | E - T | T
T → T * F | T / F | F
F → ( E ) | number
```

Dalam grammar ini:

- **Nonterminals**: $E$ (expression), $T$ (term), $F$ (factor)

- **Terminals**: `+`, `-`, `*`, `/`, `(`, `)`, `number`

- **Start symbol**: $E$

Grammar ini dapat menghasilkan ekspresi seperti `3 + 4 * 5`, `(2 + 3) * 4`, dll.

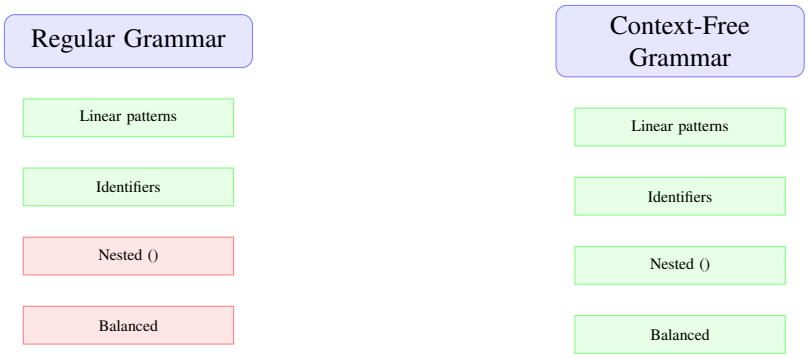
### 5.4.3 Perbedaan Regular Grammar dan Context-Free Grammar

Penting untuk memahami perbedaan antara regular grammar (yang digunakan untuk lexical analysis) dan context-free grammar:

- **Regular Grammar**: Hanya dapat menangani struktur linear, tidak dapat menangani nested structures seperti parentheses yang seimbang
- **Context-Free Grammar**: Dapat menangani struktur nested dan recursive, seperti:
  - Parentheses matching: `( ( ( ) ) )`
  - Nested blocks: `{ { } }`
  - Recursive function calls
  - Nested if-statements

Inilah mengapa CFG digunakan untuk syntax analysis, sementara regular grammar cukup untuk lexical analysis.

Gambar 5.3 menunjukkan perbandingan kemampuan regular grammar dan CFG.




Gambar 5.3: Perbandingan kemampuan Regular Grammar vs CFG

## 5.5 BNF dan EBNF Notasi

### 5.5.1 Backus-Naur Form (BNF)

BNF adalah notasi metalanguage yang dikembangkan oleh John Backus dan Peter Naur untuk mendefinisikan syntax bahasa pemrograman. BNF menggunakan simbol-simbol berikut:

- $::=$ atau $\rightarrow$: Menandakan "didefinisikan sebagai"
- $|$: Menandakan alternatif (OR)

- `<nonterminal>`: Nonterminal symbol (biasanya dalam angle brackets)
- `terminal`: Terminal symbol (biasanya tanpa angle brackets)

Contoh grammar dalam BNF:

```
<expression> ::= <expression> + <term>
               | <expression> - <term>
               | <term>

<term> ::= <term> * <factor>
         | <term> / <factor>
         | <factor>

<factor> ::= ( <expression> )
           | <number>
```

### 5.5.2 Extended BNF (EBNF)

EBNF memperluas BNF dengan konstruksi tambahan untuk membuat grammar lebih kompak dan mudah dibaca:

- **Optional**: `[ ... ]` atau `?` - elemen opsional (nol atau satu kali)
- **Repetition**:
  - `*`: Nol atau lebih kali
  - `+`: Satu atau lebih kali
- **Grouping**: `( ... )` - untuk mengelompokkan
- **Terminal strings**: `"..."` atau `'...'` - untuk literal strings

Contoh grammar yang sama dalam EBNF:

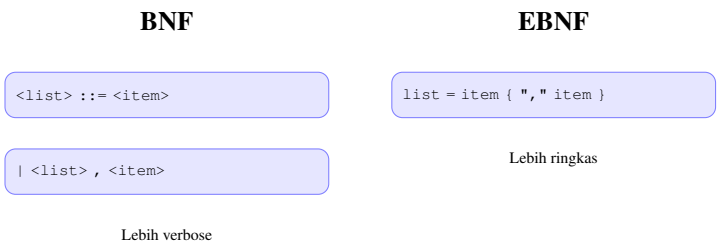
```
expression = term { ("+" | "-") term }
term       = factor { ("*" | "/") factor }
factor     = "(" expression ")" | number
```

EBNF lebih ringkas dan mudah dibaca. Banyak spesifikasi bahasa modern menggunakan EBNF, termasuk ISO standard untuk grammar notation.

Gambar 5.4 menunjukkan perbandingan notasi BNF dan EBNF.

### 5.5.3 Contoh Grammar untuk Konstruksi Bahasa

Berikut contoh grammar untuk beberapa konstruksi bahasa pemrograman:




Gambar 5.4: Perbandingan notasi BNF dan EBNF

### If-Statement

```
<if_statement> ::= if ( <expression> ) <statement>
                 | if ( <expression> ) <statement> else <statement>
```

Dalam EBNF:

```
if_statement = "if" "(" expression ")" statement
               [ "else" statement ]
```

### While Loop

```
<while_statement> ::= while ( <expression> ) <statement>
```

Dalam EBNF:

```
while_statement = "while" "(" expression ")" statement
```

### Variable Declaration

```
<declaration> ::= <type> <identifier> [ = <expression> ] ;
```

Dalam EBNF:

```
declaration = type identifier [ "=" expression ] ";"
```

## 5.6 Derivation

Derivation adalah proses menerapkan aturan produksi untuk menghasilkan string terminal dari start symbol. Terdapat dua jenis derivation yang penting:

### 5.6.1 Leftmost Derivation

Leftmost derivation selalu mengganti nonterminal paling kiri terlebih dahulu pada setiap langkah.

Contoh untuk ekspresi `3 + 4 * 5` dengan grammar:

```
E → E + T | E - T | T
T → T * F | T / F | F
F → ( E ) | number
```

Leftmost derivation:

$$\begin{aligned} E &\Rightarrow E + T \\ &\Rightarrow T + T \\ &\Rightarrow F + T \\ &\Rightarrow \texttt{3} + T \\ &\Rightarrow \texttt{3} + T * F \\ &\Rightarrow \texttt{3} + F * F \\ &\Rightarrow \texttt{3} + \texttt{4} * F \\ &\Rightarrow \texttt{3} + \texttt{4} * \texttt{5} \end{aligned}$$

### 5.6.2 Rightmost Derivation

Rightmost derivation selalu mengganti nonterminal paling kanan terlebih dahulu pada setiap langkah.

Rightmost derivation untuk `3 + 4 * 5`:

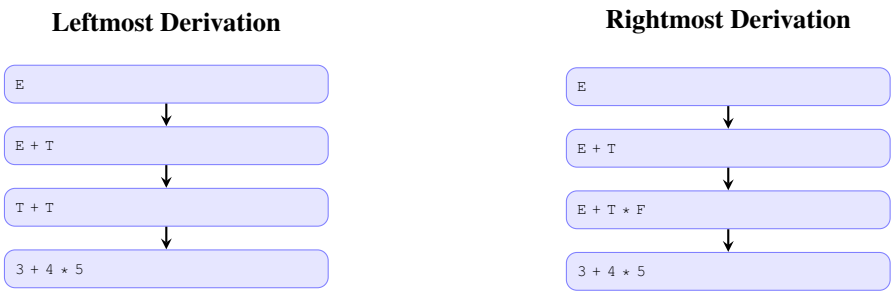
$$\begin{aligned} E &\Rightarrow E + T \\ &\Rightarrow E + T * F \\ &\Rightarrow E + T * \texttt{5} \\ &\Rightarrow E + F * \texttt{5} \\ &\Rightarrow E + \texttt{4} * \texttt{5} \\ &\Rightarrow T + \texttt{4} * \texttt{5} \\ &\Rightarrow F + \texttt{4} * \texttt{5} \\ &\Rightarrow \texttt{3} + \texttt{4} * \texttt{5} \end{aligned}$$

Gambar 5.5 menunjukkan perbandingan leftmost dan rightmost derivation.

### 5.6.3 Pentingnya Derivation

Derivation penting karena:

- Menunjukkan bagaimana string dihasilkan dari grammar
- Menentukan urutan penggantian nonterminal (penting untuk parsing)




Gambar 5.5: Perbandingan leftmost dan rightmost derivation

- Leftmost derivation digunakan dalam top-down parsing
- Rightmost derivation digunakan dalam bottom-up parsing

## 5.7 Parse Tree

Parse tree (juga disebut derivation tree atau concrete syntax tree) adalah representasi visual dari bagaimana string diturunkan dari grammar.

### 5.7.1 Struktur Parse Tree

Parse tree memiliki struktur berikut:

- **Root**: Labeled dengan start symbol $S$
- **Internal nodes**: Nonterminal symbols
- **Leaves**: Terminal symbols (dari kiri ke kanan membentuk input string)
- **Edges**: Menunjukkan aplikasi aturan produksi

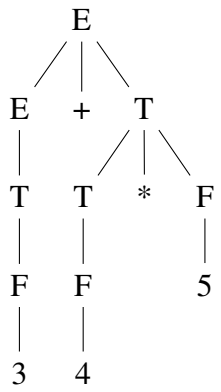
### 5.7.2 Contoh Parse Tree

Untuk ekspresi `3 + 4 * 5` dengan grammar sebelumnya, parse tree-nya adalah:

### 5.7.3 Parse Tree vs Abstract Syntax Tree (AST)

Perbedaan penting antara parse tree dan AST:

- **Parse Tree (Concrete Syntax Tree)**:
  - Mencakup semua detail sintaksis, termasuk punctuation
  - Setiap node sesuai dengan aturan produksi



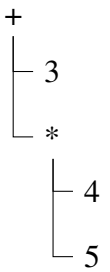

Gambar 5.6: Parse tree untuk ekspresi `3 + 4 * 5`

  - Lebih verbose, mencakup informasi yang tidak diperlukan untuk fase selanjutnya

- **Abstract Syntax Tree (AST)**:

  - Menghilangkan detail sintaksis yang tidak relevan (seperti parentheses grouping yang sudah jelas dari struktur)
  - Fokus pada struktur semantik program
  - Lebih kompak dan efisien untuk analisis semantik dan code generation

Contoh: Untuk ekspresi `3 + 4 * 5`, AST-nya lebih sederhana:




Gambar 5.7: AST untuk ekspresi `3 + 4 * 5`

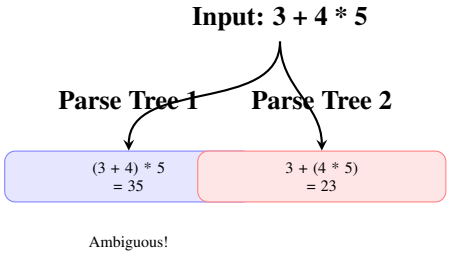
AST menghilangkan node-node intermediate seperti $E$, $T$, $F$ yang tidak diperlukan untuk pemahaman semantik.

## 5.8 Ambiguity dalam Grammar

### 5.8.1 Definisi Ambiguity

Grammar dikatakan **ambiguous** jika terdapat setidaknya satu string dalam bahasa yang dapat memiliki lebih dari satu parse tree (atau derivation) yang berbeda. Ambiguity adalah masalah karena dapat menyebabkan interpretasi yang berbeda dari program yang sama.

Gambar 5.8 menunjukkan contoh ambiguity dalam grammar.




Gambar 5.8: Contoh ambiguity: dua parse tree berbeda untuk input yang sama

### 5.8.2 Contoh Grammar Ambiguous

Pertimbangkan grammar berikut untuk ekspresi:

```
E → E + E | E * E | number
```

Grammar ini ambiguous karena ekspresi `3 + 4 * 5` dapat di-parse dengan dua cara:
**Parse Tree 1** (mengasumsikan + memiliki precedence lebih tinggi):

```
      E
     /|\
    E + E
   /|   |\
  E * E 5
  |   |
  3   4
```

Ini akan mengevaluasi sebagai `(3 + 4) * 5 = 35`
**Parse Tree 2** (mengasumsikan * memiliki precedence lebih tinggi):

```
      E
     /|\
    E * E
   /|   |
  E + E 5
  |   |
  3   4
```

Ini akan mengevaluasi sebagai `3 + (4 * 5) = 23`

### 5.8.3 Mengatasi Ambiguity

Ada beberapa cara untuk mengatasi ambiguity:

1. **Menulis Grammar yang Unambiguous**: Menggunakan grammar yang secara eksplisit mendefinisikan precedence dan associativity. Contoh:

```
E → E + T | E − T | T
T → T * F | T / F | F
F → ( E ) | number
```

Grammar ini unambiguous karena:

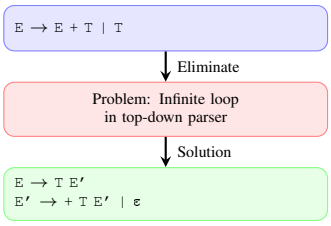
- Precedence: * dan / lebih tinggi dari + dan - (karena $T$ lebih dalam dari $E$)
- Associativity: Left-associative untuk semua operator (karena left-recursive grammar)

2. **Disambiguating Rules**: Beberapa parser generator (seperti Yacc/Bison) memungkinkan penentuan precedence dan associativity secara eksplisit tanpa mengubah grammar.

3. **Operator Precedence Parsing**: Menggunakan tabel precedence untuk menentukan urutan evaluasi.

## 5.9 Left Recursion dan Left Factoring

Gambar 5.9 menunjukkan konsep left recursion.




Gambar 5.9: Left recursion dan eliminasi

### 5.9.1 Left Recursion

Left recursion terjadi ketika nonterminal muncul di posisi paling kiri dari produksinya sendiri. Contoh:

```
E → E + T | T
```

Left recursion dapat menyebabkan masalah pada top-down parser (khususnya recursive descent) karena dapat menyebabkan infinite loop. Parser akan terus mencoba mem-parse $E$ tanpa pernah maju.

**Eliminasi Left Recursion**:
Grammar dengan left recursion:

```
A → A a | b
```

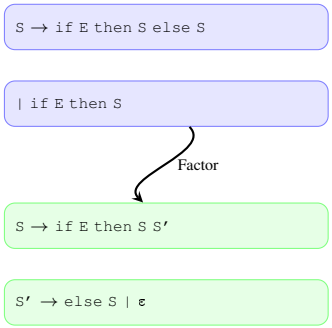
Dapat diubah menjadi:

```
A  → b A'
A' → a A' | epsilon
```

Contoh: `E → E + T | T` menjadi:

```
E  → T E'
E' → + T E' | epsilon
```

Gambar 5.10 menunjukkan konsep left factoring.




Gambar 5.10: Left factoring: sebelum dan sesudah

### 5.9.2 Left Factoring

Left factoring diperlukan ketika beberapa produksi dimulai dengan simbol yang sama, membuat parser tidak dapat memutuskan produksi mana yang harus digunakan tanpa lookahead lebih lanjut.

Contoh grammar yang membutuhkan left factoring:

```
S → if E then S else S
  | if E then S
```

Setelah left factoring:

```
S → if E then S S'
S' → else S | epsilon
```

## 5.10 Contoh Praktis: Grammar untuk Ekspresi Aritmatika

Mari kita buat grammar lengkap untuk ekspresi aritmatika yang dapat menangani:

- Operasi: +, -, *, /, mod
- Precedence: * dan / lebih tinggi dari + dan -

- Associativity: Left-associative
- Parentheses untuk grouping
- Unary minus
- Integer dan floating point numbers

Grammar dalam EBNF:

```
expression = term { ("+" | "-") term }
term       = factor { ("*" | "/" | "mod") factor }
factor     = [ "-" ] ( "(" expression ")" | number )
number     = integer | float
integer    = digit { digit }
float      = integer "." integer
digit      = "0" | "1" | ... | "9"
```

Atau dalam BNF:

```
<expression> ::= <term> | <expression> + <term> | <expression> - <term>
<term>       ::= <factor> | <term> * <factor> | <term> / <factor>
                 | <term> mod <factor>
<factor>     ::= - <factor> | ( <expression> ) | <number>
<number>     ::= <integer> | <float>
<integer>    ::= <digit> | <integer> <digit>
<float>      ::= <integer> . <integer>
<digit>      ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

## 5.11 Manual Parsing: Latihan Derivation

Mari kita lakukan manual parsing untuk ekspresi `2 + 3 * 4` menggunakan grammar:

```
E → E + T | E - T | T
T → T * F | T / F | F
F → ( E ) | number
```

**Leftmost Derivation**:

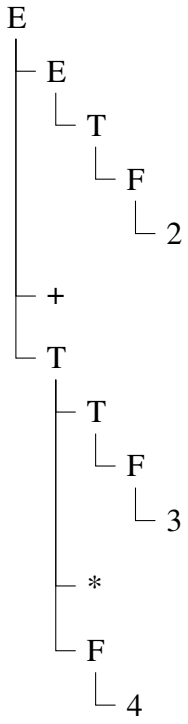
1. $E \Rightarrow E + T$
2. $E + T \Rightarrow T + T$
3. $T + T \Rightarrow F + T$
4. $F + T \Rightarrow 2 + T$
5. $2 + T \Rightarrow 2 + T * F$

6. $2 + T * F \Rightarrow 2 + F * F$

7. $2 + F * F \Rightarrow 2 + 3 * F$

8. $2 + 3 * F \Rightarrow 2 + 3 * 4$

**Parse Tree**:




Gambar 5.11: Parse tree untuk ekspresi `2 + 3 * 4`

## 5.12 Kesimpulan

Dalam bab ini, kita telah mempelajari:

1. Context-free grammar adalah alat formal untuk mendefinisikan syntax bahasa pemrograman
2. BNF dan EBNF adalah notasi untuk menulis grammar
3. Derivation menunjukkan bagaimana string dihasilkan dari grammar
4. Parse tree merepresentasikan struktur sintaksis program
5. Ambiguity dalam grammar dapat menyebabkan interpretasi yang berbeda
6. Left recursion dan left factoring adalah isu penting dalam parsing

Pemahaman tentang CFG dan konsep-konsep terkait ini menjadi dasar penting untuk mempelajari teknik parsing (top-down dan bottom-up) yang akan dibahas dalam bab-bab selanjutnya.

## 5.13 Referensi dan Bahan Bacaan Lanjutan

Untuk memperdalam pemahaman tentang context-free grammar dan parsing, mahasiswa disarankan membaca:

- **Dragon Book**: Aho, Lam, Sethi, & Ullman (2006). *Compilers: Principles, Techniques, and Tools* [1] - Bab 4: Syntax Analysis
- **Engineering a Compiler**: Cooper & Torczon (2011) [5] - Bab 3: Scanners dan Bab 4: Parsers
- **UC San Diego CSE 231**: Course materials tentang syntax analysis [3]
- **Northeastern University CS 4410**: Materials tentang parsing techniques [4]
- **Johns Hopkins University EN.601.428**: Course tentang syntax trees dan parsing [6]

# Bab 6

## Top-Down Parsing dan Recursive Descent

### 6.1 Tujuan Pembelajaran

Setelah mempelajari bab ini, mahasiswa diharapkan mampu:

1. Menjelaskan konsep top-down parsing dan perbedaannya dengan bottom-up parsing
2. Memahami LL parsing dan karakteristiknya
3. Mengimplementasikan recursive descent parser untuk grammar sederhana
4. Menangani precedence dan associativity dalam recursive descent parser
5. Mengimplementasikan error recovery pada recursive descent parser
6. Mengintegrasikan lexer dengan recursive descent parser
7. Mengevaluasi ekspresi aritmatika menggunakan recursive descent parser

### 6.2 Pendahuluan

Pada bab ini kita mengimplementasikan **parser hand-written** (recursive descent) untuk **grammar proyek subset C** (Bab 5, Bagian 5.3). Parser ini mengenali program, statement, declaration, assignment, print-statement, dan ekspresi dengan precedence/associativity yang sama dengan parser proyek di Bab 8 (`simplec.y`), sehingga berfungsi sebagai implementasi manual untuk bahasa yang sama dengan proyek.

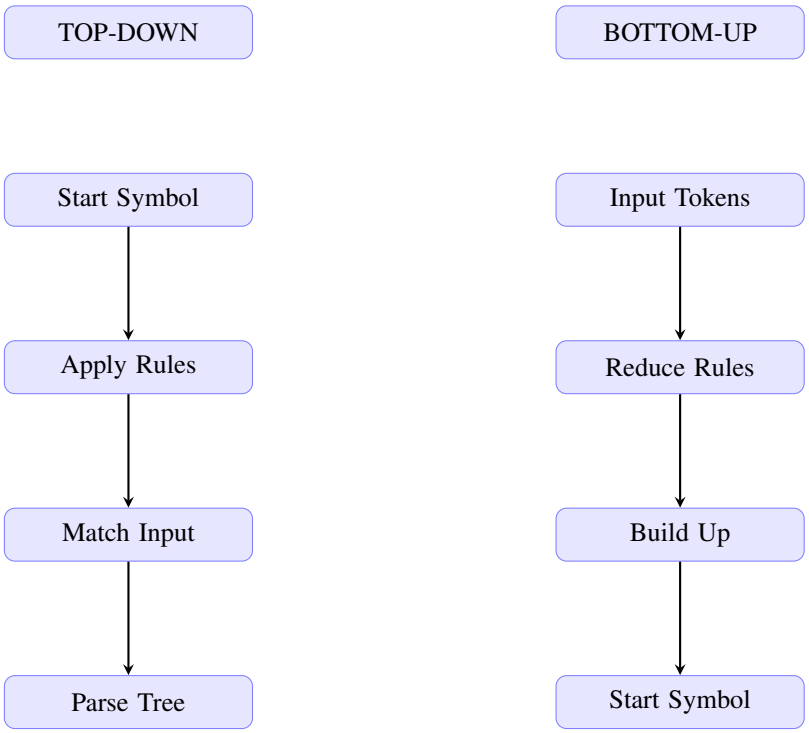
Setelah mempelajari lexical analysis dan context-free grammar pada bab-bab sebelumnya, kita sekarang akan mempelajari bagaimana mengimplementasikan parser yang menganalisis struktur sintaksis dari stream token yang dihasilkan oleh lexer. Top-down parsing adalah salah satu pendekatan yang paling intuitif dan mudah diimplementasikan secara manual.

Menurut sumber terbuka:

> "Top-down parsers (recursive descent) – easy to hand-write; better for LL(1) grammars, when unambiguous. Works by writing functions for grammar nonterminals (e.g. expression(), term(), factor()) that consume tokens one at a time."[8]

Pendekatan top-down parsing dimulai dari start symbol grammar dan mencoba menurunkan (derive) input dengan membangun parse tree dari root ke leaves. Ini berbeda dengan bottom-up parsing yang membangun parse tree dari leaves ke root.

Gambar 6.1 menunjukkan perbandingan top-down dan bottom-up parsing.




Gambar 6.1: Perbandingan top-down dan bottom-up parsing

## 6.3 Konsep Top-Down Parsing

### 6.3.1 Definisi Top-Down Parsing

Top-down parsing adalah teknik parsing yang dimulai dari start symbol grammar dan mencoba menurunkan input dengan menerapkan production rules dari atas ke bawah. Parser mencoba mencocokkan input dengan memprediksi production mana yang harus digunakan berdasarkan lookahead token.

Karakteristik utama top-down parsing:

- Membangun parse tree dari root (start symbol) ke leaves (terminals)

- Menggunakan leftmost derivation
- Memerlukan lookahead untuk memprediksi production yang tepat
- Dapat diimplementasikan secara recursive atau iterative dengan stack

### 6.3.2 LL Parsing

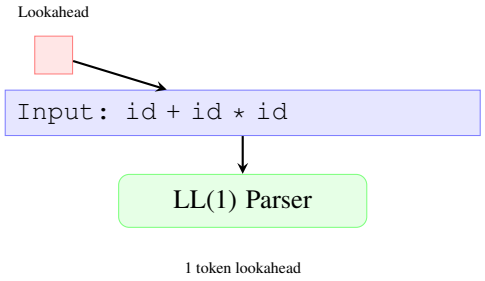
LL parsing adalah kelas top-down parsing yang membaca input dari **L**eft ke right dan menghasilkan **L**eftmost derivation. Notasi LL(k) menunjukkan bahwa parser menggunakan k token lookahead untuk membuat keputusan parsing.

Menurut sumber dari USNA:

> "Top-down parsing starts from the start symbol and tries to rewrite it to match the input, building a parse tree from root to leaves. LL parsing means scanning input Left-to-right, producing a Leftmost derivation, using k-token lookahead (usually LL(1))."[1]

LL(1) adalah yang paling umum digunakan karena hanya memerlukan satu token lookahead, membuat implementasinya lebih sederhana dan efisien.

Gambar 6.2 menunjukkan konsep LL parsing.




Gambar 6.2: Konsep LL parsing dengan lookahead

### 6.3.3 Keuntungan dan Keterbatasan Top-Down Parsing

**Keuntungan:**

- Mudah diimplementasikan secara manual (recursive descent)
- Error messages yang lebih intuitif (dapat menunjukkan posisi error dengan tepat)
- Tidak memerlukan preprocessing grammar yang kompleks (untuk grammar LL(1))

[1] https://www.usna.edu/Users/cs/wcbrown/courses/F20SI413/lec/l09/lec.html

- Cocok untuk grammar yang sudah dalam bentuk yang sesuai

**Keterbatasan:**

- Tidak dapat menangani left recursion secara langsung
- Memerlukan grammar yang sudah di-factoring untuk menghindari ambiguity
- Tidak sekuat LR parsing dalam hal kemampuan parsing
- Beberapa grammar memerlukan transformasi sebelum dapat di-parse dengan top-down

## 6.4 Recursive Descent Parsing

### 6.4.1 Konsep Recursive Descent

Recursive descent parsing adalah teknik implementasi top-down parsing di mana setiap non-terminal dalam grammar direpresentasikan sebagai sebuah fungsi. Fungsi-fungsi ini saling memanggil secara recursive sesuai dengan struktur grammar.

Menurut sumber dari Ernest Chu:

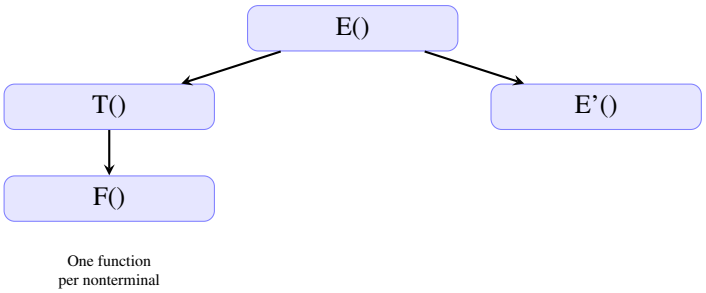
> "Recursive-descent parsing is a hand-written parser (one function per non-terminal), possibly with backtracking. When you eliminate left recursion and factor grammar properly, you can build deterministic predictive parsers (LL(1))—recursive descent without backtracking."[2]

Struktur dasar recursive descent parser:

1. Setiap non-terminal memiliki fungsi sendiri
2. Fungsi membaca token dari input stream
3. Fungsi memanggil fungsi lain sesuai dengan production rules
4. Terminal dicocokkan langsung dengan token saat ini

Gambar 6.3 menunjukkan struktur recursive descent parser.

[2] https://ernestchu.github.io/course-notes/courses/cse360-design-and-implementation-of-compiler/syntax-analysis/top-down-parsing.html




Gambar 6.3: Struktur recursive descent parser

### 6.4.2 Implementasi Dasar

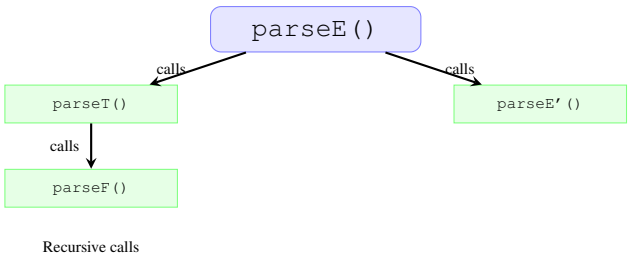
Mari kita lihat contoh implementasi recursive descent parser untuk grammar ekspresi aritmatika sederhana. Grammar yang akan kita gunakan:

```
E  -> T E'
E' -> + T E' | epsilon
T  -> F T'
T' -> * F T' | epsilon
F  -> ( E ) | id | num
```

Grammar ini sudah dalam bentuk yang sesuai untuk LL(1) parsing karena:

- Tidak ada left recursion
- Sudah di-factoring (E' dan T' menangani associativity)
- Setiap production dapat diputuskan dengan satu token lookahead

Gambar 6.4 menunjukkan contoh pemanggilan fungsi dalam recursive descent parser.




Gambar 6.4: Contoh pemanggilan fungsi recursive descent

Implementasi dalam C++:

Listing 6.1: Struktur dasar recursive descent parser

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
```

```
4
5  enum TokenType {
6      TOK_ID, TOK_NUM, TOK_PLUS, TOK_MUL,
7      TOK_LPAREN, TOK_RPAREN, TOK_END, TOK_ERROR
8  };
9
10 struct Token {
11     TokenType type;
12     std::string lexeme;
13     int line, col;
14 };
15
16 class RecursiveDescentParser {
17 private:
18     std::vector<Token> tokens;
19     size_t current;
20     Token lookahead;
21
22     void nextToken() {
23         if (current < tokens.size()) {
24             lookahead = tokens[current++];
25         } else {
26             lookahead = {TOK_END, "", 0, 0};
27         }
28     }
29
30     void match(TokenType expected) {
31         if (lookahead.type == expected) {
32             nextToken();
33         } else {
34             error("Expected " + tokenToString(expected) +
35                   " but got " + lookahead.lexeme);
36         }
37     }
38
39     void error(const std::string& msg) {
40         std::cerr << "Syntax error at line " << lookahead.line
41                   << ", col " << lookahead.col << ": " << msg << std::
   ↪ endl;
42         exit(1);
43     }
44
45 public:
46     RecursiveDescentParser(const std::vector<Token>& t)
47         : tokens(t), current(0) {
48         nextToken();
49     }
50
51     // Grammar: E -> T E'
52     void parseE() {
53         parseT();
54         parseEPrime();
55     }
56
```

```
57      // Grammar: E' -> + T E' | epsilon
58      void parseEPrime() {
59          if (lookahead.type == TOK_PLUS) {
60              match(TOK_PLUS);
61              parseT();
62              parseEPrime();
63          }
64          // else: epsilon production, do nothing
65      }
66
67      // Grammar: T -> F T'
68      void parseT() {
69          parseF();
70          parseTPrime();
71      }
72
73      // Grammar: T' -> * F T' | epsilon
74      void parseTPrime() {
75          if (lookahead.type == TOK_MUL) {
76              match(TOK_MUL);
77              parseF();
78              parseTPrime();
79          }
80          // else: epsilon production, do nothing
81      }
82
83      // Grammar: F -> ( E ) | id | num
84      void parseF() {
85          if (lookahead.type == TOK_LPAREN) {
86              match(TOK_LPAREN);
87              parseE();
88              match(TOK_RPAREN);
89          } else if (lookahead.type == TOK_ID) {
90              match(TOK_ID);
91          } else if (lookahead.type == TOK_NUM) {
92              match(TOK_NUM);
93          } else {
94              error("Expected identifier, number, or '('");
95          }
96      }
97
98      void parse() {
99          parseE();
100         if (lookahead.type != TOK_END) {
101             error("Extra input after expression");
102         }
103         std::cout << "Parse successful!" << std::endl;
104     }
105 };
```

## 6.5 Handling Precedence dan Associativity

### 6.5.1 Konsep Precedence

Precedence menentukan urutan evaluasi operator ketika beberapa operator muncul dalam ekspresi yang sama. Misalnya, dalam ekspresi `a + b * c`, operator `*` memiliki precedence lebih tinggi daripada +, sehingga dievaluasi terlebih dahulu.

Dalam recursive descent parser, precedence di-handle melalui struktur grammar. Operator dengan precedence lebih tinggi berada di level yang lebih dalam dalam parse tree.

### 6.5.2 Handling Associativity

Associativity menentukan bagaimana operator dengan precedence yang sama dievaluasi. Ada dua jenis:

- **Left-associative**: Dievaluasi dari kiri ke kanan, misalnya `a - b - c` = `(a - b) - c`
- **Right-associative**: Dievaluasi dari kanan ke kiri, misalnya `a = b = c` = `a = (b = c)`

Dalam grammar yang kita gunakan, E' dan T' menggunakan left recursion yang diubah menjadi right recursion untuk menangani left associativity dengan benar.

Contoh grammar untuk menangani precedence dan associativity:

```
E  -> T E'         (expression level, lowest precedence)
E' -> + T E' | epsilon  (addition, left-associative)
T  -> F T'         (term level, higher precedence)
T' -> * F T' | epsilon  (multiplication, left-associative)
F  -> ( E ) | id  (factor level, highest precedence)
```

Struktur ini memastikan bahwa:

- Operator `*` memiliki precedence lebih tinggi daripada + (T berada di bawah E)
- Kedua operator left-associative (menggunakan right recursion dengan tail call)
- Parentheses memiliki precedence tertinggi (F)

### 6.5.3 Implementasi dengan Evaluasi

Berikut adalah implementasi recursive descent parser yang tidak hanya mem-parse tetapi juga mengevaluasi ekspresi:

Listing 6.2: Recursive descent parser dengan evaluasi

```
class ExpressionEvaluator {
private:
    std::vector<Token> tokens;
    size_t current;
    Token lookahead;

    void nextToken() {
        if (current < tokens.size()) {
            lookahead = tokens[current++];
        } else {
            lookahead = {TOK_END, "", 0, 0};
        }
    }

    void match(TokenType expected) {
        if (lookahead.type == expected) {
            nextToken();
        } else {
            throw std::runtime_error("Syntax error");
        }
    }

public:
    ExpressionEvaluator(const std::vector<Token>& t)
        : tokens(t), current(0) {
        nextToken();
    }

    // E -> T E'
    // Returns value of expression
    int parseE() {
        int value = parseT();
        return parseEPrime(value);
    }

    // E' -> + T E' | epsilon
    // Accumulates addition operations
    int parseEPrime(int left) {
        if (lookahead.type == TOK_PLUS) {
            match(TOK_PLUS);
            int right = parseT();
            return parseEPrime(left + right);
        }
        return left;  // epsilon production
    }

    // T -> F T'
    int parseT() {
        int value = parseF();
        return parseTPrime(value);
    }

    // T' -> * F T' | epsilon
```

```
    // Accumulates multiplication operations
    int parseTPrime(int left) {
        if (lookahead.type == TOK_MUL) {
            match(TOK_MUL);
            int right = parseF();
            return parseTPrime(left * right);
        }
        return left;  // epsilon production
    }

    // F -> ( E ) | num
    int parseF() {
        if (lookahead.type == TOK_LPAREN) {
            match(TOK_LPAREN);
            int value = parseE();
            match(TOK_RPAREN);
            return value;
        } else if (lookahead.type == TOK_NUM) {
            int value = std::stoi(lookahead.lexeme);
            match(TOK_NUM);
            return value;
        } else {
            throw std::runtime_error("Expected number or '('");
        }
    }

    int evaluate() {
        int result = parseE();
        if (lookahead.type != TOK_END) {
            throw std::runtime_error("Extra input");
        }
        return result;
    }
};
```

## 6.6 Error Recovery pada Recursive Descent

### 6.6.1 Pentingnya Error Recovery

Error recovery adalah kemampuan parser untuk melanjutkan parsing setelah menemukan error, sehingga dapat melaporkan multiple errors dalam satu pass. Tanpa error recovery, parser akan berhenti pada error pertama.

### 6.6.2 Strategi Error Recovery

Beberapa strategi error recovery yang umum digunakan:

### Synchronization Points

Menentukan synchronization points (token-token yang dapat digunakan untuk recovery), seperti:

- Statement terminators (`;`, `}`)
- Keywords yang menandai awal konstruksi baru (`if`, `while`, `return`)
- Operator yang jelas (+, −, ⋆)

### Panic Mode Recovery

Ketika error ditemukan, parser membuang token sampai menemukan synchronization point:

Listing 6.3: Panic mode error recovery

```
void parseE() {
    parseT();
    parseEPrime();
}

void parseEPrime() {
    if (lookahead.type == TOK_PLUS) {
        match(TOK_PLUS);
        parseT();
        parseEPrime();
    } else if (!isValidFollow(lookahead.type)) {
        // Error recovery: skip until synchronization point
        error("Expected '+' or end of expression");
        while (!isSynchronizationPoint(lookahead.type) &&
               lookahead.type != TOK_END) {
            nextToken();
        }
    }
    // else: valid follow token, epsilon production
}

bool isSynchronizationPoint(TokenType t) {
    return t == TOK_RPAREN || t == TOK_END ||
           t == TOK_SEMICOLON;
}

bool isValidFollow(TokenType t) {
    return t == TOK_RPAREN || t == TOK_END ||
           t == TOK_SEMICOLON || t == TOK_PLUS;
}
```

### Error Production

Menambahkan production khusus untuk menangani error:

```
E' -> + T E' | epsilon | error E'
```

Ketika error ditemukan, parser dapat menggunakan error production untuk recovery.

### 6.6.3 Implementasi Error Recovery yang Lebih Baik

Berikut adalah implementasi yang lebih robust dengan error recovery:

Listing 6.4: Error recovery yang lebih baik

```
class ParserWithRecovery {
private:
    int errorCount;
    std::vector<Token> tokens;
    size_t current;
    Token lookahead;

    void nextToken() {
        if (current < tokens.size()) {
            lookahead = tokens[current++];
        } else {
            lookahead = {TOK_END, "", 0, 0};
        }
    }

    void error(const std::string& msg) {
        errorCount++;
        std::cerr << "Error at line " << lookahead.line
                  << ", col " << lookahead.col << ": "
                  << msg << std::endl;
    }

    void synchronize() {
        // Skip tokens until synchronization point
        while (lookahead.type != TOK_END) {
            if (isSynchronizationPoint(lookahead.type)) {
                return;
            }
            nextToken();
        }
    }

    bool isSynchronizationPoint(TokenType t) {
        return t == TOK_RPAREN || t == TOK_SEMICOLON ||
               t == TOK_END || t == TOK_PLUS || t == TOK_MUL;
    }

public:
    ParserWithRecovery(const std::vector<Token>& t)
        : tokens(t), current(0), errorCount(0) {
        nextToken();
    }

    void parseE() {
        try {
            parseT();
```

```
            parseEPrime();
        } catch (...) {
            error("Error in expression");
            synchronize();
        }
    }

    void parseEPrime() {
        if (lookahead.type == TOK_PLUS) {
            match(TOK_PLUS);
            parseT();
            parseEPrime();
        } else if (!isValidFollow(lookahead.type)) {
            error("Expected '+' or end of expression");
            synchronize();
        }
    }

    int getErrorCount() const { return errorCount; }
};
```

## 6.7 Integrasi Lexer dengan Parser

### 6.7.1 Arsitektur Integrasi

Dalam implementasi praktis, lexer dan parser bekerja bersama dalam pipeline:

```
Source Code -> Lexer -> Token Stream -> Parser -> Parse Tree/AST
```

Lexer membaca source code karakter demi karakter dan menghasilkan stream token. Parser kemudian membaca token dari stream ini.

### 6.7.2 Implementasi Terintegrasi

Berikut adalah contoh implementasi lexer dan parser yang terintegrasi:

Listing 6.5: Lexer dan parser terintegrasi

```
#include <iostream>
#include <string>
#include <cctype>

class IntegratedLexerParser {
private:
    std::string input;
    size_t pos;
    int line, col;
    Token lookahead;

    // Lexer functions
    void skipWhitespace() {
        while (pos < input.size() && isspace(input[pos])) {
```

```
            if (input[pos] == '\n') {
                line++;
                col = 1;
            } else {
                col++;
            }
            pos++;
        }
    }

    Token nextToken() {
        skipWhitespace();

        if (pos >= input.size()) {
            return {TOK_END, "", line, col};
        }

        char c = input[pos];
        int startCol = col;

        // Identifier: [a-zA-Z][a-zA-Z0-9]*
        if (isalpha(c)) {
            std::string lexeme;
            while (pos < input.size() && isalnum(input[pos])) {
                lexeme += input[pos++];
                col++;
            }
            return {TOK_ID, lexeme, line, startCol};
        }

        // Number: [0-9]+
        if (isdigit(c)) {
            std::string lexeme;
            while (pos < input.size() && isdigit(input[pos])) {
                lexeme += input[pos++];
                col++;
            }
            return {TOK_NUM, lexeme, line, startCol};
        }

        // Operators and punctuation
        pos++;
        col++;
        switch (c) {
            case '+': return {TOK_PLUS, "+", line, startCol};
            case '*': return {TOK_MUL, "*", line, startCol};
            case '(': return {TOK_LPAREN, "(", line, startCol};
            case ')': return {TOK_RPAREN, ")", line, startCol};
            default: return {TOK_ERROR, std::string(1, c), line, startCol
↪ };
        }
    }

    void match(TokenType expected) {
```

```
68          if (lookahead.type == expected) {
69              lookahead = nextToken();
70          } else {
71              std::cerr << "Error: Expected " << expected
72                        << " but got " << lookahead.lexeme
73                        << " at line " << lookahead.line
74                        << ", col " << lookahead.col << std::endl;
75              exit(1);
76          }
77      }
78
79  public:
80      IntegratedLexerParser(const std::string& s)
81          : input(s), pos(0), line(1), col(1) {
82          lookahead = nextToken();
83      }
84
85      // Parser functions (same as before)
86      void parseE() {
87          parseT();
88          parseEPrime();
89      }
90
91      void parseEPrime() {
92          if (lookahead.type == TOK_PLUS) {
93              match(TOK_PLUS);
94              parseT();
95              parseEPrime();
96          }
97      }
98
99      void parseT() {
100         parseF();
101         parseTPrime();
102     }
103
104     void parseTPrime() {
105         if (lookahead.type == TOK_MUL) {
106             match(TOK_MUL);
107             parseF();
108             parseTPrime();
109         }
110     }
111
112     void parseF() {
113         if (lookahead.type == TOK_LPAREN) {
114             match(TOK_LPAREN);
115             parseE();
116             match(TOK_RPAREN);
117         } else if (lookahead.type == TOK_ID) {
118             match(TOK_ID);
119         } else if (lookahead.type == TOK_NUM) {
120             match(TOK_NUM);
121         } else {
```

```
            std::cerr << "Error: Expected id, num, or '('" << std::endl;
            exit(1);
        }
    }

    void parse() {
        parseE();
        if (lookahead.type != TOK_END) {
            std::cerr << "Error: Extra input" << std::endl;
            exit(1);
        }
        std::cout << "Parse successful!" << std::endl;
    }
};
```

## 6.8 Contoh Lengkap: Parser untuk Ekspresi Aritmatika

Berikut adalah contoh lengkap implementasi recursive descent parser untuk ekspresi aritmatika yang dapat menangani:

- Operasi penjumlahan dan perkalian
- Precedence (perkalian lebih tinggi dari penjumlahan)
- Left associativity
- Parentheses
- Identifier dan literal angka
- Error reporting yang informatif

Listing 6.6: Parser lengkap untuk ekspresi aritmatika

```
#include <iostream>
#include <string>
#include <vector>
#include <cctype>
#include <stdexcept>

enum TokenType {
    TOK_ID, TOK_NUM, TOK_PLUS, TOK_MINUS, TOK_MUL, TOK_DIV,
    TOK_LPAREN, TOK_RPAREN, TOK_END, TOK_ERROR
};

struct Token {
    TokenType type;
    std::string lexeme;
    int line, col;

    Token(TokenType t, const std::string& l, int ln, int c)
```

```
        : type(t), lexeme(l), line(ln), col(c) {}
};

class ArithmeticParser {
private:
    std::string input;
    size_t pos;
    int line, col;
    Token lookahead;
    std::vector<std::string> errors;

    void skipWhitespace() {
        while (pos < input.size() && isspace(input[pos])) {
            if (input[pos] == '\n') {
                line++;
                col = 1;
            } else {
                col++;
            }
            pos++;
        }
    }

    Token nextToken() {
        skipWhitespace();

        if (pos >= input.size()) {
            return Token(TOK_END, "", line, col);
        }

        char c = input[pos];
        int startLine = line, startCol = col;

        // Identifier
        if (isalpha(c) || c == '_') {
            std::string lexeme;
            while (pos < input.size() &&
                   (isalnum(input[pos]) || input[pos] == '_')) {
                lexeme += input[pos++];
                col++;
            }
            return Token(TOK_ID, lexeme, startLine, startCol);
        }

        // Number
        if (isdigit(c)) {
            std::string lexeme;
            while (pos < input.size() && isdigit(input[pos])) {
                lexeme += input[pos++];
                col++;
            }
            return Token(TOK_NUM, lexeme, startLine, startCol);
        }

```

```
72          // Operators
73          pos++;
74          col++;
75          switch (c) {
76              case '+': return Token(TOK_PLUS, "+", startLine, startCol);
77              case '-': return Token(TOK_MINUS, "-", startLine, startCol);
78              case '*': return Token(TOK_MUL, "*", startLine, startCol);
79              case '/': return Token(TOK_DIV, "/", startLine, startCol);
80              case '(': return Token(TOK_LPAREN, "(", startLine, startCol);
81              case ')': return Token(TOK_RPAREN, ")", startLine, startCol);
82              default:
83                  return Token(TOK_ERROR, std::string(1, c), startLine,
↪ startCol);
84          }
85      }
86
87      void match(TokenType expected) {
88          if (lookahead.type == expected) {
89              lookahead = nextToken();
90          } else {
91              std::string msg = "Expected " + tokenToString(expected) +
92                                " but got " + lookahead.lexeme +
93                                " at line " + std::to_string(lookahead.line)
↪ +
94                                ", col " + std::to_string(lookahead.col);
95              errors.push_back(msg);
96              throw std::runtime_error(msg);
97          }
98      }
99
100     std::string tokenToString(TokenType t) {
101         switch (t) {
102             case TOK_ID: return "identifier";
103             case TOK_NUM: return "number";
104             case TOK_PLUS: return "'+'";
105             case TOK_MINUS: return "'-'";
106             case TOK_MUL: return "'*'";
107             case TOK_DIV: return "'/'";
108             case TOK_LPAREN: return "'('";
109             case TOK_RPAREN: return "')'";
110             case TOK_END: return "end of input";
111             default: return "unknown";
112         }
113     }
114
115 public:
116     ArithmeticParser(const std::string& s)
117         : input(s), pos(0), line(1), col(1) {
118         lookahead = nextToken();
119     }
120
121     // E -> T E'
122     void parseE() {
123         parseT();
```

```
124         parseEPrime();
125     }
126
127     // E' -> (+ | -) T E' | epsilon
128     void parseEPrime() {
129         if (lookahead.type == TOK_PLUS || lookahead.type == TOK_MINUS) {
130             TokenType op = lookahead.type;
131             match(op);
132             parseT();
133             parseEPrime();
134         }
135     }
136
137     // T -> F T'
138     void parseT() {
139         parseF();
140         parseTPrime();
141     }
142
143     // T' -> (* | /) F T' | epsilon
144     void parseTPrime() {
145         if (lookahead.type == TOK_MUL || lookahead.type == TOK_DIV) {
146             TokenType op = lookahead.type;
147             match(op);
148             parseF();
149             parseTPrime();
150         }
151     }
152
153     // F -> ( E ) | id | num
154     void parseF() {
155         if (lookahead.type == TOK_LPAREN) {
156             match(TOK_LPAREN);
157             parseE();
158             match(TOK_RPAREN);
159         } else if (lookahead.type == TOK_ID) {
160             match(TOK_ID);
161         } else if (lookahead.type == TOK_NUM) {
162             match(TOK_NUM);
163         } else {
164             std::string msg = "Expected identifier, number, or '(' at
↪ line " +
165                               std::to_string(lookahead.line) +
166                               ", col " + std::to_string(lookahead.col);
167             errors.push_back(msg);
168             throw std::runtime_error(msg);
169         }
170     }
171
172     bool parse() {
173         try {
174             parseE();
175             if (lookahead.type != TOK_END) {
176                 errors.push_back("Extra input after expression");
```

```
                return false;
            }
            return errors.empty();
        } catch (...) {
            return false;
        }
    }

    void printErrors() {
        for (const auto& err : errors) {
            std::cerr << err << std::endl;
        }
    }
};

int main() {
    std::string input;
    std::cout << "Enter expression: ";
    std::getline(std::cin, input);

    ArithmeticParser parser(input);
    if (parser.parse()) {
        std::cout << "Parse successful!" << std::endl;
    } else {
        std::cout << "Parse failed!" << std::endl;
        parser.printErrors();
    }

    return 0;
}
```

## 6.9 Kesimpulan

Dalam bab ini, kita telah mempelajari:

1. Top-down parsing adalah teknik parsing yang membangun parse tree dari root ke leaves
2. LL parsing adalah kelas top-down parsing yang membaca input left-to-right dan menghasilkan leftmost derivation
3. Recursive descent parsing adalah implementasi top-down parsing di mana setiap non-terminal direpresentasikan sebagai fungsi
4. Precedence dan associativity di-handle melalui struktur grammar yang tepat
5. Error recovery memungkinkan parser untuk melanjutkan setelah menemukan error
6. Lexer dan parser dapat diintegrasikan dalam satu implementasi yang efisien

Recursive descent parsing adalah teknik yang sangat cocok untuk implementasi manual parser karena mudah dipahami dan diimplementasikan. Namun, perlu diingat bahwa grammar harus dalam bentuk yang sesuai (tanpa left recursion, sudah di-factoring) untuk dapat di-parse dengan pendekatan ini. Parser yang dibahas di bab ini mengimplementasikan grammar proyek subset C (Bab 5); parser proyek dengan Bison dibangun di Bab 8 (`simplec.y`).

## 6.10 Referensi dan Bahan Bacaan Lanjutan

Untuk memperdalam pemahaman tentang top-down parsing dan recursive descent, mahasiswa disarankan membaca:

- **Dragon Book**: Aho, Lam, Sethi, & Ullman (2006). *Compilers: Principles, Techniques, and Tools* [1] - Bab 4: Syntax Analysis, Section 4.4: Top-Down Parsing
- **Engineering a Compiler**: Cooper & Torczon (2011) [5] - Bab 3: Scanners, Section 3.4: Top-Down Parsing
- **OpenGenus Tutorial**: Build Lexer [8] - Bagian tentang recursive descent parsing
- **USNA Course Notes**: Top-Down Parsing [3]
- **Ernest Chu Course Notes**: Syntax Analysis - Top-Down Parsing [4]
- **TutorialsPoint**: Compiler Design - Top Down Parser [5]

[3] https://www.usna.edu/Users/cs/wcbrown/courses/F20SI413/lec/l09/lec.html

[4] https://ernestchu.github.io/course-notes/courses/cse360-design-and-implementation-of-compiler/syntax-analysis/top-down-parsing.html

[5] https://www.tutorialspoint.com/compiler_design/compiler_design_top_down_parser.htm

# Bab 7

## Bottom-Up Parsing, LR Parser, dan Parser Generator

### 7.1 Tujuan Pembelajaran

Setelah mempelajari bab ini, mahasiswa diharapkan mampu:

1. Menjelaskan konsep bottom-up parsing dan perbedaannya dengan top-down parsing
2. Memahami shift-reduce parsing dan operasi-operasinya
3. Menjelaskan berbagai jenis LR parser (LR(0), SLR(1), CLR(1), LALR(1))
4. Memahami konstruksi LR parsing table untuk grammar sederhana
5. Menggunakan parser generator (Bison/Yacc) untuk membuat parser
6. Mengintegrasikan Flex lexer dengan Bison parser
7. Menambahkan semantic actions untuk membangun AST
8. Mengimplementasikan error handling dalam parser generator

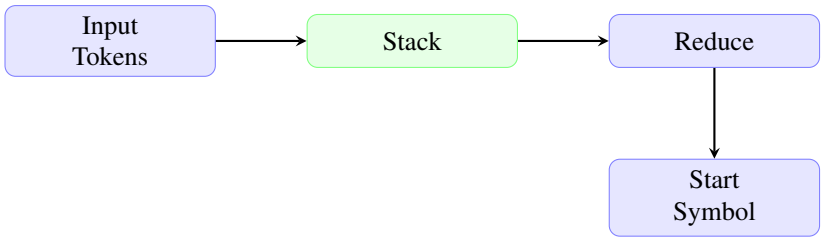
### 7.2 Pendahuluan

Setelah mempelajari top-down parsing pada bab sebelumnya, kita sekarang akan mempelajari pendekatan alternatif yang lebih powerful: bottom-up parsing. Menurut sumber terbuka:

> "Bottom-up parsers (LR, LALR, GLR) – more powerful; often generated by tools like Bison/Yacc. The choice affects ease of specification and parsing power."[2]

Bottom-up parsing membangun parse tree dari leaves (token) ke root (start symbol), yang merupakan kebalikan dari top-down parsing. Pendekatan ini lebih powerful karena dapat menangani lebih banyak jenis grammar, termasuk grammar dengan left recursion yang tidak dapat ditangani langsung oleh top-down parser.

Gambar 7.1 menunjukkan alur bottom-up parsing.




Gambar 7.1: Alur bottom-up parsing

## 7.3 Konsep Bottom-Up Parsing

### 7.3.1 Definisi Bottom-Up Parsing

Bottom-up parsing adalah teknik parsing yang dimulai dari input tokens dan mencoba membangun parse tree dari bawah ke atas, dengan tujuan mencapai start symbol. Parser menggunakan rightmost derivation dalam reverse, yaitu membangun derivation dari kanan ke kiri.

Karakteristik utama bottom-up parsing:

- Membangun parse tree dari leaves (terminals) ke root (start symbol)
- Menggunakan rightmost derivation dalam reverse
- Menggunakan stack untuk menyimpan state parsing
- Lebih powerful daripada top-down parsing (dapat menangani lebih banyak grammar)
- Umumnya diimplementasikan menggunakan parsing table yang di-generate

### 7.3.2 Handle dan Reduction

Konsep penting dalam bottom-up parsing adalah **handle**. Handle adalah substring dari sentential form saat ini yang cocok dengan right-hand side (RHS) dari suatu production rule, dan reduction terhadap handle ini akan membawa kita lebih dekat ke start symbol.

Menurut definisi formal:

> "A handle is a substring of the current sentential form that matches the RHS of a production and whose reduction must lead toward the start symbol."[1]

Contoh: Jika kita memiliki grammar:

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

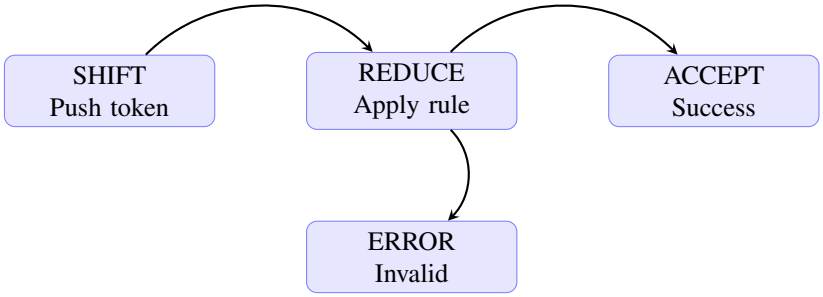
[1] https://ebooks.inflibnet.ac.in/csp10/chapter/top-down-parser-parsing-tableshift-reduce-parser/

Dan sentential form saat ini adalah `id + id * id`, maka handle yang tepat adalah `id` (yang dapat di-reduce menjadi F, kemudian T, kemudian E).

## 7.4 Shift-Reduce Parsing

### 7.4.1 Konsep Shift-Reduce

Gambar 7.2 menunjukkan empat operasi dasar dalam shift-reduce parsing.




Gambar 7.2: Operasi-operasi dalam shift-reduce parsing

Shift-reduce parsing adalah implementasi dasar dari bottom-up parsing yang menggunakan stack dan empat operasi dasar. Menurut GeeksforGeeks:

> "Shift-Reduce Parser uses a stack and four basic operations: 1. Shift: push the next input symbol onto the stack. 2. Reduce: when the top of stack matches RHS of some grammar rule, pop it and push the LHS nonterminal. 3. Accept: if the stack has start symbol and input is exhausted. 4. Error: no valid shift/reduce possible."[2]

### 7.4.2 Operasi Shift

Operasi **shift** memindahkan token berikutnya dari input ke stack. Ini dilakukan ketika parser belum menemukan handle yang lengkap di stack.

Contoh: Jika stack berisi `[E, +]` dan input berikutnya adalah `id`, maka operasi shift akan menghasilkan stack `[E, +, id]`.

### 7.4.3 Operasi Reduce

Operasi **reduce** mengganti handle di top of stack dengan left-hand side (LHS) dari production rule yang sesuai. Handle harus cocok persis dengan RHS dari suatu production.

[2] https://www.geeksforgeeks.org/compiler-design/shift-reduce-parser-compiler/

Contoh: Jika stack berisi `[E, +, T, *, F]` dan kita memiliki production `F -> id`, dan top of stack adalah `id` yang cocok dengan RHS, maka reduce akan menghasilkan stack `[E, +, T, *, F]`.

### 7.4.4 Operasi Accept

Operasi **accept** terjadi ketika:

- Stack hanya berisi start symbol (atau augmented start symbol)
- Input sudah habis (hanya end marker $ tersisa)

Ini menandakan bahwa parsing berhasil dan input valid.

### 7.4.5 Operasi Error

Operasi **error** terjadi ketika tidak ada operasi shift atau reduce yang valid. Ini berarti input tidak valid menurut grammar.

### 7.4.6 Contoh Shift-Reduce Parsing

Mari kita lihat contoh parsing ekspresi `id + id` dengan grammar sederhana:

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

| Stack | Input | Action | Production |
|---|---|---|---|
| $ | id + id $ | Shift | |
| $ id | + id $ | Reduce | T -> id |
| $ T | + id $ | Reduce | E -> T |
| $ E | + id $ | Shift | |
| $ E + | id $ | Shift | |
| $ E + id | $ | Reduce | T -> id |
| $ E + T | $ | Reduce | E -> E + T |
| $ E | $ | Accept | |

Tabel 7.1: Contoh shift-reduce parsing untuk `id + id`

## 7.5 LR Parsers

### 7.5.1 Definisi LR Parser

LR parser adalah kelas bottom-up parser yang membaca input dari **L**eft ke right dan menghasilkan **R**ightmost derivation dalam reverse. Notasi LR(k) menunjukkan bahwa parser menggunakan k token lookahead.

Menurut GeeksforGeeks:

> "LR parsers read input Left-to-right and produce a Rightmost derivation in reverse. They use a parsing table to decide when to shift and when to reduce."[3]

LR parser menggunakan dua tabel utama:

- **Action Table**: Menentukan aksi (shift, reduce, accept, error) berdasarkan state saat ini dan lookahead token
- **GOTO Table**: Menentukan state berikutnya setelah reduce berdasarkan state saat ini dan non-terminal yang dihasilkan

### 7.5.2 Jenis-jenis LR Parser

Terdapat beberapa varian LR parser, masing-masing dengan karakteristik berbeda:

#### LR(0)

LR(0) adalah varian paling sederhana yang tidak menggunakan lookahead. Karakteristik:

- Tidak memerlukan lookahead token
- Tabel parsing kecil
- Sangat terbatas dalam kemampuan parsing (banyak grammar menghasilkan conflict)
- Jarang digunakan dalam praktik

#### SLR(1) - Simple LR

SLR(1) menggunakan 1 token lookahead dan Follow sets untuk menentukan kapan melakukan reduce. Karakteristik:

- Menggunakan LR(0) item sets
- Reduce hanya dilakukan jika lookahead token berada dalam Follow set dari non-terminal yang di-reduce
- Lebih powerful daripada LR(0)
- Tabel lebih kecil daripada CLR(1)
- Masih dapat menghasilkan conflict untuk beberapa grammar

[3] https://www.geeksforgeeks.org/bottom-up-or-shift-reduce-parsers-set-2/

Menurut GeeksforGeeks:

> "SLR(1) uses LR(0) item sets, and reduction is allowed on lookahead symbols in Follow(A) for production A -> $\alpha$ when the item [A -> $\alpha$ •] appears in the state. This can lead to conflicts CLR(1) avoids."[4]

### CLR(1) - Canonical LR

CLR(1) adalah varian paling powerful yang menggunakan full LR(1) items dengan lookahead spesifik. Karakteristik:

- Menggunakan LR(1) items (production dengan lookahead spesifik)
- Reduce hanya dilakukan pada lookahead token yang spesifik
- Dapat menangani lebih banyak grammar daripada SLR(1)
- Tabel parsing sangat besar (banyak states)
- Lebih lambat dalam konstruksi tabel

### LALR(1) - Look-Ahead LR

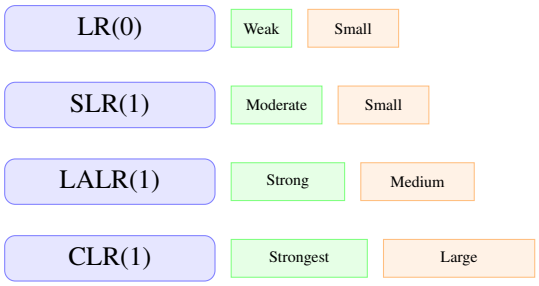
LALR(1) adalah kompromi praktis antara SLR(1) dan CLR(1). Karakteristik:

- Merge states dari CLR(1) yang memiliki LR(0) core yang sama
- Menggabungkan lookahead sets dari states yang di-merge
- Jumlah states sama atau mendekati SLR(1)
- Lebih powerful daripada SLR(1), hampir sekuat CLR(1)
- Digunakan oleh Yacc dan Bison (parser generator populer)

Menurut GeeksforGeeks:

> "LALR(1) merges states in the CLR(1) automaton that have identical LR(0) cores (i.e. same productions & dot positions), combining their lookahead sets. Then construct table using merged states. This reduces size while often preserving correctness."[5]

Gambar 7.3 menunjukkan perbandingan visual varian LR parser.




Gambar 7.3: Perbandingan varian LR parser: power vs table size

| Variant | Lookahead | Table Size | Parsing Power |
|---|---|---|---|
| LR(0) | None | Smallest | Weakest |
| SLR(1) | 1 token (Follow sets) | Small | Moderate |
| LALR(1) | 1 token (merged) | Small-Medium | Strong |
| CLR(1) | 1 token (full) | Largest | Strongest |

Tabel 7.2: Perbandingan varian LR parser

### 7.5.3 Perbandingan LR Parser Variants

## 7.6 Konstruksi LR Parsing Table

### 7.6.1 Augmented Grammar

Langkah pertama dalam konstruksi LR parsing table adalah membuat **augmented grammar**. Kita menambahkan production baru:

```
S' -> S
```

di mana S adalah start symbol asli. Ini memungkinkan state accept yang unambiguous.

### 7.6.2 LR Items

LR item adalah production dengan dot (•) yang menandai posisi parsing saat ini. Format: `A -> ` $\alpha$ • $\beta$

Contoh:

- `E -> • E + T`: Belum membaca apapun dari production ini
- `E -> E • + T`: Sudah membaca E, menunggu +
- `E -> E + • T`: Sudah membaca E dan +, menunggu T
- `E -> E + T •`: Sudah membaca seluruh RHS, siap untuk reduce

[4] https://www.geeksforgeeks.org/bottom-up-or-shift-reduce-parsers-set-2/

[5] https://www.geeksforgeeks.org/compiler-design/lalr-parser-with-examples/

**LR(0) Items**

LR(0) item hanya berisi production dengan dot, tanpa informasi lookahead.

**LR(1) Items**

LR(1) item adalah LR(0) item yang ditambahkan dengan lookahead token. Format: `[A -> α • β, a]` di mana `a` adalah lookahead token.

### 7.6.3 Closure Operation

Closure operation menambahkan semua production yang relevan ke set items. Jika kita memiliki item `[A -> α • B β]` dalam set, kita menambahkan semua items `[B -> • γ]` untuk setiap production `B -> γ`.

Algoritma closure:

1. Mulai dengan set items awal
2. Untuk setiap item `[A -> α • B β]`:
    - Tambahkan semua items `[B -> • γ]` untuk setiap production `B -> γ`
    - Jika LR(1), hitung lookahead untuk items baru
3. Ulangi sampai tidak ada item baru yang ditambahkan

### 7.6.4 GOTO Operation

GOTO operation memindahkan dot melewati simbol grammar X. Jika kita memiliki item `[A -> α • X β]` dan membaca X, kita mendapatkan item `[A -> α X • β]`.

Algoritma GOTO:

1. Mulai dengan set items I dan simbol grammar X
2. Untuk setiap item `[A -> α • X β]` dalam I:
    - Tambahkan `[A -> α X • β]` ke set baru
3. Ambil closure dari set baru

### 7.6.5 Canonical Collection of Item Sets

Canonical collection adalah kumpulan semua state yang mungkin dalam LR automaton. Konstruksinya:

1. Mulai dengan `I_0 = closure({S' -> • S, $})`

2. Untuk setiap state I dan setiap simbol grammar X:
    - Hitung `GOTO(I, X)`
    - Jika hasilnya non-empty dan belum ada, tambahkan sebagai state baru
3. Ulangi sampai tidak ada state baru

### 7.6.6 Konstruksi Action dan GOTO Tables

Setelah canonical collection dibuat, kita konstruksi dua tabel:

#### Action Table

Action table menentukan aksi berdasarkan state dan lookahead token:

- **Shift**: Jika `GOTO(I, a) = J` untuk terminal `a`, maka `action[I, a] = shift J`
- **Reduce**: Jika item `[A -> α •, a]` ada di state I, maka `action[I, a] = reduce A -> α`
- **Accept**: Jika item `[S' -> S •, $]` ada di state I, maka `action[I, $] = accept`
- **Error**: Jika tidak ada aksi yang valid

#### GOTO Table

GOTO table menentukan state berikutnya setelah reduce:

- Jika `GOTO(I, A) = J` untuk non-terminal `A`, maka `goto[I, A] = J`

### 7.6.7 Contoh Konstruksi Parsing Table (Simplified)

Mari kita lihat contoh sederhana untuk grammar:

```
S -> A A
A -> a A | b
```

Augmented grammar:

```
S' -> S
S -> A A
A -> a A | b
```

Langkah-langkah konstruksi (disederhanakan):

1. Buat I_0 dengan closure dari `S' -> • S`

2. Hitung GOTO untuk setiap simbol
3. Lanjutkan sampai semua state ditemukan
4. Konstruksi action dan goto tables

## 7.7 GLR Parsing (Generalized LR)

### 7.7.1 Konsep GLR

GLR (Generalized LR) adalah ekstensi dari LR parsing yang dapat menangani ambiguous grammar atau grammar yang akan menghasilkan conflict dalam tabel LR biasa.

Menurut Wikipedia:

> "GLR extends LR parsing to handle ambiguous grammars or grammars that would cause conflicts in LR tables. It allows multiple possible parse actions in a state and pursues them in parallel."[6]

GLR parser menjaga multiple stacks atau parse trees aktif secara bersamaan ketika terjadi conflict, dan merge stack prefixes yang mungkin untuk berbagi pekerjaan.

### 7.7.2 Kapan Menggunakan GLR

GLR parsing berguna untuk:

- Grammar yang ambiguous (memiliki multiple parse trees valid)
- Grammar yang tidak LR(1) tetapi masih ingin di-parse secara deterministik
- Bahasa dengan syntax yang extensible
- Natural language processing

## 7.8 Parser Generator: Bison dan Yacc

### 7.8.1 Pengenalan Parser Generator

Parser generator adalah tool yang secara otomatis menghasilkan parser dari specification grammar. Menurut sumber dari IT Trip:

> "Bison / YACC: define grammar in a .y file, specify %token s, grammar rules, actions, etc. Generates C parser (or C++ variants). Flex + Bison: use Flex to build the lexer (.l file), Bison for parser, integrate them via tokens."[11]

[6] https://en.wikipedia.org/wiki/GLR_parser

Keuntungan menggunakan parser generator:

- Menghemat waktu development
- Mengurangi kemungkinan error
- Mudah di-maintain (ubah grammar, regenerate parser)
- Menghasilkan parser yang efisien
- Mendukung semantic actions untuk membangun AST

### 7.8.2 Yacc (Yet Another Compiler Compiler)

Yacc adalah parser generator yang dikembangkan di Bell Labs pada tahun 1970-an. Yacc menghasilkan LALR(1) parser dari grammar specification.

### 7.8.3 Bison (GNU Yacc)

Bison adalah implementasi open source dari Yacc yang dikembangkan oleh GNU Project. Bison lebih powerful dan memiliki fitur tambahan:

- Mendukung LALR(1), LR(1), dan GLR parsing
- Mendukung C++ output
- Error recovery yang lebih baik
- Dokumentasi yang lebih lengkap

### 7.8.4 Struktur File Bison (.y)

File Bison memiliki struktur berikut:

```
%{
/* C/C++ code: includes, declarations */
%}

/* Bison declarations: tokens, types, precedence */
%token NUMBER IDENTIFIER
%left '+' '-'
%left '*' '/'

%%
/* Grammar rules */
expression:
    expression '+' term { /* semantic action */ }
```

```
    | term
    ;

term:
    term '*' factor { /* semantic action */ }
    | factor
    ;

factor:
    NUMBER { /* semantic action */ }
    | IDENTIFIER { /* semantic action */ }
    | '(' expression ')' { /* semantic action */ }
    ;

%%
/* User code: helper functions */
```

### 7.8.5 Integrasi Flex dan Bison

Flex dan Bison dirancang untuk bekerja bersama:

1. **Flex file (.l)**: Mendefinisikan token patterns

```
%{
#include "parser.tab.h"  // Generated by Bison
%}
%%
[0-9]+     { yylval = atoi(yytext); return NUMBER; }
[a-zA-Z]+  { return IDENTIFIER; }
\+         { return '+'; }
\*         { return '*'; }
%%
```

2. **Bison file (.y)**: Mendefinisikan grammar dan semantic actions

```
%token NUMBER IDENTIFIER
%%
expression: expression '+' term | term;
term: term '*' factor | factor;
factor: NUMBER | IDENTIFIER | '(' expression ')';
%%
```

3. **Compilation**:

```
flex lexer.l
bison -d parser.y
gcc lex.yy.c parser.tab.c -o parser
```

### 7.8.6 Semantic Actions

Semantic actions adalah kode C/C++ yang dieksekusi ketika production rule di-reduce. Actions dapat:

- Membangun AST nodes
- Mengevaluasi ekspresi
- Memvalidasi semantik
- Menghasilkan output

Contoh semantic action untuk membangun AST:

```
expression:
    expression '+' term
    {
        $$ = create_binary_op(PLUS, $1, $3);
    }
    | term
    {
        $$ = $1;
    }
    ;
```

Di mana:

- `$$`: Nilai yang dihasilkan oleh production (LHS)
- `$1, $2, ...`: Nilai dari simbol-simbol di RHS

### 7.8.7 Error Handling dalam Bison

Bison menyediakan mekanisme error handling:

```
%error-verbose  // Better error messages

expression:
    expression '+' term
    | error '+' term  // Error recovery: skip until '+'
    | term
    ;
```

Error recovery rules memungkinkan parser untuk:

- Mendeteksi error
- Melakukan recovery (skip tokens sampai synchronization point)
- Melanjutkan parsing
- Menghasilkan multiple error messages

## 7.9 Perbandingan Top-Down vs Bottom-Up Parsing

### 7.9.1 Perbandingan Karakteristik

| Aspek | Top-Down | Bottom-Up |
|---|---|---|
| Parse Tree Direction | Root -> Leaves | Leaves -> Root |
| Derivation | Leftmost | Rightmost (reverse) |
| Implementation | Recursive descent | Table-driven |
| Lookahead | Usually 1 token | Usually 1 token |
| Left Recursion | Problem | No problem |
| Right Recursion | No problem | Less efficient |
| Parsing Power | LL(1) grammars | LR(1) grammars |
| Error Detection | Early | Later |
| Error Messages | More intuitive | Less intuitive |
| Table Size | Small | Larger |

Tabel 7.3: Perbandingan top-down dan bottom-up parsing

### 7.9.2 Kapan Menggunakan Masing-masing

**Gunakan Top-Down Parsing jika:**

- Grammar sudah dalam bentuk yang sesuai (tidak ada left recursion)
- Error messages yang intuitif penting
- Implementasi manual diperlukan
- Grammar relatif sederhana

**Gunakan Bottom-Up Parsing jika:**

- Grammar memiliki left recursion
- Parsing power yang lebih besar diperlukan
- Menggunakan parser generator (Bison/Yacc)
- Grammar kompleks dengan banyak precedence levels

## 7.10 Kesimpulan

Dalam bab ini, kita telah mempelajari:

1. Bottom-up parsing membangun parse tree dari leaves ke root menggunakan rightmost derivation dalam reverse

2. Shift-reduce parsing menggunakan empat operasi: shift, reduce, accept, dan error
3. LR parser adalah kelas bottom-up parser yang powerful dengan berbagai varian (LR(0), SLR(1), CLR(1), LALR(1))
4. Konstruksi LR parsing table melibatkan augmented grammar, LR items, closure, GOTO, dan canonical collection
5. Parser generator seperti Bison/Yacc secara otomatis menghasilkan parser dari grammar specification
6. Integrasi Flex dan Bison memungkinkan pembangunan lexer dan parser yang terintegrasi
7. Semantic actions memungkinkan pembangunan AST selama parsing
8. Bottom-
   up parsing lebih powerful tetapi top-down parsing lebih mudah diimplementasikan secara manual

Pemahaman tentang bottom-up parsing dan parser generator ini penting untuk mengimplementasikan parser yang robust dan efisien untuk bahasa pemrograman yang kompleks.

## 7.11 Referensi dan Bahan Bacaan Lanjutan

Untuk memperdalam pemahaman tentang bottom-up parsing dan parser generator, mahasiswa disarankan membaca:

- **Dragon Book**: Aho, Lam, Sethi, & Ullman (2006). *Compilers: Principles, Techniques, and Tools* [1] - Bab 4: Syntax-Directed Translation, Bab 5: Bottom-Up Parsers
- **Engineering a Compiler**: Cooper & Torczon (2011) [5] - Bab 3: Scanners and Parsers
- **flex & bison**: Levine (2009) [10] - Buku lengkap tentang Flex dan Bison
- **GeeksforGeeks**: Tutorial tentang shift-reduce parsing dan LR parsers[7]
- **IT Trip**: Tutorial tentang integrasi Flex dan Bison [11]
- **UC San Diego CSE 231**: Course materials tentang parser construction [3]
- **Northeastern University CS 4410**: Comprehensive compiler design course [4]

[7] https://www.geeksforgeeks.org/compiler-design/shift-reduce-parser-compiler/

# Bab 8

## Parser Generator (Bison/Yacc) dan Praktikum Parser

### 8.1 Tujuan Pembelajaran

Setelah mempelajari bab ini, mahasiswa diharapkan mampu:

1. Memahami konsep dan keuntungan menggunakan parser generator
2. Menulis grammar specification menggunakan Bison/Yacc
3. Mengintegrasikan Flex lexer dengan Bison parser
4. Mengimplementasikan semantic actions untuk membangun AST
5. Menangani error dalam parser yang dihasilkan Bison
6. Membangun parser lengkap untuk subset bahasa sederhana

### 8.2 Pendahuluan

Pada bab ini kita menyelesaikan **parser proyek compiler subset C** dengan Bison. Lexer proyek sudah ada di Bab 4 (file `simplec.l`); grammar proyek telah didefinisikan di Bab 1 dan Bab 5 (Bagian 1.9, 5.3). File `simplec.y` yang dibangun di bab ini mengimplementasikan grammar tersebut dan bersama `simplec.l` menjadi tulang punggung codebase proyek.

Setelah mempelajari implementasi parser secara manual (recursive descent) dan memahami konsep LR parsing, kita akan mempelajari cara menggunakan **parser generator** untuk menghasilkan parser secara otomatis dari grammar specification. Pendekatan ini memiliki beberapa keuntungan:

- **Produktivitas**: Menghemat waktu dan mengurangi kesalahan dalam implementasi parser
- **Maintainability**: Grammar dapat diubah dengan mudah tanpa menulis ulang parser secara manual

- **Konsistensi**: Parser yang dihasilkan konsisten dengan grammar specification
- **Error Handling**: Parser generator menyediakan mekanisme error handling yang lebih baik

Menurut sumber dari IT Trip:

> "Bison / YACC: define grammar in a .y file, specify %token s, grammar rules, actions, etc. Generates C parser (or C++ variants). Flex + Bison: use Flex to build the lexer (.l file), Bison for parser, integrate them via tokens."[11]

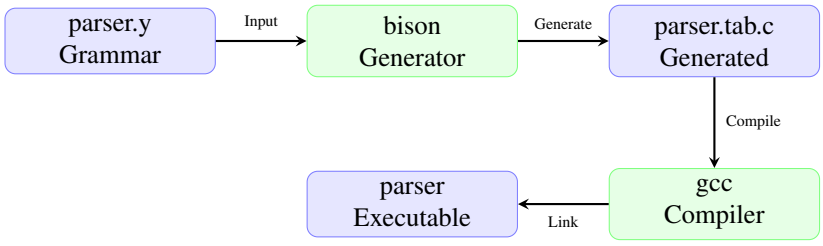
### 8.2.1 Bison vs Yacc

**Yacc** (Yet Another Compiler Compiler) adalah parser generator yang dikembangkan di Bell Labs pada tahun 1970-an. **Bison** adalah implementasi GNU dari Yacc dengan beberapa peningkatan:

- Dukungan untuk C++ output
- Fitur tambahan untuk error recovery
- Dukungan untuk GLR (Generalized LR) parsing
- Dokumentasi yang lebih lengkap
- Open source dan aktif dikembangkan

Dalam buku ini, kita akan menggunakan **Bison** karena lebih modern dan tersedia secara luas. Namun, konsep yang dipelajari juga berlaku untuk Yacc.

Gambar 8.1 menunjukkan alur kerja Bison parser generator.




Gambar 8.1: Workflow Bison parser generator

## 8.3 Struktur File Grammar Bison

File grammar Bison memiliki ekstensi `.y` dan terdiri dari tiga bagian utama yang dipisahkan oleh `%%`:

### 8.3.1 Bagian 1: Definitions (Prologue)

Bagian ini berisi:

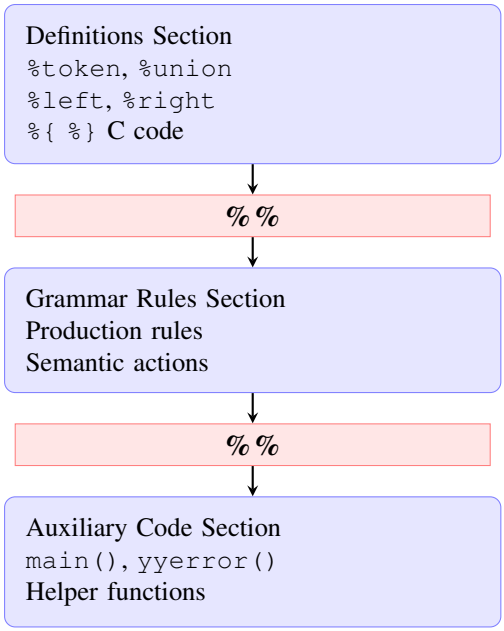
- Deklarasi token (`%token`)
- Deklarasi tipe semantic value (`%union`, `%type`)
- Precedence dan associativity (`%left`, `%right`, `%nonassoc`)
- Kode C yang akan disisipkan (`%{ ... %}`)
- Deklarasi start symbol (`%start`)

### 8.3.2 Bagian 2: Grammar Rules

Bagian ini berisi aturan-aturan grammar dengan semantic actions. Formatnya:

```
nonterminal: production1 { action1 }
           | production2 { action2 }
           | ...
           ;
```

Gambar 8.2 menunjukkan struktur file grammar Bison.



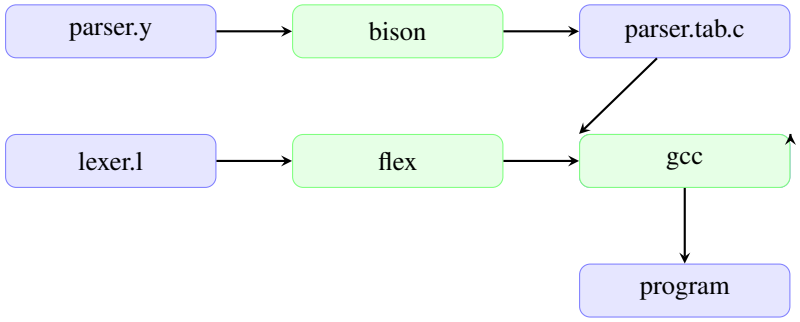

Gambar 8.2: Struktur file grammar Bison

### 8.3.3 Bagian 3: Auxiliary Code

Bagian ini berisi kode C tambahan seperti:

- Fungsi `main()`
- Fungsi `yyerror()` untuk error handling
- Fungsi pendukung lainnya

Gambar 8.3 menunjukkan workflow lengkap dari specification hingga executable.




Gambar 8.3: Workflow lengkap Flex + Bison

## 8.4 Semantic Values dan Semantic Actions

### 8.4.1 Konsep Semantic Values

Setiap token (terminal) dan nonterminal dalam grammar dapat membawa **semantic value**—sebuah nilai data yang diasosiasikan dengan simbol tersebut. Semantic value dapat berupa:

- Integer (untuk literal angka)
- String (untuk identifier)
- Pointer ke AST node
- Tipe data kompleks lainnya

Dalam Bison:

- `$$` merujuk pada semantic value dari **left-hand side (LHS)** (hasil produksi)
- `$1`, `$2`, ..., `$n` merujuk pada semantic value dari simbol ke-1, ke-2, ..., ke-n di **right-hand side (RHS)**

### 8.4.2 Semantic Actions

**Semantic actions** adalah blok kode C yang dieksekusi ketika sebuah produksi di-reduce. Actions dapat digunakan untuk:

- Membangun AST
- Mengevaluasi ekspresi
- Memvalidasi semantik
- Menghasilkan output

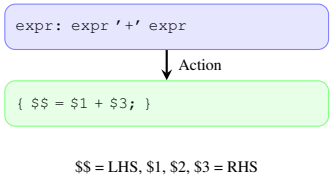
Contoh sederhana:

Listing 8.1: Contoh semantic action sederhana

```
expr: expr '+' expr { $$ = $1 + $3; }
    | expr '-' expr { $$ = $1 - $3; }
    | NUMBER        { $$ = $1; }
    ;
```

Pada contoh di atas, ketika parser menemukan `expr '+' expr`, action `$$ = $1 + $3` akan dieksekusi untuk menghitung jumlah dari kedua ekspresi.

Gambar 8.4 menunjukkan bagaimana semantic actions bekerja dalam Bison.




Gambar 8.4: Semantic actions dalam Bison

## 8.5 Typing Semantic Values dengan %union

Untuk mendukung lebih dari satu tipe semantic value, kita menggunakan `%union`:

Listing 8.2: Deklarasi %union untuk berbagai tipe

```
%union {
    int intval;           // Untuk integer literals
    double dval;          // Untuk floating point
    char *strval;         // Untuk string/identifier
    ASTNode *astnode;     // Untuk AST nodes
}
```

Kemudian kita mendeklarasikan token dan nonterminal dengan tipe tertentu:

```
%token <intval> NUMBER
%token <strval> IDENTIFIER
%type <astnode> expr statement
```

## 8.6 Contoh Lengkap: Calculator dengan Bison

Mari kita buat contoh lengkap parser untuk calculator sederhana menggunakan Flex dan Bison.

### 8.6.1 File Lexer (calc.l)

Listing 8.3: File lexer untuk calculator

```
%{
#include "calc.tab.h"  // Header yang dihasilkan Bison
#include <stdlib.h>
%}

%%
[0-9]+          { yylval.intval = atoi(yytext); return NUMBER; }
[0-9]+\.[0-9]+  { yylval.dval = atof(yytext); return FLOAT; }
[ \t\n]         { /* skip whitespace */ }
"+"             { return PLUS; }
"-"             { return MINUS; }
"*"             { return MULTIPLY; }
"/"             { return DIVIDE; }
"("             { return LPAREN; }
")"             { return RPAREN; }
.               { return yytext[0]; }
%%

int yywrap() { return 1; }
```

### 8.6.2 File Parser (calc.y)

Listing 8.4: File parser untuk calculator

```
%{
#include <stdio.h>
#include <stdlib.h>

extern int yylex();
extern int yyparse();
void yyerror(const char *s);
%}

%union {
    int intval;
    double dval;
}

%token <intval> NUMBER
%token <dval> FLOAT
%token PLUS MINUS MULTIPLY DIVIDE LPAREN RPAREN

%left PLUS MINUS
```

```
%left MULTIPLY DIVIDE

%type <dval> expr

%%
input:
    /* empty */
  | input expr '\n' { printf("Result = %g\n", $2); }
  ;

expr:
    NUMBER          { $$ = (double)$1; }
  | FLOAT           { $$ = $1; }
  | expr '+' expr { $$ = $1 + $3; }
  | expr '-' expr { $$ = $1 - $3; }
  | expr '*' expr { $$ = $1 * $3; }
  | expr '/' expr {
        if ($3 == 0.0) {
          yyerror("Division by zero");
          $$ = 0.0;
        } else {
          $$ = $1 / $3;
        }
      }
  | '(' expr ')'  { $$ = $2; }
  ;

%%

void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}

int main() {
    printf("Calculator - Enter expressions (Ctrl+D to exit)\n");
    yyparse();
    return 0;
}
```

### 8.6.3 Kompilasi dan Eksekusi

Untuk mengompilasi program:

```
flex calc.l          # Generate lexer
bison -d calc.y      # Generate parser (d flag untuk header)
gcc lex.yy.c calc.tab.c -o calc -lfl
./calc
```

## 8.7 Integrasi Flex dan Bison

### 8.7.1 Interface antara Lexer dan Parser

Integrasi antara Flex lexer dan Bison parser dilakukan melalui:

1. **Token Definitions**: Bison menghasilkan file header (misalnya `calc.tab.h`) yang berisi definisi token. Lexer meng-include header ini.
2. **Token Values**: Lexer mengisi `yylval` dengan semantic value sebelum mengembalikan token.
3. **Function Calls**: Parser memanggil `yylex()` untuk mendapatkan token berikutnya.

### 8.7.2 Contoh Integrasi

Listing 8.5: Contoh integrasi Flex dan Bison

```
// Dalam calc.l
%{
#include "calc.tab.h"  // Include header dari Bison
%}

// Dalam calc.y
%{
extern int yylex();    // Deklarasi fungsi lexer
%}
```

## 8.8 Semantic Actions untuk Membangun AST

Salah satu penggunaan utama semantic actions adalah untuk membangun **Abstract Syntax Tree (AST)**. Mari kita lihat contoh yang lebih kompleks.

### 8.8.1 Struktur AST Node

Pertama, kita definisikan struktur untuk AST node:

Listing 8.6: Definisi AST node

```
typedef enum {
    NODE_NUMBER,
    NODE_ADD,
    NODE_SUB,
    NODE_MUL,
    NODE_DIV
} NodeType;

typedef struct ASTNode {
    NodeType type;
```

```
    double value;  // Untuk NODE_NUMBER
    struct ASTNode *left;   // Untuk operasi binary
    struct ASTNode *right;
} ASTNode;

ASTNode* create_number_node(double value) {
    ASTNode *node = malloc(sizeof(ASTNode));
    node->type = NODE_NUMBER;
    node->value = value;
    node->left = node->right = NULL;
    return node;
}

ASTNode* create_binary_node(NodeType type, ASTNode *left, ASTNode *right)
↪ {
    ASTNode *node = malloc(sizeof(ASTNode));
    node->type = type;
    node->left = left;
    node->right = right;
    return node;
}
```

### 8.8.2 Grammar dengan AST Building

Listing 8.7: Grammar dengan semantic actions untuk AST

```
%union {
    double dval;
    ASTNode *astnode;
}

%token <dval> NUMBER
%type <astnode> expr

%%
expr:
    NUMBER { $$ = create_number_node($1); }
  | expr '+' expr { $$ = create_binary_node(NODE_ADD, $1, $3); }
  | expr '-' expr { $$ = create_binary_node(NODE_SUB, $1, $3); }
  | expr '*' expr { $$ = create_binary_node(NODE_MUL, $1, $3); }
  | expr '/' expr { $$ = create_binary_node(NODE_DIV, $1, $3); }
  | '(' expr ')' { $$ = $2; }
  ;
```

## 8.9 Precedence dan Associativity

Untuk menangani precedence dan associativity tanpa membuat grammar yang ambigu, Bison menyediakan deklarasi khusus:

### 8.9.1 Deklarasi Precedence

```
%left '+' '-'      // Left-associative, precedence rendah
%left '*' '/'      // Left-associative, precedence tinggi
%right '^'         // Right-associative (exponentiation)
%nonassoc '<' '>'  // Non-associative (comparison)
```

Urutan deklarasi menentukan precedence: yang dideklarasikan terakhir memiliki precedence tertinggi.

### 8.9.2 Contoh Penggunaan

Listing 8.8: Grammar dengan precedence

```
%left PLUS MINUS
%left MULTIPLY DIVIDE
%right POWER

%%
expr:
    NUMBER
  | expr PLUS expr    // Precedence rendah
  | expr MINUS expr
  | expr MULTIPLY expr // Precedence tinggi
  | expr DIVIDE expr
  | expr POWER expr   // Precedence tertinggi, right-associative
  ;
```

Dengan deklarasi ini, ekspresi `2 + 3 * 4` akan di-parse sebagai `2 + (3 * 4)`, dan ekspresi `2^3^4` akan di-parse sebagai `2^(3^4)`.

## 8.10 Error Handling dalam Bison

### 8.10.1 Error Token

Bison menyediakan token khusus `error` untuk error recovery. Ketika parser menemukan error, ia dapat mencoba recovery dengan menggunakan produksi yang mengandung `error`.

### 8.10.2 Contoh Error Recovery

Listing 8.9: Error recovery dalam Bison

```
%%
input:
    /* empty */
  | input line
  ;

line:
```

```
    expr '\n' { printf("Result = %g\n", $1); }
  | error '\n' {
        yyerrok;  // Reset error state
        yyclearin; // Clear lookahead token
        printf("Syntax error, please try again.\n");
      }
  ;
```

### 8.10.3 Fungsi yyerror

Fungsi `yyerror()` dipanggil ketika terjadi syntax error. Kita dapat mengimplementasikannya untuk memberikan pesan error yang informatif:

Listing 8.10: Implementasi yyerror yang lebih informatif

```
void yyerror(const char *s) {
    extern int yylineno;
    fprintf(stderr, "Error at line %d: %s\n", yylineno, s);
}
```

### 8.10.4 Location Tracking

Untuk memberikan informasi lokasi yang lebih akurat, kita dapat menggunakan location tracking:

Listing 8.11: Location tracking dalam Bison

```
%locations

%%
expr: expr '+' expr {
    @$.first_line = @1.first_line;
    @$.first_column = @1.first_column;
    @$.last_line = @3.last_line;
    @$.last_column = @3.last_column;
    if (/* some error condition */) {
        yyerror("Error in expression");
    }
    $$ = $1 + $3;
}
```

## 8.11 Mid-Rule Actions

Bison mendukung **mid-rule actions**—actions yang muncul di tengah-tengah produksi, sebelum seluruh RHS di-match. Ini berguna untuk:

- Validasi intermediate state

- Inisialisasi variabel
- Side effects tertentu

Listing 8.12: Contoh mid-rule action

```
stmt:
    IDENTIFIER { printf("Variable: %s\n", $1); }
    '='
    { /* mid-rule action */
      check_variable($1);
    }
    expr { assign($1, $4); }
  ;
```

Catatan penting: Hanya action terakhir yang menentukan `$$` untuk LHS.

## 8.12 Contoh Praktis: Parser Proyek Subset C

Contoh berikut adalah **parser proyek compiler subset C**: file `simplec.l` (lexer) dan `simplec.y` (parser) memenuhi spesifikasi token (Bab 1, Bagian 1.9) dan grammar proyek (Bab 5, Bagian 5.3). Subset yang didukung:

- Deklarasi variabel (int, float)
- Assignment statements
- Ekspresi aritmatika
- Print statements

### 8.12.1 File Lexer (simplec.l)

Listing 8.13: Lexer untuk subset C

```
%{
#include "simplec.tab.h"
#include <string.h>
%}

%%
"int"       { return INT; }
"float"     { return FLOAT; }
"print"     { return PRINT; }
[a-zA-Z_][a-zA-Z0-9_]* {
    yylval.strval = strdup(yytext);
    return IDENTIFIER;
}
[0-9]+      { yylval.intval = atoi(yytext); return NUMBER; }
[0-9]+\.[0-9]+ { yylval.dval = atof(yytext); return FLOAT_LITERAL; }
```

```
[ \t\n]      { /* skip */ }
"="          { return ASSIGN; }
";"          { return SEMICOLON; }
"+"          { return PLUS; }
"-"          { return MINUS; }
"*"          { return MULTIPLY; }
"/"          { return DIVIDE; }
"("          { return LPAREN; }
")"          { return RPAREN; }
.            { return yytext[0]; }
%%

int yywrap() { return 1; }
```

### 8.12.2 File Parser (simplec.y)

Listing 8.14: Parser untuk subset C

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

extern int yylex();
void yyerror(const char *s);

// Symbol table sederhana
typedef struct {
    char *name;
    int type;  // 0 = int, 1 = float
    union {
        int intval;
        double dval;
    } value;
} Symbol;

Symbol symbols[100];
int sym_count = 0;
%}

%union {
    int intval;
    double dval;
    char *strval;
}

%token <intval> NUMBER
%token <dval> FLOAT_LITERAL
%token <strval> IDENTIFIER
%token INT FLOAT PRINT ASSIGN SEMICOLON
%token PLUS MINUS MULTIPLY DIVIDE LPAREN RPAREN

%left PLUS MINUS
```

```
%left MULTIPLY DIVIDE

%type <dval> expr

%%
program:
    /* empty */
  | program statement
  ;

statement:
    declaration SEMICOLON
  | assignment SEMICOLON
  | print_stmt SEMICOLON
  ;

declaration:
    INT IDENTIFIER {
        // Add to symbol table
        symbols[sym_count].name = strdup($2);
        symbols[sym_count].type = 0;
        sym_count++;
        printf("Declared int variable: %s\n", $2);
    }
  | FLOAT IDENTIFIER {
        symbols[sym_count].name = strdup($2);
        symbols[sym_count].type = 1;
        sym_count++;
        printf("Declared float variable: %s\n", $2);
    }
  ;

assignment:
    IDENTIFIER ASSIGN expr {
        // Find variable in symbol table and assign
        printf("Assigning %g to %s\n", $3, $1);
    }
  ;

print_stmt:
    PRINT expr {
        printf("Print: %g\n", $2);
    }
  ;

expr:
    NUMBER { $$ = (double)$1; }
  | FLOAT_LITERAL { $$ = $1; }
  | IDENTIFIER {
        // Lookup in symbol table
        $$ = 0.0; // Simplified
    }
  | expr PLUS expr { $$ = $1 + $3; }
  | expr MINUS expr { $$ = $1 - $3; }
```

```
90    | expr MULTIPLY expr { $$ = $1 * $3; }
91    | expr DIVIDE expr {
92          if ($3 == 0.0) {
93              yyerror("Division by zero");
94              $$ = 0.0;
95          } else {
96              $$ = $1 / $3;
97          }
98      }
99    | LPAREN expr RPAREN { $$ = $2; }
100   ;
101
102 %%
103
104 void yyerror(const char *s) {
105     fprintf(stderr, "Syntax error: %s\n", s);
106 }
107
108 int main() {
109     printf("Simple C Parser - Enter statements (Ctrl+D to exit)\n");
110     yyparse();
111     return 0;
112 }
```

## 8.13 Best Practices dan Tips

### 8.13.1 Organisasi File

- Pisahkan definisi AST node ke file header terpisah
- Gunakan Makefile untuk mengotomatisasi kompilasi
- Dokumentasikan grammar dengan baik

### 8.13.2 Debugging

- Gunakan flag `-v` pada Bison untuk menghasilkan file `.output` yang berisi informasi tentang parsing table
- Gunakan `YYDEBUG` untuk debugging parser
- Tambahkan print statements dalam semantic actions untuk tracing

### 8.13.3 Performance

- Hindari semantic actions yang terlalu kompleks
- Gunakan `%destructor` untuk membersihkan memory jika menggunakan pointer

- Pertimbangkan menggunakan GLR parsing untuk grammar yang ambigu

## 8.14 Kesimpulan

Dalam bab ini, kita telah mempelajari:

1. Konsep parser generator dan keuntungannya
2. Struktur file grammar Bison dengan tiga bagian utama
3. Semantic values dan semantic actions untuk membangun AST
4. Integrasi Flex lexer dengan Bison parser
5. Precedence dan associativity dalam grammar
6. Error handling dan recovery dalam Bison
7. Contoh praktis parser untuk calculator dan subset C

Parser generator seperti Bison sangat powerful untuk membangun parser yang robust dan maintainable. Dengan memahami konsep-konsep ini, kita dapat membangun parser untuk bahasa yang lebih kompleks.

## 8.15 Referensi dan Bahan Bacaan Lanjutan

Untuk memperdalam pemahaman tentang parser generator, mahasiswa disarankan membaca:

- **flex & bison**: Levine, J. R. (2009). *flex & bison: Text Processing Tools* [10] - Buku lengkap tentang Flex dan Bison
- **Dragon Book**: Aho, Lam, Sethi, & Ullman (2006). *Compilers: Principles, Techniques, and Tools* [1] - Bab 4: Syntax-Directed Translation
- **GNU Bison Manual**: `https://www.gnu.org/software/bison/manual/` - Dokumentasi resmi Bison dengan contoh lengkap
- **IT Trip Tutorial**: Tutorial tentang C Parser dengan Flex dan Bison [11]
- **Engineering a Compiler**: Cooper & Torczon (2011) [5] - Bab tentang parser generators

# Bab 9

## Abstract Syntax Tree (AST) dan Struktur Data

### 9.1 Tujuan Pembelajaran

Setelah mempelajari bab ini, mahasiswa diharapkan mampu:

1. Memahami konsep Abstract Syntax Tree (AST) dan perbedaannya dengan parse tree
2. Merancang struktur data AST yang sesuai untuk berbagai konstruk bahasa
3. Mengimplementasikan AST node classes/structs dalam C/C++
4. Menerapkan berbagai metode tree traversal (pre-order, post-order, in-order)
5. Memodifikasi parser untuk membangun AST selama proses parsing
6. Membuat AST visualizer untuk debugging dan pembelajaran

### 9.2 Pendahuluan

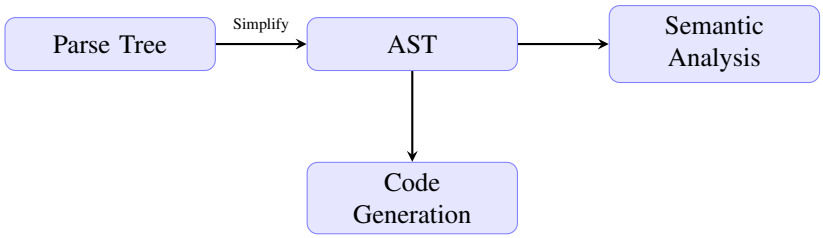
Dalam proyek compiler subset C, parser (Bab 8, `simplec.y`) membangun AST sebagai representasi internal program. Definisinya untuk subset C disajikan di Bagian 9.3; contoh berikut mengacu ke node types tersebut.

Setelah parser berhasil memverifikasi bahwa token-token membentuk struktur yang valid menurut grammar, parser perlu membangun representasi internal program. Representasi ini dapat berupa parse tree (concrete syntax tree) atau Abstract Syntax Tree (AST).

Menurut sumber terbuka:

> "Abstract Syntax Trees represent the nested structure of language constructs (expressions, statements, declarations). AST representation: whether nodes represent every symbol from the grammar or a simplified form; whether annotations (types, scopes) are added later."[2]

Gambar 9.1 menunjukkan posisi AST dalam pipeline kompilator.




Gambar 9.1: Posisi AST dalam pipeline kompilator

### 9.2.1 Perbedaan Parse Tree dan AST

Perbedaan utama antara parse tree dan AST:

- **Parse Tree (Concrete Syntax Tree)**:
  - Mencakup semua detail sintaksis, termasuk semua non-terminal dari grammar
  - Setiap node sesuai dengan aturan produksi grammar
  - Lebih verbose dan mencakup informasi yang tidak diperlukan untuk fase selanjutnya
  - Contoh: Node untuk operator precedence yang sudah jelas dari struktur
- **Abstract Syntax Tree (AST)**:
  - Menghilangkan detail sintaksis yang tidak relevan
  - Fokus pada struktur semantik program
  - Lebih kompak dan efisien untuk analisis semantik dan code generation
  - Hanya menyertakan informasi yang diperlukan untuk fase-fase selanjutnya

### 9.2.2 Contoh Perbandingan

Untuk ekspresi `3 + 4 * 5`, parse tree-nya akan mencakup semua non-terminal:

Sedangkan AST-nya lebih sederhana dan langsung mencerminkan semantik:

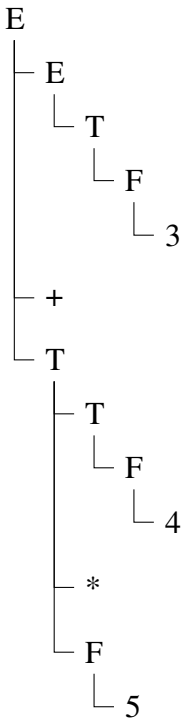
AST menghilangkan node-node intermediate seperti $E$, $T$, $F$ yang tidak diperlukan untuk pemahaman semantik.

Gambar 9.4 menunjukkan berbagai jenis node dalam AST.

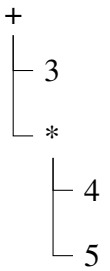
Precedence operator sudah tercermin dalam struktur tree.

## 9.3 AST Proyek Subset C

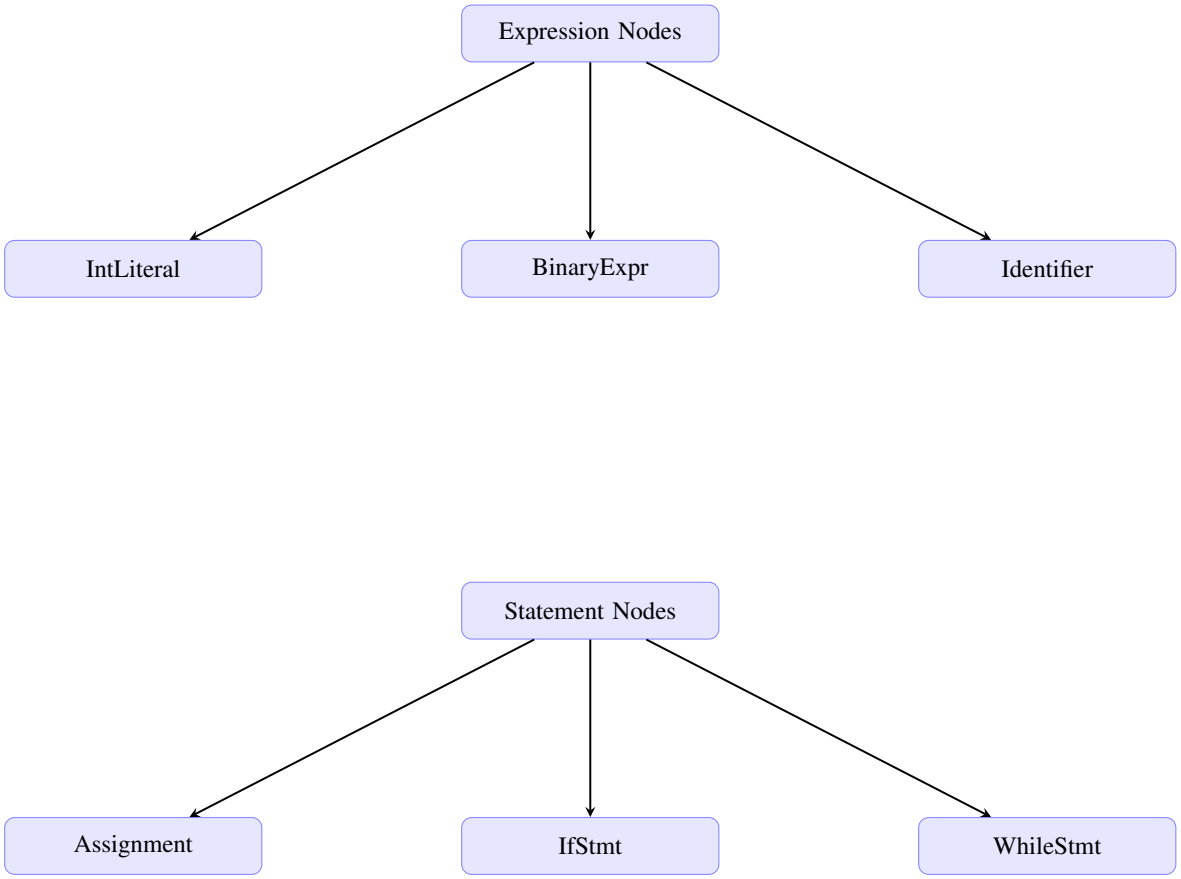
Untuk proyek compiler subset C (spesifikasi Bab 1, grammar Bab 5), AST didefinisikan sebagai berikut. Parser proyek di Bab 8 (`simplec.y`) membangun AST ini melalui semantic actions; definisi node mengacu ke grammar proyek (Bagian 5.3).




Gambar 9.2: Parse tree untuk ekspresi `3 + 4 * 5`




Gambar 9.3: AST untuk ekspresi `3 + 4 * 5`




Gambar 9.4: Klasifikasi jenis node dalam Abstract Syntax Tree (AST)

### 9.3.1 Node Types AST Proyek

- **Program**: root; berisi daftar statement (sequence).
- **Declaration**: `int` identifier atau `float` identifier; menyimpan nama dan tipe.
- **Assignment**: identifier = expr; menyimpan nama variabel dan pointer ke node ekspresi.
- **PrintStmt**: `print ( arg )` ; arg berupa string-literal atau expr.
- **Expr**: ekspresi; subtipe: BinOp (op, left, right), Number (literal), FloatLiteral, Identifier (nama), StringLiteral (nilai).
- **BinOp**: operator biner (+, −, ⋆, /); anak kiri dan kanan bertipe expr.

Struktur data dapat diimplementasikan dalam `ast.h` / `ast.c` di folder proyek (`proyek-compiler-subset-c/`). Traversal (pre-order, post-order) untuk semantic analysis dan code generation mengacu ke node-node ini. Bab 10 (symbol table) dan Bab 11 (type checking) memakai AST ini sebagai input; Bab 12 (IR) menghasilkan three-address code atau quadruples dari AST proyek.

## 9.4 Struktur Data AST

AST terdiri dari node-node yang merepresentasikan berbagai konstruk bahasa. Setiap node memiliki:

- **Node Type**: Jenis node (expression, statement, declaration, dll.)
- **Children**: Node-node anak (untuk operasi biner, blok statement, dll.)
- **Attributes**: Informasi tambahan (nilai literal, nama identifier, operator, dll.)
- **Location**: Informasi posisi dalam source code (untuk error reporting)

### 9.4.1 Node Types dalam AST

Node types yang umum digunakan dalam AST:

#### Expression Nodes

- **Literal Nodes**: Integer, float, string, boolean, null
- **Identifier Nodes**: Nama variabel, fungsi, tipe
- **Binary Expression Nodes**: Operasi biner (+, -, *, /, ==, !=, <, >, dll.)

- **Unary Expression Nodes**: Operasi unary (negation, logical NOT, address-of, dereference)
- **Function Call Nodes**: Pemanggilan fungsi dengan daftar argumen
- **Array Access Nodes**: Akses elemen array dengan index
- **Member Access Nodes**: Akses member struct/class (dot notation)

**Statement Nodes**

- **Expression Statement**: Statement yang merupakan ekspresi (dengan semicolon)
- **Variable Declaration**: Deklarasi variabel dengan tipe dan optional initializer
- **Assignment Statement**: Assignment ke variabel atau l-value lainnya
- **If Statement**: Conditional statement dengan condition, then-branch, optional else-branch
- **While Statement**: Loop dengan condition dan body
- **For Statement**: Loop dengan init, condition, increment, dan body
- **Return Statement**: Return dengan optional expression
- **Block Statement**: Blok statement yang berisi daftar statement
- **Break/Continue Statement**: Control flow untuk loop

**Declaration Nodes**

- **Function Declaration**: Deklarasi fungsi dengan parameter, return type, dan body
- **Type Declaration**: Deklarasi tipe baru (struct, enum, typedef)
- **Variable Declaration**: Deklarasi variabel global atau local

**Program Node**

- **Program/Module Node**: Root node yang berisi semua deklarasi dan statement dalam program

## 9.5 Implementasi AST dalam C++

Dalam C++, AST biasanya diimplementasikan menggunakan inheritance dan virtual functions. Berikut adalah contoh implementasi dasar:

### 9.5.1 Base Node Class

Listing 9.1: Base AST Node Class

```cpp
// ASTNode.hpp
#ifndef ASTNODE_HPP
#define ASTNODE_HPP

#include <string>
#include <memory>
#include <vector>

// Forward declaration
class ASTVisitor;

// Base class untuk semua AST nodes
class ASTNode {
public:
    virtual ~ASTNode() = default;

    // Visitor pattern untuk traversal
    virtual void accept(ASTVisitor& visitor) = 0;

    // Location information untuk error reporting
    struct Location {
        int line;
        int column;
        std::string filename;
    };

    Location location;
};

#endif
```

### 9.5.2 Expression Nodes

Listing 9.2: Expression Node Classes

```cpp
// Expression.hpp
#ifndef EXPRESSION_HPP
#define EXPRESSION_HPP

#include "ASTNode.hpp"
#include <string>
#include <memory>

// Base class untuk semua expressions
class Expr : public ASTNode {
public:
    // Type information (akan diisi oleh semantic analyzer)
    std::string type;
};

```

```
16 // Integer literal node
17 class IntLiteral : public Expr {
18 public:
19     int value;
20
21     IntLiteral(int val) : value(val) {}
22     void accept(ASTVisitor& visitor) override;
23 };
24
25 // Identifier node (variable name, function name, etc.)
26 class Identifier : public Expr {
27 public:
28     std::string name;
29
30     Identifier(const std::string& n) : name(n) {}
31     void accept(ASTVisitor& visitor) override;
32 };
33
34 // Binary expression node (+, -, *, /, ==, etc.)
35 class BinaryExpr : public Expr {
36 public:
37     std::string op;  // Operator: "+", "-", "*", "/", "==", etc.
38     std::unique_ptr<Expr> left;
39     std::unique_ptr<Expr> right;
40
41     BinaryExpr(const std::string& op,
42                std::unique_ptr<Expr> l,
43                std::unique_ptr<Expr> r)
44         : op(op), left(std::move(l)), right(std::move(r)) {}
45
46     void accept(ASTVisitor& visitor) override;
47 };
48
49 // Unary expression node (-, !, etc.)
50 class UnaryExpr : public Expr {
51 public:
52     std::string op;
53     std::unique_ptr<Expr> operand;
54
55     UnaryExpr(const std::string& op, std::unique_ptr<Expr> opnd)
56         : op(op), operand(std::move(opnd)) {}
57
58     void accept(ASTVisitor& visitor) override;
59 };
60
61 // Function call node
62 class FunctionCall : public Expr {
63 public:
64     std::unique_ptr<Identifier> functionName;
65     std::vector<std::unique_ptr<Expr>> arguments;
66
67     FunctionCall(std::unique_ptr<Identifier> name)
68         : functionName(std::move(name)) {}
69
```

```
    void accept(ASTVisitor& visitor) override;
};

#endif
```

### 9.5.3 Statement Nodes

Listing 9.3: Statement Node Classes

```
// Statement.hpp
#ifndef STATEMENT_HPP
#define STATEMENT_HPP

#include "ASTNode.hpp"
#include "Expression.hpp"
#include <memory>
#include <vector>

// Base class untuk semua statements
class Stmt : public ASTNode {
};

// Expression statement (expression followed by semicolon)
class ExprStmt : public Stmt {
public:
    std::unique_ptr<Expr> expression;

    ExprStmt(std::unique_ptr<Expr> expr)
        : expression(std::move(expr)) {}

    void accept(ASTVisitor& visitor) override;
};

// Variable declaration statement
class VarDecl : public Stmt {
public:
    std::string typeName;
    std::string varName;
    std::unique_ptr<Expr> initializer;  // Optional

    VarDecl(const std::string& type, const std::string& name)
        : typeName(type), varName(name), initializer(nullptr) {}

    VarDecl(const std::string& type, const std::string& name,
            std::unique_ptr<Expr> init)
        : typeName(type), varName(name), initializer(std::move(init)) {}

    void accept(ASTVisitor& visitor) override;
};

// Assignment statement
class AssignStmt : public Stmt {
public:
```

```
45      std::unique_ptr<Identifier> left;
46      std::unique_ptr<Expr> right;
47
48      AssignStmt(std::unique_ptr<Identifier> l, std::unique_ptr<Expr> r)
49          : left(std::move(l)), right(std::move(r)) {}
50
51      void accept(ASTVisitor& visitor) override;
52  };
53
54  // If statement
55  class IfStmt : public Stmt {
56  public:
57      std::unique_ptr<Expr> condition;
58      std::unique_ptr<Stmt> thenBranch;
59      std::unique_ptr<Stmt> elseBranch;  // Optional
60
61      IfStmt(std::unique_ptr<Expr> cond, std::unique_ptr<Stmt> thenStmt)
62          : condition(std::move(cond)),
63            thenBranch(std::move(thenStmt)),
64            elseBranch(nullptr) {}
65
66      IfStmt(std::unique_ptr<Expr> cond,
67             std::unique_ptr<Stmt> thenStmt,
68             std::unique_ptr<Stmt> elseStmt)
69          : condition(std::move(cond)),
70            thenBranch(std::move(thenStmt)),
71            elseBranch(std::move(elseStmt)) {}
72
73      void accept(ASTVisitor& visitor) override;
74  };
75
76  // While statement
77  class WhileStmt : public Stmt {
78  public:
79      std::unique_ptr<Expr> condition;
80      std::unique_ptr<Stmt> body;
81
82      WhileStmt(std::unique_ptr<Expr> cond, std::unique_ptr<Stmt> b)
83          : condition(std::move(cond)), body(std::move(b)) {}
84
85      void accept(ASTVisitor& visitor) override;
86  };
87
88  // Block statement (compound statement)
89  class BlockStmt : public Stmt {
90  public:
91      std::vector<std::unique_ptr<Stmt>> statements;
92
93      void addStatement(std::unique_ptr<Stmt> stmt) {
94          statements.push_back(std::move(stmt));
95      }
96
97      void accept(ASTVisitor& visitor) override;
98  };
```

```
99
100 // Return statement
101 class ReturnStmt : public Stmt {
102 public:
103     std::unique_ptr<Expr> expression;  // Optional
104
105     ReturnStmt() : expression(nullptr) {}
106     ReturnStmt(std::unique_ptr<Expr> expr) : expression(std::move(expr))
    ↪ {}
107
108     void accept(ASTVisitor& visitor) override;
109 };
110
111 #endif
```

### 9.5.4 Program Node

Listing 9.4: Program Node Class

```
1  // Program.hpp
2  #ifndef PROGRAM_HPP
3  #define PROGRAM_HPP
4
5  #include "ASTNode.hpp"
6  #include "Statement.hpp"
7  #include <vector>
8  #include <memory>
9
10 // Root node untuk seluruh program
11 class Program : public ASTNode {
12 public:
13     std::vector<std::unique_ptr<Stmt>> statements;
14
15     void addStatement(std::unique_ptr<Stmt> stmt) {
16         statements.push_back(std::move(stmt));
17     }
18
19     void accept(ASTVisitor& visitor) override;
20 };
21
22 #endif
```

## 9.6 Visitor Pattern untuk Tree Traversal

Visitor pattern memisahkan algoritma traversal dari struktur node. Ini memungkinkan kita menambahkan operasi baru tanpa memodifikasi kelas node.

### 9.6.1 Visitor Interface

Listing 9.5: AST Visitor Interface

```
// ASTVisitor.hpp
#ifndef ASTVISITOR_HPP
#define ASTVISITOR_HPP

// Forward declarations
class IntLiteral;
class Identifier;
class BinaryExpr;
class UnaryExpr;
class FunctionCall;
class ExprStmt;
class VarDecl;
class AssignStmt;
class IfStmt;
class WhileStmt;
class BlockStmt;
class ReturnStmt;
class Program;

class ASTVisitor {
public:
    virtual ~ASTVisitor() = default;

    // Expression visitors
    virtual void visit(IntLiteral& node) = 0;
    virtual void visit(Identifier& node) = 0;
    virtual void visit(BinaryExpr& node) = 0;
    virtual void visit(UnaryExpr& node) = 0;
    virtual void visit(FunctionCall& node) = 0;

    // Statement visitors
    virtual void visit(ExprStmt& node) = 0;
    virtual void visit(VarDecl& node) = 0;
    virtual void visit(AssignStmt& node) = 0;
    virtual void visit(IfStmt& node) = 0;
    virtual void visit(WhileStmt& node) = 0;
    virtual void visit(BlockStmt& node) = 0;
    virtual void visit(ReturnStmt& node) = 0;

    // Program visitor
    virtual void visit(Program& node) = 0;
};

#endif
```

### 9.6.2 Implementasi Accept Methods

Setiap node class mengimplementasikan method `accept` yang memanggil method `visit` yang sesuai pada visitor:

Listing 9.6: Accept Methods Implementation

```cpp
// Expression.cpp
#include "Expression.hpp"
#include "ASTVisitor.hpp"

void IntLiteral::accept(ASTVisitor& visitor) {
    visitor.visit(*this);
}

void Identifier::accept(ASTVisitor& visitor) {
    visitor.visit(*this);
}

void BinaryExpr::accept(ASTVisitor& visitor) {
    visitor.visit(*this);
    if (left) left->accept(visitor);
    if (right) right->accept(visitor);
}

void UnaryExpr::accept(ASTVisitor& visitor) {
    visitor.visit(*this);
    if (operand) operand->accept(visitor);
}

void FunctionCall::accept(ASTVisitor& visitor) {
    visitor.visit(*this);
    if (functionName) functionName->accept(visitor);
    for (auto& arg : arguments) {
        if (arg) arg->accept(visitor);
    }
}
```

## 9.7 Tree Traversal Methods

Ada tiga metode utama untuk traversing tree: pre-order, post-order, dan in-order.

### 9.7.1 Pre-Order Traversal

Pre-order traversal mengunjungi node sebelum children-nya. Urutan: Root → Left → Right.

Contoh untuk ekspresi `3 + 4 * 5`:

```
Visit: +
Visit: 3
Visit: *
Visit: 4
Visit: 5
```

Pre-order berguna untuk:

- Menyalin tree
- Prefix notation (Polish notation)

- Print struktur tree

### 9.7.2 Post-Order Traversal

Post-order traversal mengunjungi node setelah children-nya. Urutan: Left → Right → Root.

Contoh untuk ekspresi `3 + 4 * 5`:

```
Visit: 3
Visit: 4
Visit: 5
Visit: *
Visit: +
```

Post-order berguna untuk:

- Menghapus tree (deallocate memory)
- Postfix notation (Reverse Polish notation)
- Evaluasi ekspresi
- Code generation (stack-based)

### 9.7.3 In-Order Traversal

In-order traversal mengunjungi left child, kemudian node, kemudian right child. Urutan: Left → Root → Right.

Contoh untuk ekspresi `3 + 4 * 5`:

```
Visit: 3
Visit: +
Visit: 4
Visit: *
Visit: 5
```

In-order berguna untuk:

- Infix notation (seperti yang ditulis dalam source code)
- Binary search tree operations

### 9.7.4 Implementasi Traversal dengan Visitor

Berikut adalah contoh implementasi PrettyPrinter visitor yang menggunakan pre-order traversal:

Listing 9.7: PrettyPrinter Visitor

```
// PrettyPrinter.hpp
#ifndef PRETTYPRINTER_HPP
#define PRETTYPRINTER_HPP

#include "ASTVisitor.hpp"
#include <iostream>
#include <string>

class PrettyPrinter : public ASTVisitor {
private:
    int indentLevel;

    std::string indent() const {
        return std::string(indentLevel * 2, ' ');
    }

public:
    PrettyPrinter() : indentLevel(0) {}

    void visit(IntLiteral& node) override {
        std::cout << indent() << "IntLiteral: " << node.value << std::
↪ endl;
    }

    void visit(Identifier& node) override {
        std::cout << indent() << "Identifier: " << node.name << std::endl
↪ ;
    }

    void visit(BinaryExpr& node) override {
        std::cout << indent() << "BinaryExpr: " << node.op << std::endl;
        indentLevel++;
        if (node.left) node.left->accept(*this);
        if (node.right) node.right->accept(*this);
        indentLevel--;
    }

    void visit(UnaryExpr& node) override {
        std::cout << indent() << "UnaryExpr: " << node.op << std::endl;
        indentLevel++;
        if (node.operand) node.operand->accept(*this);
        indentLevel--;
    }

    void visit(FunctionCall& node) override {
        std::cout << indent() << "FunctionCall" << std::endl;
        indentLevel++;
        if (node.functionName) node.functionName->accept(*this);
        for (auto& arg : node.arguments) {
            if (arg) arg->accept(*this);
        }
        indentLevel--;
    }
```

```
52
53      void visit(ExprStmt& node) override {
54          std::cout << indent() << "ExprStmt" << std::endl;
55          indentLevel++;
56          if (node.expression) node.expression->accept(*this);
57          indentLevel--;
58      }
59
60      void visit(VarDecl& node) override {
61          std::cout << indent() << "VarDecl: " << node.typeName
62                    << " " << node.varName;
63          if (node.initializer) {
64              std::cout << " = ";
65              indentLevel++;
66              node.initializer->accept(*this);
67              indentLevel--;
68          }
69          std::cout << std::endl;
70      }
71
72      void visit(AssignStmt& node) override {
73          std::cout << indent() << "AssignStmt" << std::endl;
74          indentLevel++;
75          if (node.left) node.left->accept(*this);
76          if (node.right) node.right->accept(*this);
77          indentLevel--;
78      }
79
80      void visit(IfStmt& node) override {
81          std::cout << indent() << "IfStmt" << std::endl;
82          indentLevel++;
83          std::cout << indent() << "Condition:" << std::endl;
84          indentLevel++;
85          if (node.condition) node.condition->accept(*this);
86          indentLevel--;
87          std::cout << indent() << "Then:" << std::endl;
88          indentLevel++;
89          if (node.thenBranch) node.thenBranch->accept(*this);
90          indentLevel--;
91          if (node.elseBranch) {
92              std::cout << indent() << "Else:" << std::endl;
93              indentLevel++;
94              node.elseBranch->accept(*this);
95              indentLevel--;
96          }
97          indentLevel--;
98      }
99
100     void visit(WhileStmt& node) override {
101         std::cout << indent() << "WhileStmt" << std::endl;
102         indentLevel++;
103         std::cout << indent() << "Condition:" << std::endl;
104         indentLevel++;
105         if (node.condition) node.condition->accept(*this);
```

```
        indentLevel--;
        std::cout << indent() << "Body:" << std::endl;
        indentLevel++;
        if (node.body) node.body->accept(*this);
        indentLevel--;
        indentLevel--;
    }

    void visit(BlockStmt& node) override {
        std::cout << indent() << "BlockStmt" << std::endl;
        indentLevel++;
        for (auto& stmt : node.statements) {
            if (stmt) stmt->accept(*this);
        }
        indentLevel--;
    }

    void visit(ReturnStmt& node) override {
        std::cout << indent() << "ReturnStmt";
        if (node.expression) {
            std::cout << std::endl;
            indentLevel++;
            node.expression->accept(*this);
            indentLevel--;
        } else {
            std::cout << std::endl;
        }
    }

    void visit(Program& node) override {
        std::cout << "Program" << std::endl;
        indentLevel++;
        for (auto& stmt : node.statements) {
            if (stmt) stmt->accept(*this);
        }
        indentLevel--;
    }
};

#endif
```

## 9.8 Integrasi AST dengan Parser

Parser perlu dimodifikasi untuk membangun AST selama proses parsing, bukan hanya memverifikasi grammar. Berikut adalah contoh integrasi dengan Bison parser generator.

### 9.8.1 Bison Grammar dengan AST Building

Listing 9.8: Bison Grammar dengan AST Building

```
%{
```

```
#include "ASTNode.hpp"
#include "Expression.hpp"
#include "Statement.hpp"
#include "Program.hpp"
%}

%union {
    int intVal;
    char* strVal;
    Expr* expr;
    Stmt* stmt;
    Program* program;
}

%token <intVal> INT_LITERAL
%token <strVal> IDENTIFIER
%token PLUS MINUS MULTIPLY DIVIDE
%token ASSIGN SEMICOLON
%token IF ELSE WHILE RETURN
%token LBRACE RBRACE LPAREN RPAREN

%type <expr> expression
%type <stmt> statement block_statement
%type <program> program

%%

program:
    statement_list {
        $$ = new Program();
        // Add statements to program
    }
    ;

statement_list:
    statement_list statement {
        // Add statement to list
    }
  | /* empty */
    ;

statement:
    expression SEMICOLON {
        $$ = new ExprStmt($1);
    }
  | IDENTIFIER ASSIGN expression SEMICOLON {
        auto id = new Identifier($1);
        $$ = new AssignStmt(std::unique_ptr<Identifier>(id),
                            std::unique_ptr<Expr>($3));
    }
  | block_statement
    ;

block_statement:
```

```
    LBRACE statement_list RBRACE {
        $$ = new BlockStmt();
        // Add statements to block
    }
    ;

expression:
    INT_LITERAL {
        $$ = new IntLiteral($1);
    }
  | IDENTIFIER {
        $$ = new Identifier($1);
    }
  | expression PLUS expression {
        $$ = new BinaryExpr("+",
                            std::unique_ptr<Expr>($1),
                            std::unique_ptr<Expr>($3));
    }
  | expression MINUS expression {
        $$ = new BinaryExpr("-",
                            std::unique_ptr<Expr>($1),
                            std::unique_ptr<Expr>($3));
    }
  | expression MULTIPLY expression {
        $$ = new BinaryExpr("*",
                            std::unique_ptr<Expr>($1),
                            std::unique_ptr<Expr>($3));
    }
  | expression DIVIDE expression {
        $$ = new BinaryExpr("/",
                            std::unique_ptr<Expr>($1),
                            std::unique_ptr<Expr>($3));
    }
  | LPAREN expression RPAREN {
        $$ = $2;
    }
  ;

%%
```

## 9.9 AST Visualizer

AST visualizer berguna untuk debugging dan memahami struktur program. Berikut adalah contoh sederhana menggunakan text-based visualization:

Listing 9.9: AST Text Visualizer

```
// ASTVisualizer.hpp
#ifndef ASTVISIZER_HPP
#define ASTVISIZER_HPP

#include "ASTVisitor.hpp"
#include <iostream>
```

```
#include <string>
#include <vector>

class ASTVisualizer : public ASTVisitor {
private:
    std::vector<bool> isLastChild;

    void printPrefix() {
        for (size_t i = 0; i < isLastChild.size() - 1; i++) {
            if (isLastChild[i]) {
                std::cout << "    ";
            } else {
                std::cout << "|   ";
            }
        }
        if (!isLastChild.empty()) {
            std::cout << (isLastChild.back() ? "+-- " : "+-- ");
        }
    }

public:
    void visit(BinaryExpr& node) override {
        printPrefix();
        std::cout << "BinaryExpr: " << node.op << std::endl;

        isLastChild.push_back(false);
        if (node.left) node.left->accept(*this);
        isLastChild.back() = true;
        if (node.right) node.right->accept(*this);
        isLastChild.pop_back();
    }

    void visit(IntLiteral& node) override {
        printPrefix();
        std::cout << "IntLiteral: " << node.value << std::endl;
    }

    void visit(Identifier& node) override {
        printPrefix();
        std::cout << "Identifier: " << node.name << std::endl;
    }

    // Implement other visit methods...
};

#endif
```

## 9.10 Best Practices

Berikut adalah beberapa best practices dalam implementasi AST:

1. **Memory Management**: Gunakan smart pointers (`std::unique_ptr` atau

`std::shared_ptr`) untuk menghindari memory leaks

2. **Immutable Nodes**: AST nodes biasanya immutable setelah dibuat. Transformasi menghasilkan tree baru

3. **Location Tracking**: Simpan informasi lokasi (line, column) di setiap node untuk error reporting yang lebih baik

4. **Separation of Concerns**: AST nodes hanya menyimpan struktur, bukan algoritma. Gunakan visitor pattern untuk operasi

5. **Type Safety**: Gunakan type system yang kuat (inheritance, enums) untuk mencegah kesalahan

6. **Modularity**: Pisahkan definisi node types ke file-file terpisah untuk kemudahan maintenance

## 9.11 Kesimpulan

Dalam bab ini, kita telah mempelajari:

1. Abstract Syntax Tree (AST) adalah representasi abstrak dari struktur program yang menghilangkan detail sintaksis yang tidak relevan

2. AST terdiri dari berbagai node types: expression nodes, statement nodes, declaration nodes, dan program node

3. Implementasi AST dalam C++ menggunakan inheritance dan virtual functions, dengan smart pointers untuk memory management

4. Visitor pattern memisahkan algoritma traversal dari struktur node, memungkinkan ekstensibilitas tanpa modifikasi node classes

5. Tree traversal dapat dilakukan dengan pre-order, post-order, atau in-order, masing-masing berguna untuk tujuan berbeda

6. Parser dapat dibangun untuk menghasilkan AST selama proses parsing menggunakan semantic actions

Pemahaman tentang AST sangat penting karena AST menjadi input untuk fase-fase selanjutnya: semantic analysis, optimization, dan code generation. AST proyek subset C (Bagian 9.3) dipakai oleh symbol table (Bab 10), type checking (Bab 11), dan IR generation (Bab 12) dalam pipeline compiler proyek.

## 9.12 Referensi dan Bahan Bacaan Lanjutan

Untuk memperdalam pemahaman tentang AST, mahasiswa disarankan membaca:

- **Dragon Book**: Aho, Lam, Sethi, & Ullman (2006). *Compilers: Principles, Techniques, and Tools* [1] - Bab 2: A Simple Syntax-Directed Translator, Bab 5: Syntax-Directed Translation
- **Engineering a Compiler**: Cooper & Torczon (2011) [5] - Bab 4: Intermediate Representations
- **UC San Diego CSE 231**: Course materials tentang AST construction [3]
- **Johns Hopkins University EN.601.428**: Course tentang AST dan syntax trees [6]
- **GNU Bison Manual**: Dokumentasi tentang semantic actions dan AST building [12]

# 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

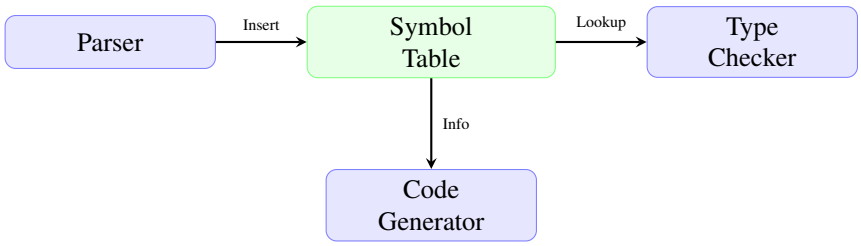
Dalam proyek compiler subset C, symbol table menyimpan informasi tentang identifier (variabel `int`/`float`) yang dideklarasikan dan dipakai dalam program. AST proyek (Bab 9, Bagian 9.3) menjadi input: deklarasi ditambahkan ke symbol table saat traversal; referensi identifier diselesaikan melalui lookup. 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."[13]

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
}
```

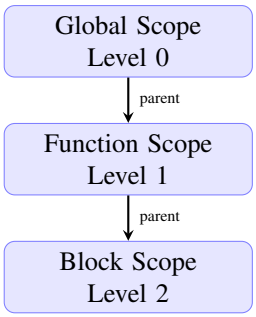
Ini mengimplementasikan aturan scoping: pencarian dimulai dari scope paling dalam (current) dan bergerak ke luar sampai menemukan deklarasi atau mencapai global scope.

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* [1] - Bab 2.7: Symbol Tables
- **Engineering a Compiler**: Cooper & Torczon (2011) [5] - Bab 5: Scoping
- **Nguyen Thanh Vu - Compiler Class Notes**: Semantic Analysis dan Symbol Tables [13]
- **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

# Bab 11

## Type Checking dan Semantic Analysis

### 11.1 Tujuan Pembelajaran

Setelah mempelajari bab ini, mahasiswa diharapkan mampu:

1. Memahami konsep semantic analysis dan perannya dalam kompilator
2. Menjelaskan sistem tipe dan aturan type checking
3. Mengimplementasikan type checker untuk ekspresi aritmatika
4. Memahami type inference dan type compatibility
5. Mengimplementasikan semantic error detection dan reporting
6. Membedakan static type checking dan dynamic type checking

### 11.2 Pendahuluan

Dalam proyek compiler subset C, type checking memverifikasi bahwa operasi dilakukan pada tipe yang kompatibel (`int`, `float`) dan bahwa setiap identifier merujuk ke deklarasi yang valid. Input: AST proyek (Bab 9, Bagian 9.3) dan symbol table proyek (Bab 10). Type checker dipanggil setelah scope resolution, sebelum IR generation (Bab 12).

Setelah fase syntax analysis menghasilkan Abstract Syntax Tree (AST), kompilator perlu memastikan bahwa program tidak hanya valid secara sintaksis, tetapi juga secara semantik. Semantic analysis adalah fase yang memverifikasi bahwa program memenuhi aturan semantik bahasa pemrograman.

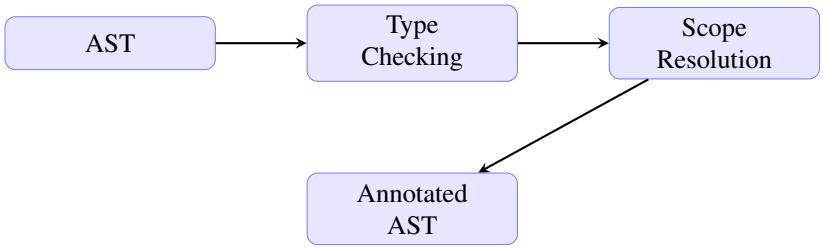
Menurut sumber dari Nguyen Thanh Vu:

> "Type checking: operator operands must be type-compatible. Return types match declared types. Implicit/explicit conversions. Semantic analysis ensures that the parse tree makes sense under language rules."[13]

Semantic analysis bertanggung jawab untuk memeriksa berbagai aspek semantik program:

- **Type Checking**: Memastikan operasi dilakukan pada tipe yang kompatibel
- **Scope Resolution**: Memastikan setiap identifier merujuk ke deklarasi yang valid
- **Name Resolution**: Menyelesaikan referensi variabel, fungsi, dan tipe
- **Contextual Checks**: Memeriksa aturan spesifik bahasa (misalnya: break hanya dalam loop, return type match, dll.)

Gambar 11.1 menunjukkan proses semantic analysis.




Gambar 11.1: Proses semantic analysis

### 11.2.1 Input dan Output Semantic Analysis

Semantic analyzer bekerja dengan:

**Input:**

- Abstract Syntax Tree (AST) dari syntax analyzer
- Symbol table yang sudah dibangun (dari bab sebelumnya)
- Type information dari deklarasi

**Output:**

- Annotated AST dengan informasi tipe pada setiap node
- Symbol table yang dilengkapi dengan informasi tipe
- Daftar semantic errors (jika ada)
- Type-checked program yang siap untuk code generation

## 11.3 Sistem Tipe (Type System)

Sistem tipe adalah kumpulan aturan yang menentukan bagaimana tipe data ditetapkan pada konstruksi program dan operasi apa yang "legal" untuk setiap tipe.

### 11.3.1 Jenis-jenis Type System

#### Static vs Dynamic Typing

- **Static Typing**: Pengecekan tipe dilakukan pada waktu kompilasi. Bahasa seperti C, C++, Java, dan Rust menggunakan static typing. Keuntungannya adalah deteksi error lebih awal dan performa runtime yang lebih baik.

- **Dynamic Typing**: Pengecekan tipe dilakukan pada waktu eksekusi. Bahasa seperti Python, JavaScript, dan Ruby menggunakan dynamic typing. Keuntungannya adalah fleksibilitas lebih tinggi, tetapi error baru terdeteksi saat runtime.

#### Nominal vs Structural Typing

- **Nominal Typing**: Kompatibilitas tipe ditentukan berdasarkan nama tipe yang dideklarasikan. Dua tipe dengan struktur yang sama tetapi nama berbeda dianggap tidak kompatibel. Contoh: Java, C++.

- **Structural Typing**: Kompatibilitas tipe ditentukan berdasarkan struktur tipe (field, method). Jika struktur cocok, tipe dianggap kompatibel meskipun nama berbeda. Contoh: TypeScript, OCaml.

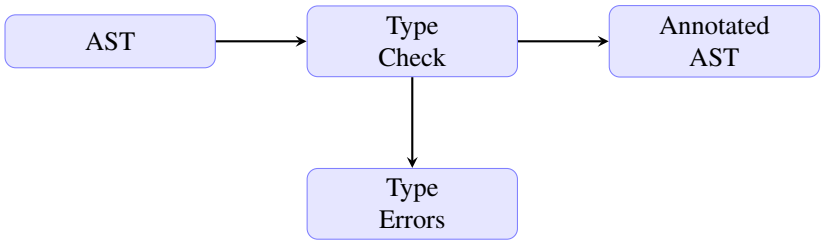
### 11.3.2 Type Hierarchy

Dalam bahasa berorientasi objek, tipe-tipe membentuk hierarki melalui inheritance. Misalnya:

```
Object
|-- Number
|   |-- Integer
|   `-- Float
|-- String
`-- Boolean
```

Hierarki ini memungkinkan subtyping, di mana tipe turunan dapat digunakan di tempat tipe induk (substitution principle).

Gambar 11.2 menunjukkan proses type checking.




Gambar 11.2: Proses type checking

## 11.4 Type Checking

Type checking adalah proses memastikan bahwa program mematuhi aturan tipe bahasa pemrograman. Menurut GeeksforGeeks:

> "The process of ensuring that a program adheres to the language's type rules. Checking for things like 'you can't add an integer to a string', that function calls match the declared parameter types, etc."[1]

### 11.4.1 Aturan Type Checking Dasar

#### Type Checking untuk Ekspresi Aritmatika

Untuk ekspresi aritmatika, aturan dasar meliputi:

1. **Literals**: Setiap literal memiliki tipe intrinsik
   - Integer literal (42) → `int`
   - Float literal (3.14) → `float`
   - String literal ("hello") → `string`
2. **Operasi Aritmatika**: Operan harus kompatibel
   - `int + int` → `int`
   - `float + float` → `float`
   - `int + float` → `float` (dengan implicit conversion)
3. **Assignment**: Tipe ekspresi harus kompatibel dengan tipe variabel
   - `int x = 42;` ✓ (valid)
   - `int x = 3.14;` × (type mismatch, atau perlu explicit cast)

[1] https://www.geeksforgeeks.org/type-checking-in-compiler-design/

**Type Checking untuk Function Calls**

Untuk pemanggilan fungsi, type checker memverifikasi:

1. Jumlah argumen sesuai dengan jumlah parameter
2. Tipe setiap argumen kompatibel dengan tipe parameter yang sesuai
3. Return type dari fungsi sesuai dengan konteks penggunaan

Contoh:

```
int add(int a, int b) { return a + b; }

// Valid call
int result = add(5, 10);  // OK

// Invalid calls
add(5);                   // ERROR: Wrong number of arguments
int x = add(5.0, 10.0);   // ERROR: Type mismatch (float vs int)
```

### 11.4.2 Implementasi Type Checker Sederhana

Berikut adalah contoh struktur data untuk type checker dalam C++:

Listing 11.1: Struktur Data untuk Type System

```
enum class TypeKind {
    INT,
    FLOAT,
    STRING,
    BOOL,
    VOID,
    ARRAY,
    FUNCTION
};

struct Type {
    TypeKind kind;
    // Untuk array: element type
    // Untuk function: parameter types dan return type
    std::vector<Type> subtypes;
};

class TypeChecker {
private:
    SymbolTable* symbolTable;

public:
    TypeChecker(SymbolTable* st) : symbolTable(st) {}

    // Type check sebuah ekspresi
    Type checkExpression(ASTNode* expr);

```

```
    // Type check sebuah statement
    void checkStatement(ASTNode* stmt);

    // Type check sebuah program
    void checkProgram(ASTNode* program);

    // Cek kompatibilitas tipe
    bool isCompatible(Type t1, Type t2);

    // Lakukan implicit conversion jika perlu
    Type promoteType(Type t1, Type t2);
};
```

### Type Checking untuk Binary Operations

Berikut adalah contoh implementasi type checking untuk operasi biner:

Listing 11.2: Type Checking untuk Binary Operations

```
Type TypeChecker::checkBinaryOp(ASTNode* node) {
    ASTBinaryOp* binOp = static_cast<ASTBinaryOp*>(node);

    Type leftType = checkExpression(binOp->left);
    Type rightType = checkExpression(binOp->right);

    switch (binOp->op) {
        case OP_PLUS:
        case OP_MINUS:
        case OP_MULTIPLY:
        case OP_DIVIDE:
            // Operasi aritmatika: int atau float
            if (leftType.kind == TypeKind::INT &&
                rightType.kind == TypeKind::INT) {
                return Type{TypeKind::INT};
            }
            if (leftType.kind == TypeKind::FLOAT ||
                rightType.kind == TypeKind::FLOAT) {
                return promoteType(leftType, rightType);
            }
            reportError("Arithmetic operation on incompatible types");
            break;

        case OP_EQUAL:
        case OP_NOT_EQUAL:
        case OP_LESS:
        case OP_GREATER:
            // Operasi perbandingan: hasilnya boolean
            if (isCompatible(leftType, rightType)) {
                return Type{TypeKind::BOOL};
            }
            reportError("Comparison on incompatible types");
            break;

        default:
```

```
            reportError("Unknown binary operator");
    }

    return Type{TypeKind::VOID}; // Error type
}
```

## 11.5 Type Inference

Type inference adalah kemampuan kompilator untuk secara otomatis menentukan tipe ekspresi tanpa memerlukan anotasi tipe eksplisit dari programmer.

### 11.5.1 Type Inference untuk Literal

Untuk literal, tipe dapat langsung diinfer:

- `42` → `int`
- `3.14` → `float`
- `"hello"` → `string`
- `true` → `bool`

### 11.5.2 Type Inference untuk Operasi

Untuk operasi, tipe hasil diinfer berdasarkan tipe operan:

- `int + int` → `int`
- `int + float` → `float` (promotion)
- `int == int` → `bool`
- `int < float` → `bool`

### 11.5.3 Type Inference untuk Variabel

Dalam beberapa bahasa (seperti C++ dengan `auto`, atau Rust), tipe variabel dapat diinfer dari initializer:

```
auto x = 42;        // x inferred as int
auto y = 3.14;      // y inferred as float
auto z = x + y;     // z inferred as float
```

### 11.5.4 Implementasi Type Inference Sederhana

Listing 11.3: Type Inference untuk Ekspresi

```
Type TypeChecker::inferType(ASTNode* expr) {
    switch (expr->nodeType) {
        case NODE_INT_LITERAL:
            return Type{TypeKind::INT};

        case NODE_FLOAT_LITERAL:
            return Type{TypeKind::FLOAT};

        case NODE_STRING_LITERAL:
            return Type{TypeKind::STRING};

        case NODE_VARIABLE: {
            ASTVariable* var = static_cast<ASTVariable*>(expr);
            Symbol* symbol = symbolTable->lookup(var->name);
            if (symbol) {
                return symbol->type;
            }
            reportError("Undeclared variable: " + var->name);
            return Type{TypeKind::VOID};
        }

        case NODE_BINARY_OP:
            return checkBinaryOp(expr);

        case NODE_FUNCTION_CALL: {
            ASTFunctionCall* call = static_cast<ASTFunctionCall*>(expr);
            Symbol* func = symbolTable->lookup(call->name);
            if (func && func->type.kind == TypeKind::FUNCTION) {
                return func->type.subtypes.back(); // Return type
            }
            reportError("Undeclared function: " + call->name);
            return Type{TypeKind::VOID};
        }

        default:
            reportError("Cannot infer type for node");
            return Type{TypeKind::VOID};
    }
}
```

## 11.6 Type Compatibility

Type compatibility menentukan apakah satu tipe dapat digunakan di tempat tipe lain. Menurut sumber dari TypeScript documentation:

> "Two types are compatible if one can be used in place of another without type errors. This often arises when assigning a value to a variable, passing arguments

to a function, etc."[2]

### 11.6.1 Aturan Type Compatibility

#### Exact Match

Dua tipe yang identik selalu kompatibel:

```
int x = 42;         // int = int (OK)
float y = 3.14;     // float = float (OK)
```

#### Implicit Conversion (Type Promotion)

Beberapa bahasa mengizinkan implicit conversion antara tipe yang "dekat":

```
int x = 42;
float y = x;        // int -> float (promotion) OK

float a = 3.14;
int b = a;          // float -> int (downgrade) ERROR atau perlu cast
```

Aturan umum untuk promotion:

- `int` → `float` (biasanya diizinkan)
- `int` → `long` (diizinkan)
- `float` → `double` (diizinkan)
- `float` → `int` (biasanya memerlukan explicit cast)

#### Subtyping

Dalam bahasa berorientasi objek, tipe turunan kompatibel dengan tipe induk:

```
class Animal { }
class Dog extends Animal { }

Animal a = new Dog();  // Dog is subtype of Animal (OK)
```

### 11.6.2 Implementasi Type Compatibility Check

Listing 11.4: Implementasi Type Compatibility

```
bool TypeChecker::isCompatible(Type t1, Type t2) {
    // Exact match
    if (t1.kind == t2.kind) {
        // Untuk tipe kompleks, perlu cek lebih detail
        if (t1.kind == TypeKind::ARRAY) {
            return isCompatible(t1.subtypes[0], t2.subtypes[0]);
```

[2] https://www.typescriptlang.org/docs/handbook/type-compatibility.html

```
        }
        if (t1.kind == TypeKind::FUNCTION) {
            // Cek parameter types dan return type
            if (t1.subtypes.size() != t2.subtypes.size()) {
                return false;
            }
            for (size_t i = 0; i < t1.subtypes.size() - 1; i++) {
                if (!isCompatible(t1.subtypes[i], t2.subtypes[i])) {
                    return false;
                }
            }
            return isCompatible(
                t1.subtypes.back(),
                t2.subtypes.back()
            );
        }
        return true;
    }

    // Implicit conversion rules
    if (t1.kind == TypeKind::INT && t2.kind == TypeKind::FLOAT) {
        return true;  // int dapat di-promote ke float
    }

    // Subtyping (jika ada)
    // ... implementasi subtyping check

    return false;
}

Type TypeChecker::promoteType(Type t1, Type t2) {
    // Jika salah satu float, hasilnya float
    if (t1.kind == TypeKind::FLOAT || t2.kind == TypeKind::FLOAT) {
        return Type{TypeKind::FLOAT};
    }
    // Default: int
    return Type{TypeKind::INT};
}
```

## 11.7 Semantic Error Detection dan Reporting

Semantic analyzer harus mendeteksi berbagai jenis error dan memberikan pesan error yang jelas dan informatif.

### 11.7.1 Jenis-jenis Semantic Error

#### Undeclared Variable

Error terjadi ketika variabel digunakan sebelum dideklarasi:

```
x = 42;  // Error: 'x' is not declared
int x;
```

### Type Mismatch

Error terjadi ketika tipe tidak kompatibel:

```
int x = "hello";  // Error: cannot assign string to int
int y = 3.14;     // Error: cannot assign float to int (tanpa cast)
```

### Undefined Function

Error terjadi ketika fungsi dipanggil tetapi tidak didefinisikan:

```
int result = add(5, 10);  // Error: function 'add' is not defined
```

### Wrong Number of Arguments

Error terjadi ketika jumlah argumen tidak sesuai:

```
int add(int a, int b) { return a + b; }
int x = add(5);  // Error: expected 2 arguments, got 1
```

### Return Type Mismatch

Error terjadi ketika return type tidak sesuai dengan deklarasi:

```
int getValue() {
    return "hello";  // Error: function should return int
}
```

## 11.7.2 Error Reporting yang Informatif

Pesan error yang baik harus:

- Menunjukkan lokasi error (baris, kolom)
- Menjelaskan jenis error dengan jelas
- Memberikan konteks yang relevan
- Menyarankan solusi jika memungkinkan

Contoh implementasi error reporting:

Listing 11.5: Error Reporting System

```
class ErrorReporter {
private:
    std::vector<Error> errors;

public:
```

```
    void reportError(const std::string& message,
                     int line, int column) {
        errors.push_back(Error{message, line, column});
        std::cerr << "Error at line " << line
                  << ", column " << column
                  << ": " << message << std::endl;
    }

    void reportTypeError(const std::string& expected,
                         const std::string& got,
                         int line, int column) {
        std::string msg = "Type mismatch: expected " + expected +
                          ", got " + got;
        reportError(msg, line, column);
    }

    bool hasErrors() const {
        return !errors.empty();
    }

    const std::vector<Error>& getErrors() const {
        return errors;
    }
};
```

## 11.8 Integrasi dengan Symbol Table

Type checking bekerja erat dengan symbol table yang telah dibangun pada fase sebelumnya. Symbol table menyediakan informasi tentang:

- Tipe setiap variabel yang dideklarasikan
- Tipe parameter dan return type setiap fungsi
- Scope di mana setiap identifier dideklarasikan

Contoh integrasi:

Listing 11.6: Type Checking dengan Symbol Table

```
Type TypeChecker::checkVariable(ASTVariable* var) {
    Symbol* symbol = symbolTable->lookup(var->name);

    if (!symbol) {
        errorReporter->reportError(
            "Undeclared variable: " + var->name,
            var->line, var->column
        );
        return Type{TypeKind::VOID};
    }

    if (symbol->kind != SymbolKind::VARIABLE) {
```

```
        errorReporter->reportError(
            var->name + " is not a variable",
            var->line, var->column
        );
        return Type{TypeKind::VOID};
    }

    return symbol->type;
}

void TypeChecker::checkAssignment(ASTAssignment* assign) {
    Type varType = checkVariable(assign->variable);
    Type exprType = checkExpression(assign->expression);

    if (!isCompatible(varType, exprType)) {
        errorReporter->reportTypeError(
            typeToString(varType),
            typeToString(exprType),
            assign->line, assign->column
        );
    }
}
```

## 11.9 Type Checking untuk Kontrol Flow

Type checker juga perlu memverifikasi kontrol flow statements:

### 11.9.1 If Statement

Listing 11.7: Type Checking untuk If Statement

```
void TypeChecker::checkIfStatement(ASTIf* ifStmt) {
    Type condType = checkExpression(ifStmt->condition);

    if (condType.kind != TypeKind::BOOL) {
        errorReporter->reportError(
            "If condition must be boolean",
            ifStmt->line, ifStmt->column
        );
    }

    checkStatement(ifStmt->thenBranch);
    if (ifStmt->elseBranch) {
        checkStatement(ifStmt->elseBranch);
    }
}
```

### 11.9.2 While Loop

Listing 11.8: Type Checking untuk While Loop

```
void TypeChecker::checkWhileLoop(ASTWhile* whileLoop) {
    Type condType = checkExpression(whileLoop->condition);

    if (condType.kind != TypeKind::BOOL) {
        errorReporter->reportError(
            "While condition must be boolean",
            whileLoop->line, whileLoop->column
        );
    }

    checkStatement(whileLoop->body);
}
```

### 11.9.3 Return Statement

Listing 11.9: Type Checking untuk Return Statement

```
void TypeChecker::checkReturn(ASTReturn* ret, Type expectedReturnType) {
    if (ret->expression) {
        Type exprType = checkExpression(ret->expression);
        if (!isCompatible(expectedReturnType, exprType)) {
            errorReporter->reportTypeError(
                typeToString(expectedReturnType),
                typeToString(exprType),
                ret->line, ret->column
            );
        }
    } else {
        if (expectedReturnType.kind != TypeKind::VOID) {
            errorReporter->reportError(
                "Function must return a value",
                ret->line, ret->column
            );
        }
    }
}
```

## 11.10 Annotated AST

Setelah type checking, setiap node AST di-annotate dengan informasi tipe. Ini memudahkan fase selanjutnya (code generation) untuk mengetahui tipe setiap ekspresi.

Listing 11.10: Annotated AST Node

```
class ASTNode {
public:
    NodeType nodeType;
    int line, column;
    Type type;  // Annotated type (setelah type checking)

```

```
    virtual ~ASTNode() = default;
};

// Contoh penggunaan
Type TypeChecker::checkExpression(ASTNode* expr) {
    Type t = inferType(expr);
    expr->type = t;  // Annotate node dengan type
    return t;
}
```

## 11.11 Kesimpulan

Dalam bab ini, kita telah mempelajari:

1. Semantic analysis memverifikasi bahwa program memenuhi aturan semantik bahasa
2. Type checking memastikan operasi dilakukan pada tipe yang kompatibel
3. Type inference memungkinkan kompilator menentukan tipe secara otomatis
4. Type compatibility menentukan apakah satu tipe dapat digunakan di tempat tipe lain
5. Semantic error detection dan reporting memberikan feedback yang jelas kepada programmer
6. Type checking terintegrasi dengan symbol table untuk mengakses informasi deklarasi
7. Annotated AST menyimpan informasi tipe untuk digunakan pada fase selanjutnya

Pemahaman tentang type checking dan semantic analysis ini penting karena memastikan bahwa program yang dikompilasi tidak hanya valid secara sintaksis, tetapi juga benar secara semantik sebelum masuk ke fase code generation. Type checking proyek subset C memakai AST proyek (Bab 9) dan symbol table (Bab 10); annotated AST menjadi input untuk IR generation (Bab 12) dan code generation (Bab 14).

## 11.12 Referensi dan Bahan Bacaan Lanjutan

Untuk memperdalam pemahaman tentang type checking dan semantic analysis, mahasiswa disarankan membaca:

- **Dragon Book**: Aho, Lam, Sethi, & Ullman (2006). *Compilers: Principles, Techniques, and Tools* [1] - Bab 6: Type Checking
- **Engineering a Compiler**: Cooper & Torczon (2011) [5] - Bab 4: Context-Sensitive Analysis

- **Nguyen Thanh Vu - Compiler Class Notes**: Semantic Analysis [13]
- **GeeksforGeeks**: Type Checking in Compiler Design [3]
- **Wikipedia - Type Inference**: [4]
- **Wikipedia - Type System**: [5]
- **TypeScript Handbook - Type Compatibility**: [6]

[3] https://www.geeksforgeeks.org/type-checking-in-compiler-design/
[4] https://en.wikipedia.org/wiki/Type_inference
[5] https://en.wikipedia.org/wiki/Type_system
[6] https://www.typescriptlang.org/docs/handbook/type-compatibility.html

# Bab 12

## Intermediate Code Generation

### 12.1 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

### 12.2 Pendahuluan

Dalam proyek compiler subset C, IR generation mengubah annotated AST (output type checking Bab 11) menjadi three-address code (TAC) atau quadruples. Format IR untuk subset C: instruksi assignment (`x = y op z`), load/store identifier, literal, dan print. IR proyek dipakai oleh code generation (Bab 14) dan optimasi (Bab 15); struktur dan generator IR dirujuk dari folder proyek.

Setelah fase Analisis Semantik (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 **Generasi Kode Intermediate (Intermediate Code Generation)**.

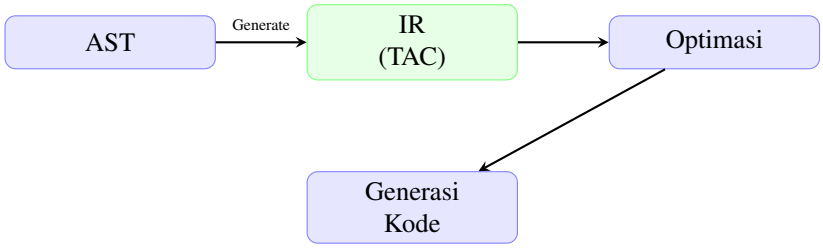
Menurut sumber dari OpenGenus:

> "Intermediate code generation transforms AST to IR (three-address code, bytecode, etc.). Design and generate intermediate code representations (e.g., three-address code, DAGs)."[8]

Representasi Intermediate (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 12.1 menunjukkan posisi IR dalam pipeline kompilator.




Gambar 12.1: Posisi IR dalam pipeline kompilator

### 12.2.1 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.

## 12.3 Format Intermediate Representation

Terdapat berbagai format IR yang umum digunakan dalam kompilator modern:

### 12.3.1 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:

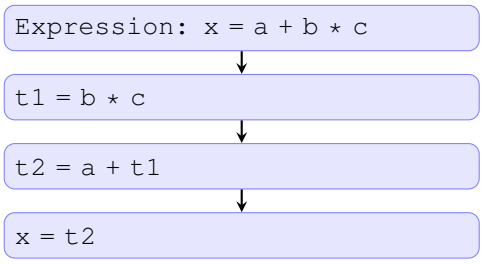
- Sederhana dan mudah dipahami
- Mirip dengan assembly code
- Memudahkan optimasi
- Mudah di-generate dari AST

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 12.2 menunjukkan contoh three-address code untuk ekspresi kompleks.




Gambar 12.2: Contoh three-address code

### 12.3.2 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)

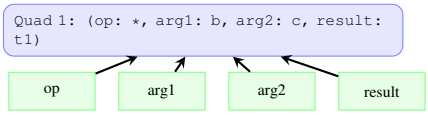
Contoh quadruple untuk `x = a + b * c`:

```
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 12.3 menunjukkan struktur quadruple.




Gambar 12.3: Struktur quadruple

### 12.3.3 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

## 12.4 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.

### 12.4.1 Struktur Data untuk TAC

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

Listing 12.1: 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";
        }
    }
};
```

### 12.4.2 Generator untuk Ekspresi

Generator untuk ekspresi aritmatika bekerja secara recursive:

Listing 12.2: 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;
    }
};
```

### 12.4.3 Generator untuk Assignment

Assignment statement menghasilkan instruksi assignment:

Listing 12.3: 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;
    }
};
```

## 12.5 Handling Control Flow Statements

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

### 12.5.1 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 12.4: 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
```

### 12.5.2 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 12.5: 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));

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

        return "";
    }
};
```

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
```

### 12.5.3 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
```

## 12.6 Handling Function Calls

Function calls memerlukan:

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

Listing 12.6: 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;
    }
};
```

## 12.7 Optimasi Dasar: Common Subexpression Elimination

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

### 12.7.1 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
```

### 12.7.2 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 12.7: 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;
        }
    }
}
```

### 12.7.3 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
```

## 12.8 Integrasi dengan Fase Sebelumnya

Intermediate code generation terintegrasi dengan fase-fase sebelumnya:

### 12.8.1 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

### 12.8.2 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

## 12.9 Contoh Lengkap: Ekspresi Kompleks

Mari kita lihat contoh lengkap generasi TAC untuk ekspresi kompleks:

### 12.9.1 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;
}
```

### 12.9.2 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
```

## 12.10 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. IR proyek subset C (TAC/quadruples dari AST proyek) menjadi input untuk code generation (Bab 14) dan optimasi (Bab 15) dalam pipeline compiler proyek.

## 12.11 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* [1] - Bab 6: Intermediate-Code Generation
- **Engineering a Compiler**: Cooper & Torczon (2011) [5] - 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

# Bab 13

## Runtime Environment dan Memory Management

### 13.1 Tujuan Pembelajaran

Setelah mempelajari bab ini, mahasiswa diharapkan mampu:

1. Memahami konsep runtime environment dan perannya dalam eksekusi program
2. Menjelaskan struktur activation record (stack frame) dan komponen-komponennya
3. Mengimplementasikan simulator runtime stack untuk function calls
4. Memahami memory layout: static, stack, dan heap
5. Menjelaskan mekanisme heap management dan garbage collection
6. Mengimplementasikan manajemen memory untuk runtime environment sederhana

### 13.2 Pendahuluan

Runtime environment adalah konteks di mana program yang telah dikompilasi dieksekusi. Menurut sumber dari StudyLib:

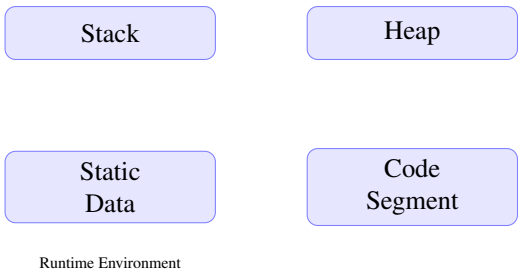
> "Runtime environment: stack, heap, activation records; garbage collection intro. Managing run-time structures (activation records, memory layout, symbol tables)."[14]

Runtime environment mencakup:

- **Memory Organization**: Bagaimana memory diorganisir dan dialokasikan untuk berbagai jenis data
- **Calling Conventions**: Mekanisme pemanggilan fungsi, passing parameter, dan return values

- **Scope Management**: Bagaimana variabel diakses berdasarkan scope-nya (local, non-local, global)
- **Memory Management**: Alokasi dan dealokasi memory untuk variabel dan data structures

Gambar 13.1 menunjukkan komponen-komponen runtime environment.




Gambar 13.1: Komponen-komponen runtime environment

Asumsi runtime untuk proyek compiler subset C (stack/activation record, layout variabel) dibahas di Bagian 13.8 dan dipakai oleh code generation (Bab 14).

Runtime environment harus dirancang dengan hati-hati karena mempengaruhi:

- Efisiensi eksekusi program
- Keamanan memory (memory safety)
- Kemampuan mendukung fitur bahasa (recursion, nested functions, closures, dll.)
- Portabilitas antar platform

## 13.3 Memory Layout

Program yang dieksekusi memiliki memory layout yang terorganisir menjadi beberapa region. Setiap region memiliki karakteristik dan tujuan penggunaan yang berbeda.

### 13.3.1 Memory Regions

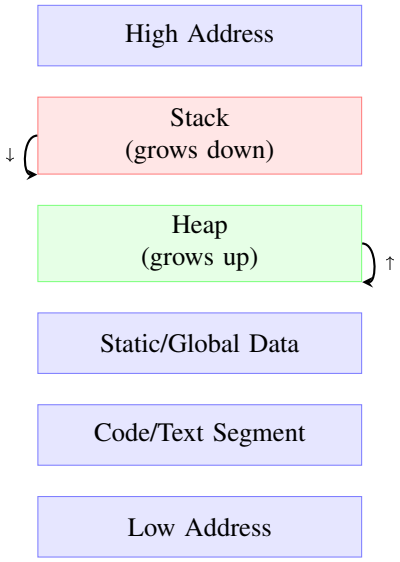
Memory address space program biasanya dibagi menjadi beberapa region utama:

1. **Code/Text Segment**: Berisi instruksi machine code yang dihasilkan compiler. Region ini biasanya read-only dan tidak dapat dimodifikasi saat runtime.
2. **Static/Global Data**: Berisi variabel global dan static yang dialokasikan pada compile time. Region ini memiliki ukuran tetap dan alamat yang diketahui saat compile time.

3. **Stack**: Region untuk activation records (stack frames) dari fungsi-fungsi yang sedang aktif. Stack tumbuh ke bawah (dari high address ke low address) dan dikelola secara otomatis.

4. **Heap**: Region untuk dynamic memory allocation. Heap tumbuh ke atas (dari low address ke high address) dan dikelola secara manual atau melalui garbage collector.

Gambar 13.2 menunjukkan layout memory secara visual dengan TikZ.




Gambar 13.2: Memory layout program

Gambar 13.3 menunjukkan layout memory yang khas:

```
High Address
    +------------------+
    |   Command Line   |
    |     Arguments    |
    +------------------+
    |       Stack      | <- Tumbuh ke bawah
    |    (grows down)  |
    |                  |
    |        ↓         |
    |                  |
    |        ↑         |
    |    (grows up)    |
    |       Heap       | <- Tumbuh ke atas
    +------------------+
    |  BSS (uninit)    |
    |  Data (init)     |
    +------------------+
    |  Text/Code       | <- Read-only
Low Address
```

Gambar 13.3: Memory layout khas untuk program yang dieksekusi

### 13.3.2 Static Memory Allocation

Static memory allocation terjadi pada compile time. Variabel yang dialokasikan secara static memiliki:

- Alamat yang tetap dan diketahui saat compile time
- Lifetime yang sama dengan program (dari awal hingga akhir eksekusi)
- Tidak memerlukan runtime overhead untuk alokasi/dealokasi

Contoh variabel static:

- Variabel global: `int global_var;`
- Variabel static lokal: `static int counter;`
- String literals dan konstanta

Keuntungan static allocation:

- Sangat efisien (tidak ada overhead runtime)
- Deterministik (alamat diketahui saat compile time)
- Tidak ada risiko memory leak

Keterbatasan:

- Tidak mendukung recursion dengan baik
- Ukuran harus diketahui saat compile time
- Tidak fleksibel untuk dynamic data structures

### 13.3.3 Stack-Based Memory Allocation

Stack digunakan untuk activation records dari fungsi-fungsi yang sedang aktif. Stack allocation memiliki karakteristik:

- **Automatic**: Alokasi dan dealokasi terjadi otomatis saat fungsi dipanggil dan kembali
- **LIFO**: Last In First Out - fungsi terakhir dipanggil adalah yang pertama kembali
- **Fast**: Alokasi/dealokasi sangat cepat (hanya mengubah stack pointer)
- **Limited Lifetime**: Data di stack hanya hidup selama fungsi aktif

Stack sangat cocok untuk:

- Local variables
- Function parameters
- Return addresses
- Temporary values

Contoh penggunaan stack:

Listing 13.1: Contoh program yang menggunakan stack

```
int factorial(int n) {
    if (n <= 1) return 1;
    int temp = n * factorial(n - 1);  // Recursive call
    return temp;
}

int main() {
    int result = factorial(5);  // Stack frames untuk main dan factorial
    return 0;
}
```

### 13.3.4 Heap-Based Memory Allocation

Heap digunakan untuk dynamic memory allocation yang tidak dapat ditangani oleh stack. Heap allocation memiliki karakteristik:

- **Manual Management**: Programmer harus secara eksplisit mengalokasikan dan membebaskan memory
- **Flexible Lifetime**: Object di heap dapat hidup lebih lama dari fungsi yang membuatnya
- **Variable Size**: Ukuran dapat ditentukan saat runtime
- **Slower**: Alokasi/dealokasi lebih lambat dibanding stack

Heap digunakan untuk:

- Dynamic arrays dan data structures
- Objects yang harus hidup lebih lama dari fungsi pembuatnya
- Shared data structures
- Large objects yang tidak muat di stack

Contoh penggunaan heap:

Listing 13.2: Contoh penggunaan heap

```
int* createArray(int size) {
    int* arr = new int[size];  // Alokasi di heap
    return arr;  // Pointer ke heap, valid setelah fungsi kembali
}

void useArray() {
    int* myArray = createArray(100);
    // Gunakan array...
    delete[] myArray;  // Dealokasi manual
}
```

## 13.4 Activation Records (Stack Frames)

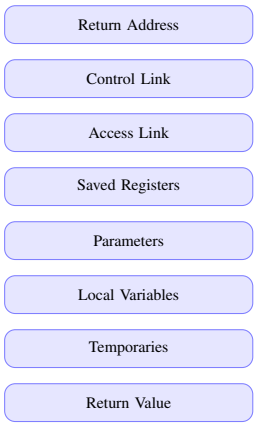
Activation record (juga disebut stack frame) adalah struktur data yang digunakan untuk menyimpan informasi tentang eksekusi satu fungsi. Setiap kali fungsi dipanggil, activation record baru dibuat di stack.

### 13.4.1 Komponen Activation Record

Activation record biasanya berisi komponen-komponen berikut:

1. **Return Address**: Alamat instruksi di caller yang harus dieksekusi setelah fungsi kembali
2. **Control Link (Dynamic Link)**: Pointer ke activation record dari caller (fungsi yang memanggil)
3. **Access Link (Static Link)**: Pointer ke activation record dari enclosing scope (untuk nested functions)
4. **Saved Registers**: Nilai register yang harus disimpan dan dikembalikan setelah fungsi selesai
5. **Parameters**: Nilai parameter yang diteruskan ke fungsi (actual parameters)
6. **Local Variables**: Variabel lokal yang dideklarasikan dalam fungsi
7. **Temporary Values**: Nilai sementara yang digunakan selama komputasi dalam fungsi
8. **Return Value**: Nilai yang dikembalikan fungsi (jika ada)

Gambar 13.4 menunjukkan struktur activation record secara visual.
Gambar 13.5 menunjukkan struktur activation record yang khas:




Gambar 13.4: Struktur activation record

```
High Address
    +----------------------+
    |  Return Address      |
    +----------------------+
    |  Control Link (FP)   | -> Activation record caller
    +----------------------+
    |  Access Link         | -> Enclosing scope (if nested)
    +----------------------+
    |  Saved Registers     |
    +----------------------+
    |  Parameters          |
    |    param1            |
    |    param2            |
    +----------------------+
    |  Local Variables     |
    |    local1            |
    |    local2            |
    +----------------------+
    |  Temporaries         |
    |    temp1             |
    |    temp2             |
    +----------------------+
    |  Return Value        |
Low Address
```

Gambar 13.5: Struktur activation record (stack frame)

### 13.4.2 Calling Sequence

Calling sequence adalah urutan instruksi yang dihasilkan compiler untuk memanggil fungsi. Terdapat dua bagian:

#### Caller Sequence (Prologue)

Instruksi yang dijalankan oleh caller sebelum memanggil fungsi:

1. Evaluasi actual parameters (dari kanan ke kiri atau kiri ke kanan, tergantung calling convention)
2. Push parameters ke stack (atau pass melalui register)
3. Save caller-saved registers
4. Push return address
5. Transfer control ke callee (CALL instruction)

#### Callee Sequence (Prologue)

Instruksi yang dijalankan oleh callee di awal fungsi:

1. Save frame pointer (FP) dari caller
2. Set FP baru ke current stack pointer (SP)
3. Allocate space untuk local variables (adjust SP)
4. Save callee-saved registers (jika diperlukan)

#### Return Sequence (Epilogue)

Instruksi yang dijalankan saat fungsi kembali:

1. Place return value (di register atau stack)
2. Restore callee-saved registers
3. Restore SP (deallocate local variables)
4. Restore FP dari control link
5. Restore return address
6. Return control ke caller (RET instruction)

### 13.4.3 Contoh Calling Sequence

Mari kita lihat contoh calling sequence untuk program sederhana:

Listing 13.3: Contoh program untuk analisis calling sequence

```c
int add(int x, int y) {
    int sum = x + y;
    return sum;
}

int main() {
    int a = 5;
    int b = 10;
    int result = add(a, b);
    return 0;
}
```

Assembly code yang dihasilkan (simplified):

```assembly
main:
    push rbp                ; Save caller's frame pointer
    mov rbp, rsp            ; Set new frame pointer
    sub rsp, 16             ; Allocate space for locals (a, b, result)

    mov [rbp-4], 5          ; a = 5
    mov [rbp-8], 10         ; b = 10

    ; Call add(a, b)
    mov eax, [rbp-8]        ; Load b
    push eax                ; Push parameter 2
    mov eax, [rbp-4]        ; Load a
    push eax                ; Push parameter 1
    call add                ; Call function
    add rsp, 8              ; Clean up parameters
    mov [rbp-12], eax       ; result = return value

    mov eax, 0              ; return 0
    mov rsp, rbp            ; Restore stack pointer
    pop rbp                  ; Restore frame pointer
    ret                      ; Return

add:
    push rbp                ; Save caller's frame pointer
    mov rbp, rsp            ; Set new frame pointer
    sub rsp, 4              ; Allocate space for local (sum)

    mov eax, [rbp+8]        ; Load x (parameter 1)
    add eax, [rbp+12]       ; Add y (parameter 2)
    mov [rbp-4], eax        ; sum = x + y
    mov eax, [rbp-4]        ; Load sum for return

    mov rsp, rbp            ; Restore stack pointer
    pop rbp                  ; Restore frame pointer
    ret                      ; Return
```

## 13.5 Implementasi Runtime Stack Simulator

Untuk memahami runtime stack dengan lebih baik, kita akan mengimplementasikan simulator sederhana dalam C++.

### 13.5.1 Struktur Data Activation Record

Listing 13.4: Struktur data untuk activation record

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

// Representasi activation record
struct ActivationRecord {
    std::string function_name;           // Nama fungsi
    void* return_address;                // Return address (simulated)
    ActivationRecord* control_link;      // Pointer ke caller's AR
    ActivationRecord* access_link;       // Pointer ke enclosing scope

    // Local variables dan parameters
    std::unordered_map<std::string, int> locals;
    std::unordered_map<std::string, int> parameters;

    // Return value
    int return_value;

    ActivationRecord(const std::string& name,
                     ActivationRecord* caller = nullptr)
        : function_name(name),
          return_address(nullptr),
          control_link(caller),
          access_link(nullptr),
          return_value(0) {}
};
```

### 13.5.2 Stack Manager

Listing 13.5: Implementasi runtime stack manager

```cpp
class RuntimeStack {
private:
    ActivationRecord* top;  // Top of stack (current activation)
    int frame_count;

public:
    RuntimeStack() : top(nullptr), frame_count(0) {}

    // Push activation record baru (function call)
    void pushFrame(const std::string& function_name) {
        ActivationRecord* new_frame =
```

```
            new ActivationRecord(function_name, top);
        new_frame->access_link = top;  // Simplified: same as control
↪ link
        top = new_frame;
        frame_count++;

        std::cout << ">>> Called: " << function_name
                  << " (Frame #" << frame_count << ")\n";
        printStack();
    }

    // Pop activation record (function return)
    void popFrame() {
        if (top == nullptr) {
            std::cerr << "Error: Cannot pop from empty stack!\n";
            return;
        }

        std::string func_name = top->function_name;
        int ret_val = top->return_value;

        ActivationRecord* old_top = top;
        top = top->control_link;
        delete old_top;
        frame_count--;

        std::cout << "<<< Returned from: " << func_name
                  << " (return value: " << ret_val << ")\n";
        printStack();
    }

    // Get current activation record
    ActivationRecord* getCurrentFrame() {
        return top;
    }

    // Set local variable di current frame
    void setLocal(const std::string& name, int value) {
        if (top == nullptr) {
            std::cerr << "Error: No active frame!\n";
            return;
        }
        top->locals[name] = value;
        std::cout << "  Set local: " << name << " = " << value << "\n";
    }

    // Get local variable dari current frame atau enclosing scopes
    int getLocal(const std::string& name) {
        ActivationRecord* frame = top;
        while (frame != nullptr) {
            if (frame->locals.find(name) != frame->locals.end()) {
                return frame->locals[name];
            }
            frame = frame->access_link;  // Check enclosing scope
```

```
        }
        std::cerr << "Error: Variable '" << name
                  << "' not found!\n";
        return 0;
    }

    // Set parameter
    void setParameter(const std::string& name, int value) {
        if (top == nullptr) {
            std::cerr << "Error: No active frame!\n";
            return;
        }
        top->parameters[name] = value;
        std::cout << "  Set parameter: " << name << " = " << value << "\n
↪ ";
    }

    // Set return value
    void setReturnValue(int value) {
        if (top == nullptr) {
            std::cerr << "Error: No active frame!\n";
            return;
        }
        top->return_value = value;
        std::cout << "  Set return value: " << value << "\n";
    }

    // Print stack untuk debugging
    void printStack() {
        std::cout << "Stack (top to bottom):\n";
        ActivationRecord* frame = top;
        int level = 0;
        while (frame != nullptr) {
            std::cout << "  [" << level << "] "
                      << frame->function_name << "\n";
            frame = frame->control_link;
            level++;
        }
        std::cout << "\n";
    }

    ~RuntimeStack() {
        while (top != nullptr) {
            popFrame();
        }
    }
};
```

### 13.5.3 Contoh Penggunaan Simulator

Listing 13.6: Contoh penggunaan runtime stack simulator

```
int main() {
```

```
    RuntimeStack stack;

    // Simulasi: main() calls factorial(5)
    stack.pushFrame("main");
    stack.setLocal("n", 5);

    // Call factorial(5)
    stack.pushFrame("factorial");
    stack.setParameter("n", 5);

    // Recursive call: factorial(4)
    stack.pushFrame("factorial");
    stack.setParameter("n", 4);

    // Recursive call: factorial(3)
    stack.pushFrame("factorial");
    stack.setParameter("n", 3);

    // Base case: factorial(1) returns 1
    stack.setReturnValue(1);
    stack.popFrame();

    // factorial(3) = 3 * factorial(2) = 3 * 2 = 6
    // (simplified, actual would need more frames)
    stack.setReturnValue(6);
    stack.popFrame();

    stack.setReturnValue(24);
    stack.popFrame();

    stack.setReturnValue(120);
    stack.popFrame();

    // main returns
    stack.popFrame();

    return 0;
}
```

## 13.6 Heap Memory Management

Heap memory management adalah proses mengalokasikan dan membebaskan memory di heap secara dinamis. Terdapat dua pendekatan utama:

### 13.6.1 Manual Memory Management

Dalam manual memory management (seperti C/C++), programmer harus secara eksplisit:

- Mengalokasikan memory: `malloc()`, `new`
- Membebaskan memory: `free()`, `delete`

Keuntungan:

- Kontrol penuh atas memory
- Tidak ada overhead garbage collector
- Predictable performance

Kekurangan:

- Rentan terhadap memory leaks
- Dangling pointers
- Double free errors
- Memory fragmentation

### 13.6.2 Allocation Algorithms

Heap manager menggunakan berbagai algoritma untuk mengalokasikan memory:

#### First Fit

Mencari block pertama yang cukup besar:

- Cepat (tidak perlu mencari semua)
- Dapat menyebabkan fragmentation

#### Best Fit

Mencari block terkecil yang cukup besar:

- Mengurangi wasted space
- Lebih lambat (harus mencari semua)
- Dapat menyebabkan banyak small fragments

#### Worst Fit

Mencari block terbesar:

- Meninggalkan large free blocks
- Dapat mengurangi fragmentation kecil

#### Buddy Allocation

Membagi memory menjadi blocks dengan ukuran power of 2:

- Mudah untuk merge adjacent blocks
- Dapat menyebabkan internal fragmentation

### 13.6.3 Garbage Collection

Garbage collection adalah automatic memory management yang membebaskan memory yang tidak lagi digunakan. Menurut sumber dari StudyLib:

> "Runtime environment: stack, heap, activation records; garbage collection intro. Managing run-time structures (activation records, memory layout, symbol tables)."[14]

#### Konsep Garbage Collection

Garbage collector mengidentifikasi dan membebaskan memory yang tidak lagi dapat diakses (unreachable) dari program. Object dianggap garbage jika:

- Tidak ada pointer/reference yang menunjuk ke object tersebut
- Tidak dapat diakses dari root set (stack, global variables, registers)

#### Strategi Garbage Collection

#### Mark and Sweep

1. **Mark Phase**: Traverse dari root set, mark semua reachable objects
2. **Sweep Phase**: Scan semua objects, free yang tidak di-mark

Keuntungan:

- Dapat menangani cyclic references
- Tidak memerlukan memory compaction

Kekurangan:

- Dapat menyebabkan fragmentation
- Stop-the-world pauses

**Reference Counting** Setiap object memiliki counter yang menghitung jumlah reference ke object tersebut. Ketika counter menjadi 0, object di-free.

Keuntungan:

- Incremental (tidak perlu stop-the-world)
- Memory dibebaskan segera saat tidak digunakan

Kekurangan:

- Tidak dapat menangani cyclic references
- Overhead untuk setiap assignment

**Copying Collector (Generational GC)** Memory dibagi menjadi young generation dan old generation. Young objects yang survive beberapa collections dipromote ke old generation.

Keuntungan:

- Efisien untuk short-lived objects
- Automatic compaction

Kekurangan:

- Memerlukan extra memory (copying)
- Overhead untuk promotion

### Implementasi Sederhana Mark and Sweep

Berikut adalah implementasi sederhana mark-and-sweep garbage collector:

Listing 13.7: Implementasi sederhana mark-and-sweep GC

```
#include <vector>
#include <unordered_set>

class GCObject {
public:
    bool marked;
    std::vector<GCObject*> references;

    GCObject() : marked(false) {}
    virtual ~GCObject() {}

    void addReference(GCObject* obj) {
        references.push_back(obj);
    }
};

class SimpleGC {
```

```
private:
    std::vector<GCObject*> heap;
    std::vector<GCObject*> roots;  // Root set (stack, globals)

public:
    // Allocate new object
    GCObject* allocate() {
        GCObject* obj = new GCObject();
        heap.push_back(obj);
        return obj;
    }

    // Add to root set
    void addRoot(GCObject* obj) {
        roots.push_back(obj);
    }

    // Mark phase: mark all reachable objects
    void mark() {
        std::vector<GCObject*> worklist = roots;

        while (!worklist.empty()) {
            GCObject* obj = worklist.back();
            worklist.pop_back();

            if (!obj->marked) {
                obj->marked = true;
                // Add all references to worklist
                for (GCObject* ref : obj->references) {
                    if (!ref->marked) {
                        worklist.push_back(ref);
                    }
                }
            }
        }
    }

    // Sweep phase: free unmarked objects
    void sweep() {
        auto it = heap.begin();
        while (it != heap.end()) {
            GCObject* obj = *it;
            if (!obj->marked) {
                delete obj;
                it = heap.erase(it);
            } else {
                obj->marked = false;  // Reset for next collection
                ++it;
            }
        }
    }

    // Run garbage collection
    void collect() {
```

```
        mark();
        sweep();
    }

    ~SimpleGC() {
        for (GCObject* obj : heap) {
            delete obj;
        }
    }
};
```

## 13.7 Memory Layout untuk Program Contoh

Mari kita analisis memory layout untuk program yang lebih kompleks:

Listing 13.8: Program contoh untuk analisis memory layout

```
int global_var = 100;           // Static/Global
static int static_var = 200;    // Static

int* createArray(int size) {    // Function
    int* arr = new int[size];   // Heap allocation
    return arr;
}

int factorial(int n) {          // Function
    static int counter = 0;     // Static local
    counter++;

    if (n <= 1) return 1;
    int temp = n * factorial(n - 1);  // Stack: recursive
    return temp;
}

int main() {                    // Function
    int local_a = 10;           // Stack: local variable
    int local_b = 20;           // Stack: local variable

    int* heap_array = createArray(100);  // Heap allocation

    int result = factorial(5);  // Stack: recursive calls

    delete[] heap_array;        // Heap deallocation
    return 0;
}
```

Memory layout saat eksekusi:

```
High Address
    +---------------------+
    |  Stack (grows down) |
    |                     |
    |  [factorial(1) AR]  | <- Top of stack
```

```
    |    n = 1             |
    |    temp = ?          |
    |    return addr       |
    +----------------------+
    |  [factorial(2) AR]   |
    |    n = 2             |
    |    temp = ?          |
    +----------------------+
    |  ...                 |
    +----------------------+
    |  [factorial(5) AR]   |
    |    n = 5             |
    |    temp = ?          |
    +----------------------+
    |  [main AR]           |
    |    local_a = 10      |
    |    local_b = 20      |
    |    heap_array = ptr  |
    |    result = ?        |
    +----------------------+
    |                      |
    |  (free space)        |
    |                      |
    +----------------------+
    |  Heap (grows up)     |
    |  [heap_array[100]]   | <- 100 integers
    +----------------------+
    |  BSS                 |
    |  (uninitialized)     |
    +----------------------+
    |  Data (initialized)  |
    |    global_var = 100  |
    |    static_var = 200  |
    |    counter = ?       |
    +----------------------+
    |  Text/Code           |
    |  (instructions)      |
Low Address
```

## 13.8 Runtime Proyek Subset C

Untuk compiler proyek subset C (Bab 1–12), asumsi runtime disederhanakan: program berupa barisan statement (tanpa fungsi tambahan di fase awal), sehingga tidak ada pemanggilan fungsi pengguna. Variabel `int`/`float` dialokasikan secara statis atau pada stack dengan layout tetap; code generator (Bab 14) memakai offset/alamat dari

symbol table. Jika kelak proyek diperluas dengan fungsi, calling convention dapat mengikuti konvensi cdecl (stack-based arguments, caller-saved/callee-saved registers) agar konsisten dengan runtime environment yang dibahas di bab ini. Rincian layout activation record dan rencana calling convention untuk proyek dicatat dalam dokumentasi folder `proyek-compiler-subset-c/` agar Bab 14 memiliki acuan saat menghasilkan kode.

## 13.9 Kesimpulan

Dalam bab ini, kita telah mempelajari:

1. Runtime environment adalah konteks eksekusi program yang mencakup memory organization, calling conventions, dan memory management
2. Memory layout terdiri dari code segment, static/global data, stack, dan heap, masing-masing dengan karakteristik dan tujuan penggunaan yang berbeda
3. Activation records (stack frames) menyimpan informasi tentang eksekusi fungsi, termasuk parameters, local variables, return address, dan links
4. Stack-based allocation cocok untuk local variables dengan automatic management, sementara heap allocation diperlukan untuk dynamic data dengan flexible lifetime
5. Garbage collection adalah teknik automatic memory management yang membebaskan unreachable objects, dengan berbagai strategi seperti mark-and-sweep, reference counting, dan generational GC

Runtime proyek subset C (layout variabel, rencana calling convention bila diperluas) memberi acuan untuk code generation di Bab 14.

Pemahaman tentang runtime environment dan memory management sangat penting untuk:

- Merancang compiler yang efisien
- Memahami bagaimana program dieksekusi
- Mengoptimalkan penggunaan memory
- Mengimplementasikan fitur bahasa seperti recursion, closures, dan dynamic allocation

## 13.10 Referensi dan Bahan Bacaan Lanjutan

Untuk memperdalam pemahaman tentang runtime environment dan memory management, mahasiswa disarankan membaca:

- **Dragon Book**: Aho, Lam, Sethi, & Ullman (2006). *Compilers: Principles, Techniques, and Tools* [1] - Bab 7: Run-Time Environments
- **Engineering a Compiler**: Cooper & Torczon (2011) [5] - Bab 6: The Procedure Abstraction
- **StudyLib - Outcomes-Based Education**: Materials tentang runtime environment dan activation records [14]
- **UC San Diego CSE 231**: Course materials tentang compiler construction dan runtime organization [3]
- **Northeastern University CS 4410**: Comprehensive compiler design course dengan coverage runtime issues [4]

# Bab 14

## Code Generation untuk Target Architecture

### 14.1 Tujuan Pembelajaran

Setelah mempelajari bab ini, mahasiswa diharapkan mampu:

1. Memahami proses code generation dari intermediate representation ke target code
2. Menjelaskan konsep instruction selection dan implementasinya
3. Mengimplementasikan register allocation sederhana (local allocation)
4. Membuat code generator untuk operasi aritmatika dan assignment
5. Memahami dan mengimplementasikan calling convention untuk function calls
6. Menghasilkan assembly code yang valid untuk target architecture (x86 atau RISC-V)

### 14.2 Pendahuluan

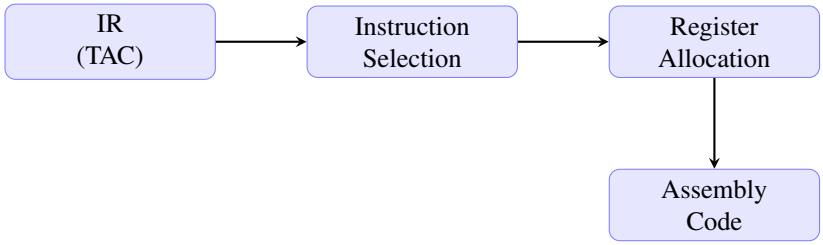
Dalam proyek compiler subset C, code generator mengambil IR dari Bab 12 (TAC/quadruples dari AST proyek) dan asumsi runtime dari Bab 13, lalu menghasilkan assembly untuk target yang dipilih (mis. x86 atau RISC-V subset). Symbol table proyek (Bab 10) menyediakan alamat/offset variabel. Output code generation proyek menjadi masukan assembler/linker dalam pipeline lengkap (Bab 16).

Code generation adalah fase terakhir dalam back-end kompilator yang bertanggung jawab untuk menghasilkan target code dari intermediate representation (IR) yang telah dioptimasi. Menurut sumber dari StudyLib:

> "Code generation: instruction selection, machine model. Implement code generation for a target architecture, mapping intermediate code into efficient target code, managing run-time structures."[14]

Code generator mengambil IR (biasanya dalam bentuk three-address code atau format serupa) dan menghasilkan kode assembly atau machine code yang dapat dieksekusi pada target architecture tertentu.

Gambar 14.1 menunjukkan proses code generation.




Gambar 14.1: Proses code generation

### 14.2.1 Tugas Code Generator

Code generator memiliki beberapa tugas utama:

1. **Instruction Selection**: Memilih instruksi machine yang tepat untuk setiap operasi IR
2. **Register Allocation**: Mengalokasikan register untuk variabel dan temporary values
3. **Instruction Scheduling**: Mengatur urutan instruksi untuk optimasi pipeline
4. **Address Assignment**: Mengalokasikan memory untuk variabel dan data structures
5. **Code Emission**: Menghasilkan assembly atau machine code dalam format yang sesuai

### 14.2.2 Input dan Output Code Generator

**Input Code Generator:**

- Optimized Intermediate Representation (TAC, quadruples, atau format IR lainnya)
- Symbol table dengan informasi tipe dan alamat variabel
- Target architecture specification (instruction set, register set, addressing modes)

**Output Code Generator:**

- Assembly code (untuk assembler) atau machine code langsung
- Relocation information (jika diperlukan)
- Debug information (optional)

## 14.3 Target Architecture

Target architecture adalah platform hardware yang menjadi tujuan kompilasi. Setiap architecture memiliki karakteristik yang berbeda yang mempengaruhi bagaimana code generator bekerja.

### 14.3.1 Karakteristik Target Architecture

Menurut dokumentasi LLVM[1], target architecture memiliki beberapa karakteristik penting:

1. **Instruction Set Architecture (ISA)**: Kumpulan instruksi yang didukung oleh processor
    - RISC (Reduced Instruction Set Computer): Instruksi sederhana, uniform, banyak register
    - CISC (Complex Instruction Set Computer): Instruksi kompleks, berbagai format, addressing modes yang kaya
2. **Register Set**: Jumlah dan jenis register yang tersedia
    - General-purpose registers
    - Special-purpose registers (stack pointer, frame pointer, dll.)
    - Floating-point registers
3. **Addressing Modes**: Cara mengakses operan (register, memory, immediate)
    - Register addressing: `ADD R1, R2`
    - Immediate addressing: `ADD R1, #42`
    - Memory addressing: `LOAD R1, [address]`
    - Indexed addressing: `LOAD R1, [R2 + offset]`
4. **Memory Model**: Bagaimana memory diorganisir dan diakses
    - Byte-addressable vs word-addressable
    - Alignment requirements
    - Endianness (little-endian vs big-endian)

[1] https://llvm.org/docs/CodeGenerator.html

### 14.3.2 Contoh Target Architecture

Dalam pembelajaran ini, kita akan fokus pada dua target architecture populer:

**1. x86-64 (AMD64)**

- CISC architecture dengan instruksi kompleks
- 16 general-purpose registers (RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, R8-R15)
- Berbagai addressing modes
- Variable-length instructions
- Little-endian

**2. RISC-V**

- RISC architecture dengan instruksi sederhana
- 32 general-purpose registers (x0-x31)
- Fixed-length instructions (32-bit atau 16-bit untuk compressed)
- Simple addressing modes
- Little-endian atau big-endian (configurable)
- Open standard, populer untuk pembelajaran

Untuk pembelajaran, kita akan menggunakan subset sederhana dari RISC-V karena lebih mudah dipahami dan diimplementasikan.

## 14.4 Instruction Selection

Instruction selection adalah proses memilih instruksi machine yang tepat untuk mengimplementasikan setiap operasi dalam IR. Tujuannya adalah menghasilkan kode yang efisien dalam hal:

- Execution time (runtime performance)
- Code size
- Energy consumption

### 14.4.1 Metode Instruction Selection

Menurut Wikipedia[2], terdapat beberapa metode instruction selection:

[2] https://en.wikipedia.org/wiki/Instruction_selection

**1. Simple Translation**

Metode paling sederhana: setiap operasi IR langsung dipetakan ke satu atau beberapa instruksi machine.

Contoh: TAC `t1 = t2 + t3` untuk RISC-V:

```
ADD t1, t2, t3
```

Kelebihan: Implementasi mudah, cepat Kekurangan: Tidak selalu optimal, tidak memanfaatkan instruksi kompleks

**2. Tree Pattern Matching**

Menggunakan expression tree dan mencocokkan subtree dengan pattern instruksi machine.

Contoh: Ekspresi `a + b * c` dapat dicocokkan dengan pattern:

- Pattern 1: MUL + ADD (dua instruksi terpisah)
- Pattern 2: FMA (fused multiply-add, satu instruksi jika tersedia)

**3. Dynamic Programming**

Menggunakan dynamic programming untuk menemukan sequence instruksi yang optimal dengan mempertimbangkan cost.

### 14.4.2 Contoh Instruction Selection

Mari kita lihat contoh instruction selection untuk operasi sederhana pada RISC-V:

**Contoh 1: Assignment**

```
TAC: x = y
RISC-V: MV x, y    (atau ADD x, y, x0 untuk register zero)
```

**Contoh 2: Arithmetic Operations**

```
TAC: t1 = a + b
RISC-V: ADD t1, a, b

TAC: t2 = c * d
RISC-V: MUL t2, c, d

TAC: t3 = t1 - t2
RISC-V: SUB t3, t1, t2
```

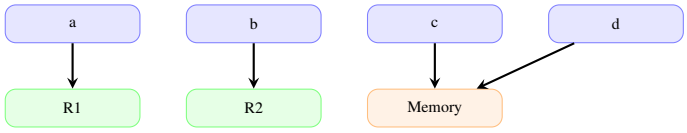
**Contoh 3: Load/Store**

```
TAC: x = mem[addr]
RISC-V: LW x, 0(addr)     (load word)

TAC: mem[addr] = x
RISC-V: SW x, 0(addr)     (store word)
```

**Contoh 4: Constant Loading**

```
TAC: x = 42
RISC-V: LI x, 42          (load immediate, pseudo-instruction)
       atau
       ADDI x, x0, 42    (add immediate dengan zero register)
```

Gambar 14.2 menunjukkan konsep register allocation.




Gambar 14.2: Register allocation: variabel dialokasikan ke register atau memory

## 14.5 Register Allocation

Register allocation adalah proses menentukan variabel dan temporary values mana yang disimpan di register (fast storage) dan mana yang di-spill ke memory. Karena jumlah register fisik terbatas, ini adalah optimasi yang constrained.

### 14.5.1 Mengapa Register Allocation Penting?

Menurut Wikipedia[3]:

- Register jauh lebih cepat daripada memory access
- Jumlah register fisik terbatas (biasanya 16-32 register)
- Program mungkin memiliki lebih banyak variabel aktif daripada jumlah register
- Register allocation yang baik dapat meningkatkan performa secara signifikan

[3] https://en.wikipedia.org/wiki/Register_allocation

### 14.5.2 Dua Fase Register Allocation

#### 1. Allocation Phase

Memutuskan *mana* nilai yang harus disimpan di register pada setiap program point. Ini melibatkan:

- Live range analysis: Kapan variabel hidup (live) dan kapan mati (dead)
- Interference graph: Grafik yang menunjukkan variabel mana yang tidak bisa menggunakan register yang sama secara bersamaan

#### 2. Assignment Phase

Memetakan nilai yang dialokasikan ke register fisik spesifik. Jika lebih banyak nilai yang perlu register daripada register yang tersedia, beberapa nilai harus di-spill ke memory.

### 14.5.3 Local Register Allocation

Local register allocation bekerja dalam satu basic block (satu entry, satu exit, tidak ada branching). Ini lebih sederhana karena control flow linear.

#### Algoritma Simple Local Allocation

Algoritma sederhana untuk local allocation:

1. Scan basic block dari awal hingga akhir
2. Untuk setiap instruksi:
    - Jika operan tidak di register, load dari memory
    - Eksekusi operasi menggunakan register
    - Jika hasil perlu disimpan dan register penuh, spill register yang paling lama tidak digunakan
3. Di akhir block, store semua register yang modified ke memory

#### Contoh Local Allocation

Misalkan kita memiliki basic block dengan TAC berikut:

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

Dengan 3 register tersedia (R1, R2, R3), allocation bisa seperti ini:

```
LOAD R1, a          ; Load a ke R1
LOAD R2, b          ; Load b ke R2
ADD R1, R1, R2      ; R1 = a + b (t1)
LOAD R2, c          ; Load c ke R2 (b tidak lagi diperlukan)
MUL R1, R1, R2      ; R1 = t1 * c (t2)
STORE R1, d         ; Store hasil ke d
```

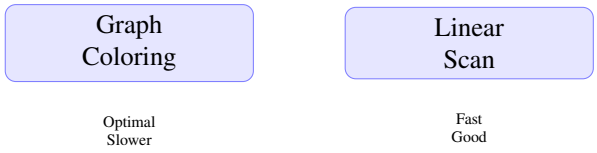
### 14.5.4 Global Register Allocation

Global register allocation bekerja lintas basic blocks atau seluruh function. Ini lebih kompleks karena perlu mempertimbangkan control flow.

Metode populer:

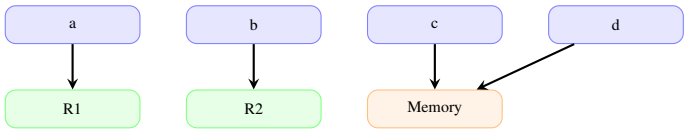
- **Graph Coloring**: Membangun interference graph dan mewarnainya dengan k warna (k = jumlah register)
- **Linear Scan**: Lebih cepat, menghitung live intervals dan assign register dalam satu pass

Gambar 14.3 menunjukkan perbandingan metode register allocation.




Gambar 14.3: Perbandingan metode register allocation

Gambar 14.4 menunjukkan konsep register allocation.




Gambar 14.4: Register allocation: variabel dialokasikan ke register atau memory

Untuk pembelajaran, kita akan fokus pada local allocation yang lebih sederhana.

## 14.6 Code Generation untuk Operasi Aritmatika

Mari kita implementasikan code generator untuk operasi aritmatika dasar. Kita akan menggunakan RISC-V sebagai target architecture.

### 14.6.1 Struktur Code Generator Sederhana

Code generator sederhana dapat diimplementasikan sebagai visitor pattern yang traverse AST atau TAC dan menghasilkan assembly code.

#### Contoh Implementasi dalam C++

Berikut adalah struktur dasar code generator:

Listing 14.1: Struktur dasar Code Generator

```
class CodeGenerator {
private:
    std::vector<std::string> assemblyCode;
    int tempCounter;
    std::map<std::string, std::string> varToReg;
    std::set<std::string> availableRegs;

public:
    CodeGenerator() : tempCounter(0) {
        // Inisialisasi register yang tersedia
        for (int i = 1; i <= 8; i++) {
            availableRegs.insert("t" + std::to_string(i));
        }
    }

    std::string getRegister(const std::string& var) {
        // Alokasi register untuk variabel
        if (varToReg.find(var) != varToReg.end()) {
            return varToReg[var];
        }

        if (!availableRegs.empty()) {
            std::string reg = *availableRegs.begin();
            availableRegs.erase(reg);
            varToReg[var] = reg;
            return reg;
        }

        // Spill ke memory jika register penuh
        return spillRegister(var);
    }

    void generateAdd(const std::string& result,
                     const std::string& op1,
                     const std::string& op2) {
        std::string reg1 = getRegister(op1);
        std::string reg2 = getRegister(op2);
        std::string regResult = getRegister(result);

        assemblyCode.push_back("ADD " + regResult + ", " +
                               reg1 + ", " + reg2);
    }

```

```
    void generateMul(const std::string& result,
                     const std::string& op1,
                     const std::string& op2) {
        std::string reg1 = getRegister(op1);
        std::string reg2 = getRegister(op2);
        std::string regResult = getRegister(result);

        assemblyCode.push_back("MUL " + regResult + ", " +
                               reg1 + ", " + reg2);
    }

    void generateLoad(const std::string& reg,
                      const std::string& var) {
        assemblyCode.push_back("LW " + reg + ", " + var);
    }

    void generateStore(const std::string& var,
                       const std::string& reg) {
        assemblyCode.push_back("SW " + reg + ", " + var);
    }

    std::vector<std::string> getAssembly() {
        return assemblyCode;
    }
};
```

### 14.6.2 Generating Code untuk Ekspresi Kompleks

Untuk ekspresi kompleks seperti `a + b * c`, kita perlu mempertimbangkan precedence:

```
TAC untuk: result = a + b * c

t1 = b * c
t2 = a + t1
result = t2

Generated RISC-V code:
LW t1, a          ; Load a
LW t2, b          ; Load b
LW t3, c          ; Load c
MUL t4, t2, t3    ; t4 = b * c
ADD t5, t1, t4    ; t5 = a + t4
SW t5, result     ; Store result
```

## 14.7 Handling Function Calls dan Calling Convention

Function calls memerlukan koordinasi antara caller dan callee untuk:

- Passing arguments

- Saving/restoring registers
- Managing stack frame
- Returning values

Calling convention mendefinisikan aturan ini.

### 14.7.1 RISC-V Calling Convention

RISC-V memiliki calling convention yang didefinisikan dalam ABI (Application Binary Interface):

#### Register Usage

- **Argument Registers**: a0-a7 (x10-x17) untuk 8 argument pertama
- **Return Register**: a0 (x10) untuk return value
- **Caller-saved Registers**: t0-t6 (x5-x7, x28-x31) - caller harus save
- **Callee-saved Registers**: s0-s11 (x8-x9, x18-x27) - callee harus save
- **Stack Pointer**: sp (x2)
- **Frame Pointer**: fp/s0 (x8)

#### Function Call Sequence

**Caller Side:**

```
# Save caller-saved registers jika diperlukan
# Pass arguments ke a0-a7
# Call function
JAL ra, function_name
# Return value di a0
# Restore caller-saved registers
```

**Callee Side (Function Prologue):**

```
function_name:
    # Save return address dan frame pointer
    ADDI sp, sp, -frame_size
    SW ra, frame_size-4(sp)
    SW fp, frame_size-8(sp)
    ADDI fp, sp, frame_size

    # Save callee-saved registers yang digunakan
    # Allocate space untuk local variables
```

**Callee Side (Function Epilogue):**

```
    # Restore callee-saved registers
    # Put return value di a0
    # Restore frame pointer dan return address
    LW fp, frame_size-8(sp)
    LW ra, frame_size-4(sp)
    ADDI sp, sp, frame_size
    RET  # atau JALR x0, 0(ra)
```

### 14.7.2 Contoh Function Call

Mari kita lihat contoh function call untuk `int add(int a, int b)`:

**Caller Code:**

```
# Prepare arguments
LI a0, 10        # First argument (a = 10)
LI a1, 20        # Second argument (b = 20)

# Call function
JAL ra, add

# Result is in a0
# Use result...
```

**Callee Code (add function):**

```
add:
    # Function prologue
    ADDI sp, sp, -16    # Allocate stack frame
    SW ra, 12(sp)       # Save return address
    SW fp, 8(sp)        # Save frame pointer
    ADDI fp, sp, 16     # Set frame pointer

    # Function body
    ADD a0, a0, a1      # a0 = a0 + a1 (result)

    # Function epilogue
    LW fp, 8(sp)        # Restore frame pointer
    LW ra, 12(sp)       # Restore return address
    ADDI sp, sp, 16     # Deallocate stack frame
    RET                 # Return
```

## 14.8 Implementasi Code Generator Lengkap

Mari kita buat implementasi code generator yang lebih lengkap yang dapat menangani berbagai operasi.

### 14.8.1 Code Generator untuk TAC

Berikut adalah contoh code generator yang mengambil TAC dan menghasilkan RISC-V assembly:

Listing 14.2: Code Generator untuk TAC ke RISC-V

```
class TACCodeGenerator {
private:
    std::vector<std::string> assembly;
    int labelCounter;
    std::map<std::string, int> varOffset;  // Offset di stack frame
    int stackOffset;

public:
    TACCodeGenerator() : labelCounter(0), stackOffset(0) {}

    void generateTAC(const TACInstruction& tac) {
        switch (tac.op) {
            case TAC_OP::ADD:
                generateAdd(tac.result, tac.arg1, tac.arg2);
                break;
            case TAC_OP::SUB:
                generateSub(tac.result, tac.arg1, tac.arg2);
                break;
            case TAC_OP::MUL:
                generateMul(tac.result, tac.arg1, tac.arg2);
                break;
            case TAC_OP::DIV:
                generateDiv(tac.result, tac.arg1, tac.arg2);
                break;
            case TAC_OP::ASSIGN:
                generateAssign(tac.result, tac.arg1);
                break;
            case TAC_OP::LOAD:
                generateLoad(tac.result, tac.arg1);
                break;
            case TAC_OP::STORE:
                generateStore(tac.arg1, tac.result);
                break;
            // ... operasi lainnya
        }
    }

    void generateAdd(const std::string& result,
                     const std::string& arg1,
                     const std::string& arg2) {
        std::string reg1 = loadToRegister(arg1);
        std::string reg2 = loadToRegister(arg2);
        std::string regResult = allocateRegister(result);

        assembly.push_back("ADD " + regResult + ", " +
                           reg1 + ", " + reg2);

        releaseRegister(reg1);
```

```
49          releaseRegister(reg2);
50      }
51
52      // ... implementasi operasi lainnya
53  };
```

## 14.9 Testing dan Validasi

Setelah code generator menghasilkan assembly, kita perlu:

1. **Assemble**: Mengubah assembly menjadi object file
2. **Link**: Menyatukan object files menjadi executable
3. **Run**: Mengeksekusi program dan memverifikasi hasilnya

### 14.9.1 Workflow Lengkap

```
Source Code (C/C++)
    ↓
[Compiler Front-end]
    ↓
TAC / IR
    ↓
[Code Generator] → Assembly Code (.s)
    ↓
[Assembler] → Object File (.o)
    ↓
[Linker] → Executable
    ↓
[Run] → Verify Output
```

### 14.9.2 Contoh Testing

Misalkan kita memiliki program sederhana:

```
int main() {
    int a = 10;
    int b = 20;
    int c = a + b;
    return c;
}
```

TAC yang dihasilkan:

```
t1 = 10
a = t1
```

```
t2 = 20
b = t2
t3 = a + b
c = t3
return c
```

Assembly yang dihasilkan (RISC-V):

```
main:
    ADDI sp, sp, -16
    SW ra, 12(sp)
    SW fp, 8(sp)
    ADDI fp, sp, 16

    # a = 10
    LI t0, 10
    SW t0, -4(fp)    # Store a di stack

    # b = 20
    LI t0, 20
    SW t0, -8(fp)    # Store b di stack

    # c = a + b
    LW t1, -4(fp)    # Load a
    LW t2, -8(fp)    # Load b
    ADD t0, t1, t2
    SW t0, -12(fp)   # Store c

    # return c
    LW a0, -12(fp)   # Return value

    LW fp, 8(sp)
    LW ra, 12(sp)
    ADDI sp, sp, 16
    RET
```

## 14.10 Kesimpulan

Dalam bab ini, kita telah mempelajari:

1. Code generation adalah fase terakhir yang mengubah IR menjadi target code
2. Instruction selection memilih instruksi machine yang tepat untuk setiap operasi IR
3. Register allocation menentukan variabel mana yang disimpan di register vs memory
4. Local register allocation lebih sederhana dan cocok untuk pembelajaran awal

5. Calling convention mengatur bagaimana function calls dilakukan
6. Code generator harus menghasilkan assembly yang valid dan dapat di-assemble, link, dan run

Implementasi code generator yang baik memerlukan pemahaman mendalam tentang target architecture dan trade-off antara code size, execution time, dan register pressure. Code generation proyek subset C memakai IR (Bab 12) dan runtime proyek (Bab 13); hasilnya terintegrasi dalam compiler lengkap yang dipresentasikan di Bab 16.

## 14.11 Referensi dan Bahan Bacaan Lanjutan

Untuk memperdalam pemahaman tentang code generation, mahasiswa disarankan membaca:

- **Dragon Book**: Aho, Lam, Sethi, & Ullman (2006). *Compilers: Principles, Techniques, and Tools* [1] - Bab 8: Code Generation
- **Engineering a Compiler**: Cooper & Torczon (2011) [5] - Bab 7-9: Code Shape, Introduction to Optimization, Scalar Optimizations
- **RISC-V Instruction Set Manual**: `https://riscv.org/technical/specifications/` - Dokumentasi lengkap instruksi RISC-V
- **LLVM Code Generator Documentation**: `https://llvm.org/docs/CodeGenerator.html` - Dokumentasi code generator LLVM
- **StudyLib - Outcomes-Based Education**: Materials tentang code generation dan runtime structures [14]
- **Wikipedia - Register Allocation**: `https://en.wikipedia.org/wiki/Register_allocation` - Artikel tentang teknik register allocation
- **Wikipedia - Instruction Selection**: `https://en.wikipedia.org/wiki/Instruction_selection` - Artikel tentang instruction selection

# Bab 15

## Optimasi Kompilator Dasar

### 15.1 Tujuan Pembelajaran

Setelah mempelajari bab ini, mahasiswa diharapkan mampu:

1. Memahami konsep optimasi kompilator dan tujuannya
2. Menjelaskan dan mengidentifikasi basic blocks dalam intermediate code
3. Mengimplementasikan optimasi lokal: constant folding dan constant propagation
4. Mengimplementasikan dead code elimination
5. Memahami dasar-dasar data-flow analysis untuk optimasi global
6. Mengevaluasi efektivitas optimasi dengan membandingkan before/after
7. Membedakan machine-independent dan machine-specific optimizations

### 15.2 Pendahuluan

Dalam proyek compiler subset C, optimasi diterapkan pada IR (Bab 12) atau pada output code generation (Bab 14): basic block, constant folding, constant propagation, dead code elimination. Konteks "compiler subset C kita" memastikan bahwa optimasi konsisten dengan AST, symbol table, dan IR proyek.

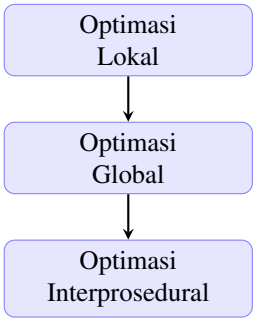
Optimasi kompilator adalah proses transformasi kode intermediate untuk meningkatkan kualitas kode yang dihasilkan tanpa mengubah semantik program. Menurut sumber dari Scribd OBE CSE Document:

> "Perform machine-independent optimizations (basic block optimizations, data-flow analysis). Local and global optimization; data-flow analysis."[1]

Tujuan optimasi kompilator meliputi:

- **Meningkatkan Performa**: Mengurangi waktu eksekusi program
- **Mengurangi Ukuran Kode**: Menghasilkan executable yang lebih kecil
- **Mengurangi Konsumsi Memory**: Mengoptimasi penggunaan memory
- **Meningkatkan Efisiensi Energy**: Mengurangi konsumsi daya (penting untuk embedded systems)

Gambar 15.1 menunjukkan level-level optimasi.




Gambar 15.1: Level-level optimasi kompilator

Namun, optimasi harus dilakukan dengan hati-hati karena:

- Optimasi yang terlalu agresif dapat meningkatkan waktu kompilasi
- Beberapa optimasi dapat membuat kode lebih sulit di-debug
- Optimasi yang salah dapat mengubah semantik program (bug)

### 15.2.1 Prinsip Optimasi

Menurut Dragon Book[1], optimasi harus mematuhi prinsip-prinsip berikut:

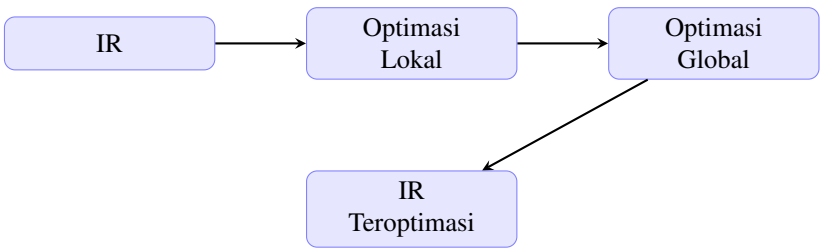
1. **Correctness**: Optimasi tidak boleh mengubah semantik program
2. **Benefit**: Optimasi harus memberikan manfaat yang signifikan
3. **Speed**: Proses optimasi tidak boleh terlalu lambat
4. **Simplicity**: Optimasi harus mudah diimplementasikan dan di-maintain

### 15.2.2 Level Optimasi

Optimasi dapat dikategorikan berdasarkan scope-nya:

- **Optimasi Lokal (Local Optimization)**: Optimasi dalam satu basic block
  - Constant folding
  - Constant propagation
  - Algebraic simplification
  - Strength reduction
- **Optimasi Global (Global Optimization)**: Optimasi lintas basic blocks
  - Common subexpression elimination
  - Loop optimization
  - Dead code elimination (global)
  - Constant propagation (global)
- **Optimasi Interprosedural (Interprocedural Optimization)**: Optimasi lintas fungsi/prosedur
  - Inlining
  - Interprocedural constant propagation
  - Whole-program optimization

Gambar 15.2 menunjukkan pipeline optimasi dalam kompilator.




Gambar 15.2: Pipeline optimasi kompilator

## 15.3 Basic Blocks

Basic block adalah fondasi untuk banyak optimasi kompilator. Menurut University of Michigan[1], basic block didefinisikan sebagai:

[1] https://web.eecs.umich.edu/~weimerw/2015-4610/ca1/ca1.html

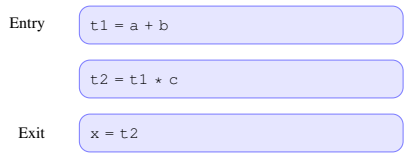
> "A basic block is a straight-line sequence of code with no jumps in except at the entry, and no jumps out except at the exit. Once execution enters it, all instructions execute sequentially."

### 15.3.1 Karakteristik Basic Block

Sebuah basic block memiliki karakteristik berikut:

1. **Single Entry Point**: Hanya ada satu titik masuk (entry point)
2. **Single Exit Point**: Hanya ada satu titik keluar (exit point)
3. **Sequential Execution**: Semua instruksi dieksekusi secara berurutan tanpa branching
4. **No Internal Control Flow**: Tidak ada jump, branch, atau call di tengah-tengah block

Gambar 15.3 menunjukkan contoh basic block.




Gambar 15.3: Contoh basic block

### 15.3.2 Identifikasi Basic Blocks

Algoritma untuk mengidentifikasi basic blocks dalam intermediate code:

1. **Leader Identification**: Tentukan leader (instruksi pertama dalam basic block)
   - Instruksi pertama dalam program adalah leader
   - Instruksi yang merupakan target dari jump/branch adalah leader
   - Instruksi setelah jump/branch/call adalah leader
2. **Block Construction**: Untuk setiap leader, buat basic block yang berisi:
   - Leader instruction
   - Semua instruksi berikutnya hingga menemukan leader berikutnya atau instruksi control flow

### 15.3.3 Contoh Identifikasi Basic Block

Perhatikan contoh three-address code berikut:

```
L1: t1 = a + b
    t2 = c * d
    t3 = t1 + t2
    if t3 > 0 goto L2
    t4 = t1 - t2
    goto L3
L2: t5 = t1 * t2
L3: t6 = t5 + 1
    return t6
```

Basic blocks yang diidentifikasi:
**Block 1 (L1):**

```
t1 = a + b
t2 = c * d
t3 = t1 + t2
if t3 > 0 goto L2
```

**Block 2 (L2):**

```
t5 = t1 * t2
```

**Block 3 (setelah goto L3):**

```
t4 = t1 - t2
goto L3
```

**Block 4 (L3):**

```
t6 = t5 + 1
return t6
```

### 15.3.4 Control Flow Graph (CFG)

Control Flow Graph adalah representasi grafis dari alur kontrol program. Menurut Wikipedia[2]:

> "A control-flow graph (CFG) is a representation of a function where each node is a basic block, and edges represent possible flow of control from one block to another."

CFG membantu dalam:

- Memahami struktur program

[2] https://en.wikipedia.org/wiki/Control-flow_graph

- Melakukan data-flow analysis
- Mengidentifikasi loop dan struktur kontrol lainnya
- Mengoptimasi lintas basic blocks

## 15.4 Constant Folding

Constant folding adalah optimasi yang mengganti ekspresi yang hanya melibatkan konstanta dengan hasil komputasinya pada waktu kompilasi. Menurut GeeksforGeeks[3]:

> "Constant folding replaces expressions involving only constants (literals) with their computed result at compile time, rather than at runtime. Example: turning '5 + 7 * 2' into '19' in the generated code."

### 15.4.1 Contoh Constant Folding

**Before optimization:**

```
t1 = 5 + 7
t2 = t1 * 2
t3 = 10 / 2
x = t2 + t3
```

**After constant folding:**

```
t1 = 12         // 5 + 7 = 12
t2 = 24         // 12 * 2 = 24
t3 = 5          // 10 / 2 = 5
x = 29          // 24 + 5 = 29
```

Atau bahkan lebih optimal:

```
x = 29          // Semua konstanta di-fold menjadi satu nilai
```

### 15.4.2 Implementasi Constant Folding

Algoritma constant folding untuk three-address code:

1. Untuk setiap instruksi dalam basic block:
   - Jika kedua operan adalah konstanta, evaluasi ekspresi
   - Ganti instruksi dengan assignment konstanta hasil
2. Ulangi hingga tidak ada lagi perubahan (iterasi mungkin diperlukan jika ada dependensi)

[3] https://www.geeksforgeeks.org/compiler-design/constant-folding/

### 15.4.3 Contoh Implementasi dalam C++

Berikut adalah contoh sederhana implementasi constant folding:

Listing 15.1: Contoh implementasi constant folding

```cpp
struct Instruction {
    string op;      // operator: +, -, *, /, =
    string result;  // variabel hasil
    string arg1;    // operand pertama
    string arg2;    // operand kedua (optional)
};

bool isConstant(const string& var,
                const map<string, int>& constants) {
    return constants.find(var) != constants.end();
}

int evaluateConstant(int val1, int val2, const string& op) {
    if (op == "+") return val1 + val2;
    if (op == "-") return val1 - val2;
    if (op == "*") return val1 * val2;
    if (op == "/") return val2 != 0 ? val1 / val2 : 0;
    return 0;
}

void constantFolding(vector<Instruction>& instructions) {
    map<string, int> constants;

    for (auto& inst : instructions) {
        if (inst.op == "=" && isNumeric(inst.arg1)) {
            // Assignment konstanta langsung
            constants[inst.result] = stoi(inst.arg1);
        } else if (inst.op != "=") {
            // Operasi biner
            if (isConstant(inst.arg1, constants) &&
                isConstant(inst.arg2, constants)) {
                int val1 = constants[inst.arg1];
                int val2 = constants[inst.arg2];
                int result = evaluateConstant(val1, val2, inst.op);
                constants[inst.result] = result;
                // Ganti instruksi dengan assignment konstanta
                inst.op = "=";
                inst.arg1 = to_string(result);
                inst.arg2 = "";
            }
        }
    }
}
```

## 15.5 Constant Propagation

Constant propagation adalah optimasi yang mengganti penggunaan variabel yang diketahui bernilai konstan dengan nilai konstanta tersebut. Menurut GeeksforGeeks[4]:

> "Constant propagation replaces variables known to be constant with their constant values, then combined with constant folding to simplify more complex expressions."

### 15.5.1 Contoh Constant Propagation

**Before optimization:**

```
x = 10
y = 20
t1 = x + 5      // x adalah konstanta 10
t2 = y * 2      // y adalah konstanta 20
z = t1 + t2
```

**After constant propagation:**

```
x = 10
y = 20
t1 = 10 + 5     // x diganti dengan 10
t2 = 20 * 2     // y diganti dengan 20
z = t1 + t2
```

**After constant folding (kombinasi):**

```
x = 10
y = 20
t1 = 15         // 10 + 5 = 15
t2 = 40         // 20 * 2 = 40
z = 55          // 15 + 40 = 55
```

### 15.5.2 Local vs Global Constant Propagation

**Local Constant Propagation:**

- Hanya dalam satu basic block
- Lebih sederhana, tidak memerlukan data-flow analysis
- Dapat dilakukan bersamaan dengan constant folding

**Global Constant Propagation:**

[4] https://www.geeksforgeeks.org/machine-independent-code-optimization-in-compiler-design/

- Lintas basic blocks
- Memerlukan data-flow analysis (reaching definitions)
- Lebih kompleks tetapi lebih powerful
- Harus mempertimbangkan multiple paths dalam CFG

### 15.5.3 Implementasi Local Constant Propagation

Algoritma untuk local constant propagation:

1. Scan basic block dari atas ke bawah
2. Maintain map variabel $\rightarrow$ nilai konstanta
3. Untuk setiap instruksi:
    - Jika assignment konstanta: update map
    - Jika penggunaan variabel: ganti dengan konstanta jika tersedia
    - Jika assignment dari variabel non-konstanta: hapus dari map (variabel tidak lagi konstanta)

## 15.6 Dead Code Elimination

Dead code elimination adalah optimasi yang menghapus kode yang tidak memiliki efek pada perilaku program yang dapat diamati. Menurut GeeksforGeeks[5]:

> "Dead code elimination removes code that has no effect on the program's observable behavior. Two main kinds: unreachable code (code that can never be executed) and assignment to variables never used (where a variable's value is computed but never read before being overwritten)."

### 15.6.1 Jenis Dead Code

**1. Unreachable Code** Kode yang tidak pernah dapat dieksekusi karena tidak ada path yang mencapainya.

Contoh:

```
x = 10
return x
y = 20      // Dead code - tidak pernah dieksekusi
z = y + 5   // Dead code
```

[5] https://www.geeksforgeeks.org/dead-code-elimination/

**2. Dead Assignments** Assignment ke variabel yang nilainya tidak pernah digunakan sebelum di-overwrite.

Contoh:

```
x = 10
x = 20      // Assignment pertama adalah dead code
y = x       // Hanya nilai kedua yang digunakan
```

### 15.6.2 Unreachable Code Elimination

Algoritma untuk menghapus unreachable code:

1. Bangun Control Flow Graph (CFG)
2. Lakukan traversal dari entry block (misalnya DFS atau BFS)
3. Mark semua basic block yang dapat dicapai
4. Hapus semua basic block yang tidak ter-mark

### 15.6.3 Dead Assignment Elimination

Algoritma untuk menghapus dead assignments (menggunakan live-variable analysis):

1. Lakukan live-variable analysis untuk menentukan variabel mana yang "live" di setiap titik
2. Variabel dikatakan "live" jika nilainya mungkin digunakan di masa depan
3. Untuk setiap assignment `x = ...`:
    - Jika `x` tidak live setelah assignment, assignment tersebut adalah dead code
    - Hapus assignment tersebut

### 15.6.4 Contoh Dead Code Elimination

**Before optimization:**

```
x = 10
y = 20
t1 = x + y
t2 = t1 * 2
x = t2
t3 = 5 + 3      // Dead: t3 tidak pernah digunakan
t4 = t3 - 2    // Dead: t4 tidak pernah digunakan
return x
```

**After dead code elimination:**

```
x = 10
y = 20
t1 = x + y
t2 = t1 * 2
x = t2
return x
```

### 15.6.5 Implementasi Dead Code Elimination

Berikut adalah contoh implementasi sederhana untuk dead assignment elimination:

Listing 15.2: Contoh implementasi dead code elimination

```
set<string> computeLiveVariables(const vector<Instruction>& insts) {
    set<string> live;

    // Scan dari bawah ke atas
    for (int i = insts.size() - 1; i >= 0; i--) {
        const auto& inst = insts[i];

        // Variabel yang digunakan adalah live
        if (!inst.arg1.empty() && !isConstant(inst.arg1)) {
            live.insert(inst.arg1);
        }
        if (!inst.arg2.empty() && !isConstant(inst.arg2)) {
            live.insert(inst.arg2);
        }

        // Variabel yang di-assign tidak lagi live setelah assignment
        // (kecuali jika digunakan di sisi kanan)
        if (live.find(inst.result) != live.end()) {
            live.erase(inst.result);
        }
    }

    return live;
}

vector<Instruction> eliminateDeadCode(
    const vector<Instruction>& instructions) {

    set<string> live = computeLiveVariables(instructions);
    vector<Instruction> optimized;

    for (const auto& inst : instructions) {
        // Skip jika assignment ke variabel yang tidak live
        if (inst.op == "=" && live.find(inst.result) == live.end()) {
            continue; // Dead assignment
        }

        optimized.push_back(inst);

```

```
40          // Update live set
41          if (!inst.arg1.empty()) live.insert(inst.arg1);
42          if (!inst.arg2.empty()) live.insert(inst.arg2);
43          live.erase(inst.result);
44      }
45
46      return optimized;
47  }
```

## 15.7 Data-Flow Analysis Dasar

Data-flow analysis adalah teknik untuk menghitung informasi tentang kemungkinan perilaku program. Menurut GeeksforGeeks[6]:

> "Data-flow analysis is a technique to compute information about possible program behaviors (how definitions, uses of variables, expressions, etc. propagate through the code). Usually done on a CFG."

Data-flow analysis adalah fondasi untuk optimasi global yang lebih advanced.

### 15.7.1 Konsep Dasar Data-Flow Analysis

Data-flow analysis bekerja dengan:

1. **Domain**: Himpunan informasi yang ingin dihitung
   - Live variables: set variabel yang live
   - Reaching definitions: set definisi yang mencapai suatu titik
   - Available expressions: set ekspresi yang sudah dihitung
2. **Transfer Function**: Bagaimana informasi berubah setelah eksekusi instruksi
3. **Meet/Join Operation**: Bagaimana menggabungkan informasi dari multiple paths
4. **Fixpoint Iteration**: Iterasi hingga mencapai fixpoint (tidak ada perubahan)

### 15.7.2 Live Variable Analysis

Live variable analysis menentukan variabel mana yang "live" (nilainya mungkin digunakan) di setiap titik program.

**Definisi:**

[6] https://www.geeksforgeeks.org/data-flow-analysis-compiler/

- Variabel `v` dikatakan **live** di titik `p` jika ada path dari `p` ke penggunaan `v` tanpa assignment ke `v` di antara keduanya
- Variabel `v` dikatakan **dead** jika tidak live

**Algoritma (Backward Analysis):**

1. Inisialisasi: `LIVE[exit] = {}`
2. Untuk setiap basic block `B` (dari exit ke entry):
    - `LIVE[B] = UNION(LIVE[successors])`
    - `LIVE[B] = LIVE[B] - DEF[B] + USE[B]`
    - `DEF[B]`: variabel yang didefinisikan di B
    - `USE[B]`: variabel yang digunakan di B
3. Ulangi hingga fixpoint

### 15.7.3 Reaching Definitions

Reaching definitions analysis menentukan definisi variabel mana yang "mencapai" suatu titik program.

**Definisi:** Definisi `d` dikatakan **reach** titik `p` jika ada path dari `d` ke `p` tanpa definisi lain untuk variabel yang sama.

**Kegunaan:**

- Constant propagation (global)
- Deteksi penggunaan variabel sebelum inisialisasi
- Optimasi lainnya

### 15.7.4 Available Expressions

Available expressions analysis menentukan ekspresi mana yang sudah dihitung dan masih valid (operand-nya belum berubah).

**Kegunaan:**

- Common subexpression elimination
- Optimasi lainnya

## 15.8 Kombinasi Optimasi

Dalam praktik, optimasi biasanya dilakukan dalam beberapa pass dan saling berinteraksi:

### 15.8.1 Order of Optimization

Urutan optimasi yang umum:

1. **Constant Folding & Propagation**: Simplifikasi ekspresi konstanta
2. **Dead Code Elimination**: Hapus kode yang tidak digunakan
3. **Common Subexpression Elimination**: Hapus komputasi duplikat
4. **Loop Optimizations**: Optimasi khusus untuk loop
5. **Register Allocation**: Alokasi register yang efisien

### 15.8.2 Iterative Optimization

Optimizer biasanya menjalankan beberapa pass hingga tidak ada lagi perubahan:

```
do {
    changed = false
    changed |= constantFolding()
    changed |= constantPropagation()
    changed |= deadCodeElimination()
    // ... optimasi lainnya
} while (changed)
```

## 15.9 Evaluasi Efektivitas Optimasi

Setelah mengimplementasikan optimasi, penting untuk mengevaluasi efektivitasnya.

### 15.9.1 Metrics untuk Evaluasi

**1. Code Size**

- Ukuran executable sebelum dan sesudah optimasi
- Jumlah instruksi dalam intermediate code
- Ukuran object files

**2. Execution Time**

- Waktu eksekusi program dengan benchmark
- Profiling untuk mengidentifikasi bottleneck
- Perbandingan before/after

**3. Memory Usage**

- Peak memory consumption
- Stack usage
- Heap allocation patterns

**4. Compilation Time**

- Waktu yang dibutuhkan untuk kompilasi
- Trade-off antara waktu kompilasi dan kualitas optimasi

### 15.9.2 Benchmarking

Langkah-langkah untuk benchmarking:

1. **Prepare Test Cases**: Siapkan berbagai test case (small, medium, large programs)
2. **Baseline Measurement**: Ukur metrik sebelum optimasi
3. **Optimized Measurement**: Ukur metrik setelah optimasi
4. **Compare Results**: Bandingkan dan hitung improvement percentage
5. **Verify Correctness**: Pastikan program masih menghasilkan output yang benar

### 15.9.3 Contoh Evaluasi

Berikut adalah contoh format laporan evaluasi:

| Metric | Before | After | Improvement |
|---|---|---|---|
| Code Size (bytes) | 1024 | 768 | 25% reduction |
| Execution Time (ms) | 100 | 75 | 25% faster |
| Instruction Count | 150 | 110 | 26.7% reduction |
| Compilation Time (s) | 2.5 | 3.1 | 24% slower |

Tabel 15.1: Contoh hasil evaluasi optimasi

## 15.10 Implementasi Praktis

Dalam bagian ini, kita akan melihat contoh implementasi optimizer sederhana yang menggabungkan beberapa optimasi dasar.

### 15.10.1 Struktur Optimizer

Listing 15.3: Struktur dasar optimizer

```
class Optimizer {
private:
    vector<BasicBlock> basicBlocks;
    ControlFlowGraph cfg;

public:
    // Identifikasi basic blocks
    void identifyBasicBlocks(const vector<Instruction>& insts);

    // Optimasi lokal dalam basic block
    void optimizeBasicBlock(BasicBlock& block);

    // Optimasi global lintas basic blocks
    void optimizeGlobal();

    // Kombinasi semua optimasi
    vector<Instruction> optimize(const vector<Instruction>& insts);
};

vector<Instruction> Optimizer::optimize(
    const vector<Instruction>& instructions) {

    // 1. Identifikasi basic blocks
    identifyBasicBlocks(instructions);

    // 2. Optimasi lokal untuk setiap basic block
    for (auto& block : basicBlocks) {
        optimizeBasicBlock(block);
    }

    // 3. Optimasi global
    optimizeGlobal();

    // 4. Reconstruct instructions dari basic blocks
    return reconstructInstructions();
}

void Optimizer::optimizeBasicBlock(BasicBlock& block) {
    bool changed = true;

    while (changed) {
        changed = false;

        // Constant folding
        changed |= constantFolding(block);

        // Constant propagation
        changed |= constantPropagation(block);

        // Dead code elimination
        changed |= deadCodeElimination(block);
```

```
52         }
53 }
```

## 15.11 Kesimpulan

Dalam bab ini, kita telah mempelajari:

1. Optimasi kompilator bertujuan meningkatkan kualitas kode tanpa mengubah semantik
2. Basic blocks adalah unit fundamental untuk optimasi lokal
3. Constant folding dan constant propagation adalah optimasi dasar yang efektif
4. Dead code elimination menghapus kode yang tidak berguna
5. Data-flow analysis adalah fondasi untuk optimasi global
6. Evaluasi efektivitas optimasi penting untuk memastikan optimasi memberikan manfaat

Optimasi kompilator adalah bidang yang luas dan kompleks. Bab ini memberikan dasar-dasar optimasi lokal. Untuk optimasi yang lebih advanced seperti loop optimization, interprocedural optimization, dan machine-specific optimization, diperlukan pemahaman yang lebih mendalam tentang data-flow analysis dan teknik optimasi lainnya. Optimasi proyek subset C diterapkan pada IR dan/atau kode yang dihasilkan; compiler lengkap (lexer, parser, AST, symbol table, type check, IR, codegen, optimasi) dipresentasikan di Bab 16.

## 15.12 Referensi dan Bahan Bacaan Lanjutan

Untuk memperdalam pemahaman tentang optimasi kompilator, mahasiswa disarankan membaca:

- **Dragon Book**: Aho, Lam, Sethi, & Ullman (2006). *Compilers: Principles, Techniques, and Tools* [1] - Bab 9: Machine-Independent Optimizations
- **Engineering a Compiler**: Cooper & Torczon (2011) [5] - Bab 8: Introduction to Optimization, Bab 9: Data-Flow Analysis
- **GeeksforGeeks**: Tutorial tentang berbagai optimasi kompilator
  - Constant Folding: https://www.geeksforgeeks.org/compiler-design/constant-folding/
  - Dead Code Elimination: https://www.geeksforgeeks.org/dead-code-elimination/

  - Data-Flow Analysis: https://www.geeksforgeeks.org/data-flow-analysis-compiler/

- **University of Michigan**: Course materials tentang compiler optimization [7]
- **LLVM Documentation**: Advanced optimization techniques [8]

[7] https://web.eecs.umich.edu/~weimerw/2015-4610/ca1/ca1.html
[8] https://llvm.org/docs/Passes.html

# Bab 16

## Project Final Presentation dan Review

### 16.1 Tujuan Pembelajaran

Bab ini memfokuskan presentasi dan review **compiler subset C** yang telah dibangun secara bertahap dari Bab 2 hingga Bab 15: spesifikasi token dan grammar (Bab 1, 2, 5), lexer (Bab 3, 4), parser (Bab 6, 8), AST (Bab 9), symbol table (Bab 10), type checking (Bab 11), IR (Bab 12), runtime (Bab 13), code generation (Bab 14), optimasi (Bab 15). Setelah mempelajari bab ini, mahasiswa diharapkan mampu:

1. Mempresentasikan project final (compiler subset C) dengan baik
2. Mendemonstrasikan kemampuan compiler subset C yang telah dibangun
3. Mengevaluasi dan membandingkan tools kompilator (parser generators vs hand-written parsers)
4. Menganalisis trade-off antara waktu kompilasi, kualitas kode, dan efisiensi runtime
5. Melakukan refleksi pembelajaran dan evaluasi diri terhadap seluruh materi
6. Menyusun dokumentasi proyek yang komprehensif

### 16.2 Pendahuluan

Bab ini merupakan puncak dari pembelajaran mata kuliah Teknik Kompilasi. Setelah mempelajari semua fase kompilasi dari Analisis Leksikal (Lexical Analysis) hingga Optimasi (Optimization), mahasiswa diharapkan telah membangun sebuah kompilator lengkap yang dapat mengkompilasi bahasa sederhana menjadi executable code.

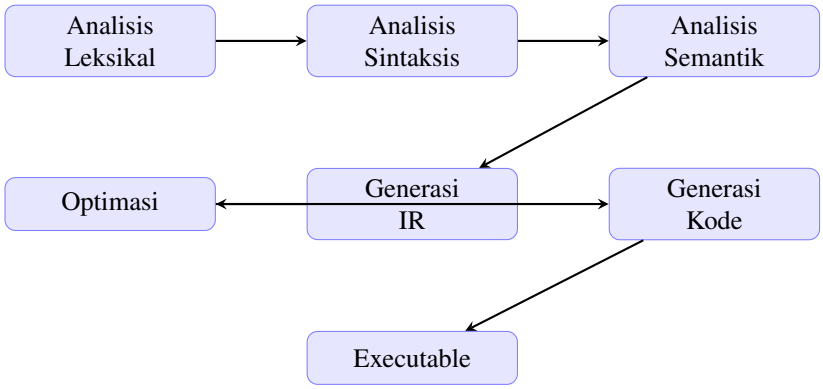
Menurut sumber dari University of Oxford:

> "Evaluate and compare compiler tools (like parser generators) and optimization approaches, analyze trade-offs between compilation time, code quality, and runtime efficiency."[15]

Project final ini tidak hanya menguji kemampuan teknis, tetapi juga kemampuan untuk:

- Mengintegrasikan semua komponen yang telah dipelajari
- Membuat keputusan desain yang tepat
- Mengevaluasi tools dan teknik yang digunakan
- Berkomunikasi secara efektif tentang hasil kerja

Gambar 16.1 menunjukkan arsitektur lengkap kompilator yang harus dibangun dalam project final.




Gambar 16.1: Arsitektur lengkap kompilator untuk project final

## 16.3 Persiapan Presentasi Project Final

Presentasi project final adalah kesempatan untuk menunjukkan hasil kerja keras selama satu semester. Berikut adalah panduan untuk mempersiapkan presentasi yang efektif:

### 16.3.1 Struktur Presentasi

Presentasi sebaiknya mengikuti struktur berikut:

**1. Pendahuluan (5 menit)**

- Perkenalan tim dan project
- Tujuan dan scope bahasa yang dikompilasi
- Overview arsitektur compiler

**2. Demo Compiler (10 menit)**

- Live demonstration: compile dan run program contoh

- Menunjukkan berbagai fitur bahasa yang didukung
- Menampilkan error handling yang baik

**3. Arsitektur dan Implementasi (15 menit)**

- Penjelasan setiap fase kompilasi
- Pilihan desain dan justifikasinya
- Tools dan teknik yang digunakan
- Tantangan yang dihadapi dan solusinya

**4. Evaluasi dan Analisis (10 menit)**

- Perbandingan hand-written vs generator tools
- Trade-off analysis
- Benchmark hasil kompilasi
- Evaluasi kualitas kode yang dihasilkan

**5. Kesimpulan dan Refleksi (5 menit)**

- Lesson learned
- Area untuk improvement
- Kesimpulan

**6. Q&A (5 menit)**

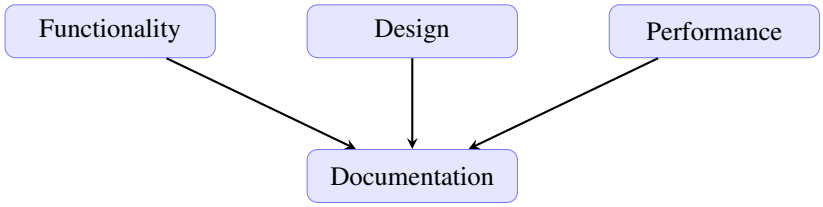
### 16.3.2 Tips Presentasi yang Efektif

1. **Persiapan Demo**: Pastikan demo berjalan lancar dengan test cases yang sudah dipersiapkan
2. **Visual Aids**: Gunakan diagram arsitektur, flowchart, dan contoh kode yang jelas
3. **Time Management**: Latih presentasi untuk memastikan sesuai waktu yang dialokasikan
4. **Anticipate Questions**: Siapkan jawaban untuk pertanyaan umum tentang desain dan implementasi
5. **Show Passion**: Tunjukkan antusiasme terhadap project yang telah dibangun

## 16.4 Demonstrasi Compiler

Demo adalah bagian penting dari presentasi. Demo yang baik menunjukkan bahwa compiler benar-benar berfungsi dan dapat digunakan secara praktis.

Gambar 16.2 menunjukkan aspek-aspek evaluasi project final.




Gambar 16.2: Aspek-aspek evaluasi project final

### 16.4.1 Preparing for Demo

**1. Test Cases yang Komprehensif**

- Program sederhana (hello world, arithmetic)
- Program dengan kontrol flow (if-else, loops)
- Program dengan fungsi dan scope
- Program dengan error (untuk menunjukkan error handling)
- Program yang lebih kompleks (menunjukkan kemampuan compiler)

**2. Environment Setup**

- Pastikan semua dependencies terinstall
- Compile compiler terlebih dahulu
- Siapkan backup plan jika ada masalah teknis
- Test di environment yang sama dengan presentasi

**3. Demo Script**

- Buat script demo yang terstruktur
- Siapkan penjelasan untuk setiap langkah
- Antisipasi kemungkinan error atau masalah

### 16.4.2 Contoh Demo Flow

Berikut adalah contoh alur demo yang efektif:

1. **Hello World**: Menunjukkan compiler dapat menghasilkan executable sederhana

```
// hello.lang
print("Hello, World!");
```

2. **Arithmetic Operations**: Menunjukkan kemampuan menangani ekspresi

```
// calc.lang
int x = 10;
int y = 20;
int result = x + y * 2;
print(result);
```

3. **Control Flow**: Menunjukkan if-else dan loops

```
// control.lang
int i = 0;
while (i < 10) {
    if (i % 2 == 0) {
        print(i);
    }
    i = i + 1;
}
```

4. **Error Handling**: Menunjukkan kualitas error messages

```
// error.lang
int x = 10;
int y = "string";  // Type error
x = undefined_var; // Undefined variable
```

5. **Complex Program**: Menunjukkan kemampuan compiler dengan program yang lebih kompleks

## 16.5 Review Materi: Fase-Fase Kompilasi

Sebelum melakukan evaluasi tools dan teknik, penting untuk mereview kembali semua fase kompilasi yang telah dipelajari:

### 16.5.1 Front-End Phases

**1. Lexical Analysis**

- Memecah source code menjadi tokens
- Implementasi: hand-written atau menggunakan Flex/re2c
- Output: stream of tokens

**2. Syntax Analysis**

- Memverifikasi struktur grammar
- Implementasi: recursive descent, LR parser, atau Bison
- Output: Abstract Syntax Tree (AST)

**3. Semantic Analysis**

- Type checking, scope resolution, name resolution
- Implementasi: tree traversal dengan symbol table
- Output: Annotated AST dengan type information

### 16.5.2 Back-End Phases

**4. Intermediate Code Generation**

- Mengkonversi AST menjadi IR (three-address code)
- Output: Intermediate Representation

**5. Code Optimization**

- Optimasi lokal dan global
- Output: Optimized IR

**6. Code Generation**

- Mengkonversi IR menjadi target code (assembly)
- Output: Assembly code atau machine code

### 16.5.3 Integration Points

Setiap fase harus terintegrasi dengan baik:

- Lexer → Parser: Token stream
- Parser → Semantic Analyzer: AST
- Semantic Analyzer → IR Generator: Annotated AST
- IR Generator → Optimizer: IR
- Optimizer → Code Generator: Optimized IR
- Code Generator → Assembler: Assembly code

## 16.6 Evaluasi Tools: Hand-Written vs Generator-Based

Salah satu aspek penting dalam project final adalah evaluasi tools yang digunakan. Mahasiswa perlu membandingkan pendekatan hand-written dengan generator-based tools.

### 16.6.1 Perbandingan Lexer: Hand-Written vs Flex/re2c

**Hand-Written Lexer**

**Keuntungan:**

- Kontrol penuh atas implementasi
- Error messages yang lebih informatif dan customizable
- Tidak ada dependency eksternal
- Dapat dioptimasi untuk kasus khusus
- Lebih mudah di-debug karena kode lebih readable

**Kekurangan:**

- Lebih banyak waktu development
- Lebih banyak kode boilerplate
- Lebih mudah terjadi error manual
- Perlu implementasi state machine secara manual

**Generator-Based Lexer (Flex/re2c)**

**Keuntungan:**

- Development lebih cepat
- Grammar specification lebih declarative
- Automatically generates efficient DFA
- Less boilerplate code
- Proven algorithms (Thompson's construction, subset construction)

**Kekurangan:**

- Generated code sulit di-debug
- Error messages kurang informatif
- Dependency pada tool eksternal
- Kurang fleksibel untuk kasus edge yang kompleks
- Build process lebih kompleks

### 16.6.2 Perbandingan Parser: Hand-Written vs Bison/Yacc

**Hand-Written Parser (Recursive Descent)**

**Keuntungan:**

- Kode lebih readable dan mudah di-maintain
- Error recovery yang lebih baik dan customizable
- Tidak ada dependency eksternal
- Dapat menangani grammar yang kompleks dengan mudah
- Lebih mudah di-debug

**Kekurangan:**

- Hanya cocok untuk LL(1) grammar
- Lebih banyak kode manual
- Lebih mudah terjadi error dalam implementasi

**Generator-Based Parser (Bison/Yacc)**

**Keuntungan:**

- Mendukung LR, LALR, GLR parsing

- Automatic table generation
- Grammar specification lebih declarative
- Proven algorithms
- Mendukung grammar yang lebih kompleks

**Kekurangan:**

- Generated code sulit di-debug
- Error messages kurang informatif
- Dependency pada tool eksternal
- Build process lebih kompleks
- Kurang fleksibel untuk error recovery yang kompleks

### 16.6.3 Case Study: Real-World Compilers

Banyak compiler production menggunakan pendekatan hybrid atau beralih dari generator ke hand-written:

**GCC (GNU Compiler Collection)**

- Menggunakan hand-written recursive descent parser untuk C dan C++
- Alasan: Better error messages, easier maintenance, better handling of complex grammar
- Trade-off: Lebih banyak kode manual, tetapi lebih maintainable

**Clang (LLVM Compiler)**

- Menggunakan hand-written parser
- Alasan: Superior error diagnostics, better recovery
- Hasil: Error messages yang sangat informatif dan helpful

**Many Educational Compilers**

- Menggunakan Flex + Bison untuk pembelajaran
- Alasan: Lebih cepat untuk development, fokus pada konsep bukan implementasi detail
- Cocok untuk: Prototyping, learning, small languages

## 16.7 Analisis Trade-Off

Menurut sumber dari University of Oxford[15], evaluasi compiler tools harus mempertimbangkan trade-off antara berbagai aspek:

### 16.7.1 Compilation Time vs Code Quality

| Pendekatan | Compilation Time | Code Quality |
|---|---|---|
| Minimal Optimization | Fast | Lower |
| Aggressive Optimization | Slow | Higher |
| Selective Optimization | Medium | Medium-High |

Tabel 16.1: Trade-off compilation time vs code quality

**Pertimbangan:**

- Development phase: Prioritize fast compilation untuk iterasi cepat
- Production build: Prioritize code quality untuk performa runtime
- Selective optimization: Balance antara keduanya

### 16.7.2 Development Time vs Maintainability

- **Generator Tools**: Faster initial development, tetapi mungkin lebih sulit di-maintain untuk grammar yang kompleks
- **Hand-Written**: Slower initial development, tetapi lebih maintainable dalam jangka panjang

### 16.7.3 Error Messages Quality

- **Hand-Written**: Dapat menghasilkan error messages yang sangat informatif dan helpful
- **Generator-Based**: Error messages cenderung generic, perlu custom handling untuk improvement

### 16.7.4 Flexibility vs Correctness

- **Generator Tools**: Enforce grammar constraints, mengurangi human error
- **Hand-Written**: Lebih fleksibel, tetapi lebih mudah terjadi error manual

## 16.8 Benchmarking dan Evaluasi Kinerja

Evaluasi compiler tidak hanya tentang correctness, tetapi juga tentang performa. Berikut adalah metrik yang dapat digunakan:

### 16.8.1 Metrik Compiler Performance

**1. Compilation Speed**

- Waktu kompilasi untuk berbagai ukuran program
- Throughput (lines/second atau tokens/second)
- Memory usage selama kompilasi

**2. Generated Code Quality**

- Execution time dari program yang dikompilasi
- Code size (executable size)
- Memory footprint
- Instruction count

**3. Compiler Correctness**

- Test coverage
- Number of bugs found
- Error detection rate

### 16.8.2 Contoh Benchmark Results

Berikut adalah contoh format untuk melaporkan hasil benchmark:

| Metric | Baseline | Optimized | Improvement |
|---|---|---|---|
| Compilation Time (s) | 2.5 | 3.1 | -24% (slower) |
| Generated Code Size (KB) | 64 | 48 | 25% reduction |
| Execution Time (ms) | 100 | 75 | 25% faster |
| Memory Usage (MB) | 8 | 6 | 25% reduction |

Tabel 16.2: Contoh hasil benchmark compiler

### 16.8.3 Test Suite

Untuk evaluasi yang komprehensif, siapkan test suite yang mencakup:

1. **Unit Tests**: Test setiap fase secara terpisah
    - Lexer tests: Valid/invalid tokens
    - Parser tests: Valid/invalid syntax
    - Semantic tests: Type checking, scope resolution
    - Code generation tests: Correctness of generated code
2. **Integration Tests**: Test seluruh pipeline
    - End-to-end compilation
    - Error propagation through phases
    - Optimization correctness
3. **Performance Tests**: Test dengan program besar
    - Compilation time
    - Memory usage
    - Generated code performance
4. **Regression Tests**: Test untuk memastikan tidak ada regresi

## 16.9 Dokumentasi Proyek

Dokumentasi yang baik adalah bagian penting dari project final. Dokumentasi harus mencakup:

### 16.9.1 README.md

README harus berisi:

- **Overview**: Deskripsi singkat tentang compiler
- **Features**: Fitur-fitur yang didukung
- **Build Instructions**: Cara mengkompilasi compiler
- **Usage**: Cara menggunakan compiler
- **Examples**: Contoh program dan cara mengkompilasinya
- **Architecture**: Overview arsitektur compiler
- **Testing**: Cara menjalankan test suite

### 16.9.2 Design Document

Design document mencakup:

- **Language Specification**: Grammar, syntax, semantics
- **Architecture Overview**: Diagram arsitektur compiler
- **Component Design**: Desain setiap fase kompilasi
- **Data Structures**: AST nodes, symbol table, IR format
- **Algorithm Choices**: Justifikasi pilihan algoritma
- **Trade-offs**: Diskusi tentang trade-off yang dibuat

### 16.9.3 API Documentation

Jika compiler menyediakan library atau API:

- Function signatures
- Parameter descriptions
- Return values
- Usage examples

## 16.10 Refleksi Pembelajaran

Refleksi adalah bagian penting dari proses pembelajaran. Mahasiswa diharapkan melakukan refleksi terhadap:

### 16.10.1 Technical Skills Acquired

- **Lexical Analysis**: Kemampuan mengimplementasikan lexer
- **Parsing**: Pemahaman tentang grammar dan parsing techniques
- **Semantic Analysis**: Kemampuan melakukan type checking dan scope resolution
- **Code Generation**: Kemampuan menghasilkan target code
- **Optimization**: Pemahaman tentang optimasi kompilator
- **Software Engineering**: Kemampuan mengintegrasikan komponen-komponen besar

### 16.10.2 Challenges Faced

Identifikasi tantangan yang dihadapi:

- Technical challenges (implementasi, debugging)
- Design challenges (trade-offs, architecture decisions)
- Time management challenges
- Team collaboration challenges (jika project team-based)

### 16.10.3 Lessons Learned

Rangkum pembelajaran:

- Apa yang bekerja dengan baik?
- Apa yang tidak bekerja seperti yang diharapkan?
- Apa yang akan dilakukan berbeda jika memulai lagi?
- Insight tentang compiler design dan implementation

### 16.10.4 Areas for Improvement

Identifikasi area untuk improvement:

- Fitur yang belum diimplementasikan
- Optimasi yang dapat ditambahkan
- Error handling yang dapat ditingkatkan
- Dokumentasi yang dapat diperbaiki
- Testing yang dapat diperluas

## 16.11 Best Practices untuk Project Final

Berdasarkan pengalaman dan best practices dari berbagai compiler projects:

### 16.11.1 Code Quality

- **Clean Code**: Kode yang readable dan well-structured
- **Modularity**: Komponen yang terpisah dengan jelas
- **Error Handling**: Comprehensive error handling dan reporting
- **Comments**: Dokumentasi yang adequate dalam kode
- **Consistency**: Konsistensi dalam coding style

### 16.11.2 Testing

- **Test Coverage**: Test coverage yang komprehensif
- **Edge Cases**: Test untuk edge cases dan error conditions
- **Automated Testing**: Automated test suite
- **Regression Testing**: Test untuk mencegah regresi

### 16.11.3 Documentation

- **Completeness**: Dokumentasi yang lengkap
- **Clarity**: Dokumentasi yang jelas dan mudah dipahami
- **Examples**: Contoh yang membantu
- **Maintenance**: Dokumentasi yang up-to-date

### 16.11.4 Presentation

- **Preparation**: Persiapan yang matang
- **Clarity**: Presentasi yang jelas dan terstruktur
- **Demo**: Demo yang berjalan lancar
- **Time Management**: Manajemen waktu yang baik

## 16.12 Kesimpulan

Bab ini telah membahas:

1. **Project Final Presentation**: Struktur dan tips untuk presentasi yang efektif
2. **Demonstrasi Compiler**: Cara mempersiapkan dan melakukan demo yang baik
3. **Review Materi**: Review semua fase kompilasi yang telah dipelajari
4. **Evaluasi Tools**: Perbandingan hand-written vs generator-based tools
5. **Analisis Trade-Off**: Trade-off antara berbagai aspek compiler design
6. **Benchmarking**: Metrik dan cara melakukan evaluasi kinerja
7. **Dokumentasi**: Best practices untuk dokumentasi proyek
8. **Refleksi Pembelajaran**: Cara melakukan refleksi yang konstruktif

Project final adalah kesempatan untuk mengintegrasikan semua pengetahuan yang telah diperoleh selama semester. **Compiler subset C** yang dibangun dari Bab 2 hingga Bab 15—dengan komponen lexer proyek (`simplec.l`), parser proyek (`simplec.y`), AST, symbol table, type checking, IR, code generation, dan optimasi—menjadi bukti nyata penguasaan fase-fase kompilasi. Melalui project ini, mahasiswa tidak hanya menunjukkan kemampuan teknis, tetapi juga kemampuan untuk membuat keputusan desain, mengevaluasi tools dan teknik, dan berkomunikasi secara efektif tentang hasil kerja.

## 16.13 Referensi dan Bahan Bacaan Lanjutan

Untuk memperdalam pemahaman tentang evaluasi compiler dan best practices:

- **University of Oxford Compilers Course**[15]: Materials tentang evaluasi tools dan trade-off analysis
- **Dragon Book**[1]: Bab tentang compiler optimization dan code generation evaluation
- **Engineering a Compiler**[5]: Best practices untuk compiler design dan implementation
- **Compiler Construction Resources**:
  - UC San Diego CSE 231[3]: Course materials dengan project examples
  - Northeastern University CS 4410[4]: Comprehensive compiler design course

  - Johns Hopkins University EN.601.428[6]: Course tentang compilers dan interpreters

- **Open Source Compiler Projects**:
  - TinyCC (TCC): https://bellard.org/tcc/ - Contoh compiler sederhana
  - AnjaneyaTripathi's C Compiler: https://github.com/AnjaneyaTripathi/c-compiler - Educational compiler dengan Flex & Bison
  - LLVM: https://llvm.org/ - Production compiler infrastructure

*Selamat! Anda telah menyelesaikan perjalanan pembelajaran Teknik Kompilasi. Project final adalah kesempatan untuk menunjukkan semua yang telah Anda pelajari dan bangun. Semoga sukses dengan presentasi dan project final Anda!*

# Bab 17

## Kumpulan Latihan

### 17.1 Soal

#### 17.1.1 Latihan Bab 1: Pengenalan Kompilator dan Fase-Fase Kompilasi

1. Jelaskan perbedaan antara kompilator dan interpreter. Berikan contoh bahasa pemrograman yang menggunakan masing-masing pendekatan.

2. Buatlah diagram yang menunjukkan alur kerja kompilator dari source code C sampai menjadi executable. Sertakan semua fase yang telah dipelajari.

3. Identifikasi fase kompilasi yang bertanggung jawab untuk:

    - Menghapus whitespace dan comments
    - Memeriksa apakah variabel dideklarasi sebelum digunakan
    - Mengoptimasi kode dengan menghilangkan kode yang tidak pernah dieksekusi
    - Mengkonversi ekspresi `a + b * c` menjadi three-address code

4. Jelaskan mengapa kompilator modern menggunakan pendekatan multi-pass. Apa keuntungannya dibanding single-pass?

5. Carilah informasi tentang satu kompilator open source (misalnya GCC, Clang, atau TinyCC) dan identifikasi:

    - Bahasa sumber dan target yang didukung
    - Tools yang digunakan untuk lexical dan syntax analysis
    - Format intermediate representation yang digunakan

### 17.1.2 Latihan Bab 2: Regular Expression dan Finite Automata

1. Buatlah regular expression untuk:
   - Email address sederhana (format: `user@domain.com`)
   - Phone number (format: `+62-812-3456-7890`)
   - C-style comment (`/* ... */`)
2. Konstruksi NFA untuk regular expression berikut menggunakan algoritma Thompson:
   - `a*b+`
   - `(a|b)*ab`
   - `[0-9]+(\.[0-9]+)?`
3. Konversi NFA dari soal nomor 2 menjadi DFA menggunakan subset construction. Gambarkan state diagram untuk DFA yang dihasilkan.
4. Implementasikan kelas `NFA` dan `DFA` dalam C++ dengan fungsi:
   - Konstruksi NFA dari regular expression (sederhana)
   - Konversi NFA ke DFA
   - Simulasi DFA untuk string input
5. Buat program recognizer yang dapat mengenali token-token berikut:
   - Identifier: `[a-zA-Z_][a-zA-Z0-9_]*`
   - Integer: `[0-9]+`
   - Float: `[0-9]+\.[0-9]+`
   - Operator: `+|-|*|/|=|==|!=`
6. Jelaskan mengapa DFA lebih efisien untuk simulasi dibanding NFA. Berikan contoh kompleksitas waktu untuk keduanya.
7. Implementasikan algoritma minimisasi DFA (dapat menggunakan versi sederhana). Uji dengan DFA yang dihasilkan dari soal nomor 3.

### 17.1.3 Latihan Bab 3: Implementasi Lexer

1. **Implementasi Dasar**: Implementasikan lexer sederhana yang dapat mengenali:
   - Identifier (huruf, angka, underscore)
   - Integer literals

- Operator dasar: `+`, `-`, `*`, `/`, `=`
- Punctuation: `;`, `,`, `(`, `)`

2. **Handling Comments**: Tambahkan support untuk:
   - Single-line comments (`//`)
   - Multi-line comments (`/* */`)
   - Error handling untuk unclosed block comments
3. **String Literals**: Implementasikan scanning untuk string literals dengan:
   - Support escape sequences: `\n`, `\t`, `\"`, `\\`
   - Error handling untuk unclosed strings
4. **Float Literals**: Extend number scanning untuk mendukung:
   - Float dengan decimal point: `3.14`
   - Scientific notation: `1.5e10`, `2.3E-5`
5. **Unit Testing**: Buat test suite yang mencakup:
   - Valid tokens dari semua kategori
   - Edge cases (unclosed strings, invalid characters, dll.)
   - Position tracking (line dan column)
6. **Error Recovery**: Implementasikan error recovery strategy:
   - Skip invalid characters dan lanjut scanning
   - Report multiple errors dalam satu pass jika memungkinkan
7. **Performance**: Analisis dan optimasi:
   - Bandingkan performa dengan lexer generator (jika tersedia)
   - Identifikasi bottleneck dalam implementasi

### 17.1.4 Latihan Bab 4: Lexer Generator

1. Buatlah specification file Flex untuk mengenali token-token berikut:
   - Keywords: `for`, `break`, `continue`
   - String literals (dengan escape sequences: `\n`, `\t`, `\"`)
   - Character literals (dalam single quotes)

- Operators: ++, −, +=, −=

2. Implementasikan lexer menggunakan re2c untuk bahasa yang sama seperti soal 1. Bandingkan kompleksitas specification antara Flex dan re2c.

3. Jelaskan kapan sebaiknya menggunakan hand-written lexer dan kapan menggunakan generator. Berikan contoh kasus untuk masing-masing.

4. Buatlah program yang mengintegrasikan Flex lexer dengan Bison parser sederhana untuk:

   - Parsing ekspresi aritmatika: `a + b * c`
   - Parsing assignment: `x = 42;`
   - Parsing conditional: `if (x > 0) { ... }`

5. Analisis performa: Buat benchmark untuk membandingkan:

   - Hand-written lexer (dari Bab 3)
   - Flex-generated lexer
   - re2c-generated lexer

   Gunakan input file yang besar (misalnya 10MB) dan ukur waktu eksekusi.

6. Jelaskan bagaimana Flex menangani longest match dan rule priority. Berikan contoh yang menunjukkan perbedaan hasil ketika urutan rule diubah.

### 17.1.5 Latihan Bab 5: Context-Free Grammar dan Parsing

1. Tuliskan grammar dalam BNF untuk:

   - Ekspresi boolean dengan operator AND, OR, NOT
   - Assignment statement: `variable = expression;`
   - For loop: `for (init; condition; update) statement`

2. Konversi grammar berikut ke EBNF:

```
<list> ::= <item> | <list> , <item>
<item> ::= <number> | <string>
```

3. Buatlah leftmost dan rightmost derivation untuk ekspresi `(2 + 3) * 4` menggunakan grammar ekspresi aritmatika yang telah dipelajari.

4. Gambarkan parse tree untuk ekspresi berikut menggunakan grammar yang sesuai:

- `1 + 2 * 3`
- `10 / 2 - 3`
- `(5 + 3) * 2`

5. Identifikasi apakah grammar berikut ambiguous. Jika ya, berikan contoh string yang dapat di-parse dengan lebih dari satu cara:

```
S → a S a | b S b | a | b | epsilon
```

6. Eliminasi left recursion dari grammar berikut:

```
A → A + B | A - B | B
B → B * C | B / C | C
C → ( A ) | number
```

7. Lakukan left factoring pada grammar berikut:

```
S → if E then S else S
  | if E then S
  | while E do S
  | id = E
```

8. Rancang grammar untuk bahasa sederhana yang mendukung:

   - Variable declarations: `int x;`
   - Assignments: `x = 5;`
   - If-else statements
   - While loops
   - Arithmetic expressions

   Tulis grammar dalam EBNF dan berikan contoh program valid.

### 17.1.6 Latihan Bab 6: Top-Down Parsing dan Recursive Descent

1. Implementasikan recursive descent parser untuk grammar berikut:

```
S -> if E then S else S | id := E | while E do S
E -> E + T | T
T -> T * F | F
F -> ( E ) | id | num
```

   **Perhatikan:** Grammar ini memiliki left recursion. Anda perlu mengeliminasi left recursion terlebih dahulu.

2. Modifikasi parser ekspresi aritmatika untuk menambahkan operator unary minus (misalnya `-5` atau `-(3 + 4)`).

3. Implementasikan error recovery pada parser yang telah Anda buat. Test dengan input yang mengandung multiple errors.

4. Buatlah parser yang dapat mengevaluasi ekspresi boolean dengan operator `AND`, `OR`, dan `NOT`. Perhatikan precedence: `NOT` > `AND` > `OR`.

5. Bandingkan recursive descent parser dengan table-driven LL parser. Apa keuntungan dan kerugian masing-masing pendekatan?

6. Implementasikan parser untuk ekspresi dengan right-associative operator (misalnya operator assignment =).

### 17.1.7 Latihan Bab 7: Bottom-Up Parsing dan Parser Generator

1. Jelaskan perbedaan antara top-down dan bottom-up parsing. Berikan contoh situasi di mana masing-masing lebih cocok digunakan.

2. Untuk grammar berikut:

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

   Lakukan shift-reduce parsing manual untuk input `id + id * id`. Tunjukkan stack, input, dan action pada setiap langkah.

3. Jelaskan perbedaan antara SLR(1), CLR(1), dan LALR(1). Mengapa LALR(1) menjadi pilihan populer untuk parser generator?

4. Buatlah file Bison sederhana untuk grammar ekspresi aritmatika dengan:

   - Operator +, -, *, / dengan precedence yang benar
   - Parentheses untuk grouping
   - Semantic actions yang mencetak ekspresi yang di-parse

5. Integrasikan Flex lexer dengan Bison parser untuk membuat calculator sederhana yang dapat mengevaluasi ekspresi aritmatika.

6. Implementasikan error recovery dalam Bison untuk menangani syntax error dengan baik. Test dengan input yang mengandung error.

7. Bandingkan performa dan kompleksitas implementasi antara recursive descent parser (top-down) dan parser yang di-generate oleh Bison (bottom-up) untuk grammar yang sama.

### 17.1.8 Latihan Bab 8: Semantic Actions dan AST Construction

1. Buatlah parser untuk ekspresi boolean yang mendukung:
   - Operasi AND (`&&`), OR (`||`), NOT (`!`)
   - Perbandingan (<, >, ==, !=)
   - Precedence yang benar
2. Modifikasi calculator untuk mendukung:
   - Variabel (assignment dan penggunaan)
   - Fungsi matematika dasar (sin, cos, sqrt)
   - Error handling yang lebih baik
3. Buatlah parser untuk konfigurasi file sederhana dengan format:

```
key1 = value1
key2 = value2
section {
    key3 = value3
}
```

4. Implementasikan semantic actions untuk membangun AST lengkap dari subset C, kemudian buat fungsi untuk:
   - Print AST dalam format tree
   - Evaluate AST (jika semua nilai diketahui)
   - Optimize AST (constant folding)
5. Pelajari dokumentasi Bison dan jelaskan perbedaan antara:
   - LALR(1) parsing (default Bison)
   - GLR parsing
   - Kapan masing-masing digunakan?

### 17.1.9 Latihan Bab 9: Abstract Syntax Tree (AST)

1. Buatlah AST untuk program berikut:

```
int x = 10;
int y = 20;
int z = x + y * 2;
```

   Gambarkan struktur AST-nya.

2. Implementasikan AST node untuk:

   - For loop statement
   - Array access expression
   - String literal
   - Boolean literal

3. Buatlah visitor yang menghitung jumlah node dalam AST.

4. Buatlah visitor yang melakukan constant folding sederhana (misalnya: `3 + 5` $\rightarrow$ `8`).

5. Modifikasi PrettyPrinter untuk menghasilkan output yang lebih mirip dengan source code asli.

6. Implementasikan AST visualizer yang menghasilkan output dalam format Graphviz DOT untuk visualisasi grafis.

7. Jelaskan mengapa AST lebih disukai daripada parse tree untuk fase-fase selanjutnya dalam kompilator.

8. Buatlah contoh program yang menunjukkan perbedaan antara pre-order, post-order, dan in-order traversal pada AST ekspresi `(a + b) * (c - d)`.

### 17.1.10 Latihan Bab 10: Symbol Table dan Scope Management

1. Implementasikan symbol table lengkap dengan fitur-fitur berikut:

   - Insert, lookup, delete operations
   - Nested scopes support
   - Shadowing detection dengan warning
   - Visualisasi symbol table

2. Buatlah test cases untuk berbagai skenario:

- Nested scopes dengan shadowing
- Duplicate declaration dalam scope yang sama
- Undeclared identifier usage
- Function parameters dalam function scope

3. Analisis program berikut dan buat visualisasi symbol table-nya:

```
    int a = 1;
    int b = 2;

    void func1(int x) {
        int a = 10;
        int c = 20;

        {
            int b = 30;
            int d = 40;
        }
    }

    void func2() {
        int x = 100;
        int y = 200;
    }

```

4. Implementasikan fitur tambahan:

   - Tracking semua usages dari setiap identifier (bukan hanya deklarasi)
   - Support untuk forward declaration
   - Scope numbering untuk debugging

5. Bandingkan implementasi symbol table menggunakan:

   - Stack of hash tables (seperti yang dibahas)
   - Single hash table dengan per-name stacks

   Apa kelebihan dan kekurangan masing-masing pendekatan?

### 17.1.11 Latihan Bab 11: Type Checking dan Semantic Analysis

1. Jelaskan perbedaan antara static type checking dan dynamic type checking. Berikan contoh bahasa pemrograman untuk masing-masing.

2. Implementasikan type checker sederhana untuk ekspresi aritmatika dengan mendukung:

- Operasi +, -, *, / untuk int dan float
- Operasi perbandingan ==, !=, <, > yang menghasilkan boolean
- Type promotion (int $\rightarrow$ float)

3. Buatlah fungsi untuk mengecek type compatibility dengan aturan:
   - Exact match selalu kompatibel
   - int dapat di-assign ke float (implicit conversion)
   - float tidak dapat di-assign ke int tanpa explicit cast
4. Implementasikan semantic error detection untuk:
   - Undeclared variable
   - Type mismatch pada assignment
   - Wrong number of arguments pada function call
5. Buatlah test cases untuk type checker yang mencakup:
   - Valid expressions (harus pass type checking)
   - Invalid expressions (harus menghasilkan error yang sesuai)
   - Edge cases (null, empty, boundary values)
6. Jelaskan mengapa annotated AST diperlukan. Bagaimana informasi tipe pada AST digunakan pada fase code generation?
7. Implementasikan type checking untuk if statement dan while loop. Pastikan kondisi harus bertipe boolean.
8. Bandingkan nominal typing dan structural typing. Berikan contoh situasi di mana masing-masing pendekatan lebih menguntungkan.

### 17.1.12 Latihan Bab 12: Intermediate Code Generation

1. Buatlah TAC untuk ekspresi berikut:
   - `a = b + c * d - e`
   - `x = (a + b) * (c - d)`
   - `y = -a + b * -c`
2. Implementasikan generator TAC untuk:
   - Unary operations (negation, logical NOT)

- Array access (`array[index]`)
- Member access (`object.member`)

3. Buatlah TAC untuk program berikut:

```
int i, sum;
sum = 0;
for (i = 1; i <= 10; i = i + 1) {
    sum = sum + i;
}
```

4. Implementasikan common subexpression elimination untuk basic block. Test dengan contoh:

```
x = a + b * c
y = a + b * c
z = (a + b) * c
w = (a + b) * c
```

5. Jelaskan perbedaan antara three-address code dan quadruples. Kapan quadruples lebih menguntungkan?

6. Buatlah generator TAC untuk switch statement. Bagaimana cara menangani multiple cases?

7. Implementasikan optimasi constant folding pada TAC generator. Contoh: `x = 3 + 5` langsung menjadi `x = 8`

8. Bandingkan pendekatan top-down (generate sambil traverse) dan bottom-up (build IR structure dulu) untuk TAC generation. Apa keuntungan masing-masing?

### 17.1.13 Latihan Bab 13: Runtime Environment dan Memory Management

1. Jelaskan perbedaan antara static, stack, dan heap allocation. Berikan contoh penggunaan masing-masing.

2. Buatlah diagram activation records untuk program berikut saat `factorial(3)` sedang dieksekusi:

```
1  int factorial(int n) {
2      if (n <= 1) return 1;
3      return n * factorial(n - 1);
4  }
5
6  int main() {
```

```
        int result = factorial(3);
        return 0;
    }

```

3. Implementasikan runtime stack simulator yang dapat menangani:
   - Function calls dengan parameters
   - Local variables
   - Return values
   - Nested function calls
4. Jelaskan calling sequence untuk fungsi dengan 3 parameters. Sertakan assembly code (simplified).
5. Bandingkan manual memory management dengan garbage collection. Apa keuntungan dan kekurangan masing-masing?
6. Implementasikan mark-and-sweep garbage collector sederhana yang dapat:
   - Mark reachable objects dari root set
   - Sweep dan free unreachable objects
   - Handle cyclic references
7. Analisis memory layout untuk program berikut dan identifikasi di mana setiap variabel dialokasikan:

```
    int global;
    static int static_var;

    void func() {
        int local;
        static int static_local;
        int* ptr = new int;
        // ...
    }

```

8. Jelaskan apa yang terjadi pada stack saat terjadi recursive call yang sangat dalam (misalnya 1000 level). Apa masalah yang mungkin terjadi?

### 17.1.14 Latihan Bab 14: Code Generation

1. Implementasikan code generator sederhana untuk operasi aritmatika dasar (+, -, *, /) yang menghasilkan RISC-V assembly.

2. Buatlah code generator untuk assignment statement. Test dengan berbagai skenario:
   - Assignment dari constant ke variabel
   - Assignment dari variabel ke variabel
   - Assignment dari ekspresi ke variabel
3. Implementasikan local register allocation dengan algoritma sederhana. Test dengan basic block yang memiliki lebih banyak variabel aktif daripada jumlah register yang tersedia.
4. Buatlah code generator untuk function call sederhana dengan 2-3 parameter. Implementasikan calling convention RISC-V.
5. Test workflow lengkap: compile → assemble → link → run untuk program sederhana yang menghitung `a * b + c`.
6. Bandingkan code yang dihasilkan untuk ekspresi `a + b * c` dengan dan tanpa optimasi register allocation. Hitung jumlah instruksi dan memory access.
7. Implementasikan code generator untuk conditional statement (if-else) dengan branch instructions.
8. Buatlah code generator untuk loop (for/while) dengan label dan jump instructions.

### 17.1.15 Latihan Bab 15: Optimasi Kompilator

1. Identifikasi basic blocks dari kode three-address berikut:

```
t1 = a + b
t2 = c * d
if t1 > t2 goto L1
t3 = t1 - t2
goto L2
L1: t4 = t1 * t2
L2: t5 = t3 + t4
return t5
```

2. Lakukan constant folding pada kode berikut:

```
t1 = 5 + 3
t2 = t1 * 2
t3 = 10 / 2
x = t2 + t3
```

3. Lakukan constant propagation dan folding pada kode berikut:

```
x = 10
y = 20
t1 = x + 5
t2 = y * 2
z = t1 + t2
```

4. Identifikasi dan hapus dead code dari kode berikut:

```
x = 10
y = 20
t1 = x + y
t2 = 5 + 3       // Dead
t3 = t2 - 2      // Dead
z = t1 * 2
return z
```

5. Implementasikan optimizer sederhana yang melakukan:

   - Constant folding
   - Constant propagation (local)
   - Dead code elimination (basic)

   Uji dengan beberapa contoh program dan evaluasi hasilnya.

6. Jelaskan perbedaan antara:

   - Local optimization vs global optimization
   - Machine-independent vs machine-specific optimization
   - Constant folding vs constant propagation

7. Buatlah benchmark untuk mengevaluasi efektivitas optimasi. Bandingkan:

   - Ukuran kode sebelum dan sesudah optimasi
   - Waktu eksekusi sebelum dan sesudah optimasi
   - Waktu kompilasi sebelum dan sesudah optimasi

### 17.1.16 Latihan Bab 16: Evaluasi dan Project Final

1. **Prepare Presentation**:

   - Buat outline presentasi untuk project final Anda
   - Siapkan demo script dengan minimal 5 test cases

- Latih presentasi untuk memastikan sesuai waktu

2. **Tool Evaluation**:
   - Buat tabel perbandingan hand-written vs generator tools yang Anda gunakan
   - Identifikasi trade-off yang Anda hadapi
   - Justifikasi pilihan tools yang Anda buat
3. **Benchmarking**:
   - Siapkan test suite dengan berbagai ukuran program
   - Ukur metrik: compilation time, code size, execution time
   - Buat laporan benchmark dengan tabel dan analisis
4. **Documentation**:
   - Tulis README.md yang komprehensif
   - Buat design document
   - Dokumentasikan API jika ada
5. **Reflection**:
   - Tulis refleksi pembelajaran (minimal 500 kata)
   - Identifikasi 3 challenges utama yang dihadapi
   - Rangkum 5 lessons learned yang paling penting
   - Identifikasi 3 area untuk improvement
6. **Peer Review**:
   - Review project final dari teman sekelas
   - Berikan feedback konstruktif
   - Bandingkan pendekatan yang berbeda

## 17.2 Jawaban

### 17.2.1 Jawaban Latihan Bab 1: Pengenalan Kompilator dan Fase-Fase Kompilasi

1. **Perbedaan antara kompilator dan interpreter:**

   **Kompilator:**

- Menerjemahkan seluruh program sekaligus sebelum dieksekusi
- Hasil translasi disimpan dalam file terpisah (executable) yang kemudian dapat dieksekusi langsung oleh sistem operasi atau mesin virtual
- Sekali compile, berkali-kali run
- Contoh bahasa: C, C++, Rust, Go

**Interpreter:**

- Menerjemahkan dan mengeksekusi program baris demi baris secara langsung tanpa menghasilkan file terpisah untuk eksekusi
- Interpret setiap kali run
- Contoh bahasa: Python, JavaScript, Ruby

**Hybrid Approach:** Beberapa bahasa modern menggunakan pendekatan hybrid, seperti Java yang dikompilasi menjadi bytecode, kemudian diinterpretasi oleh JVM (Java Virtual Machine), atau menggunakan JIT (Just-In-Time compilation). Contoh: Java, C#, Python (dengan bytecode).

2. **Diagram alur kerja kompilator dari source code C sampai menjadi executable:**

   Alur kerja kompilator meliputi fase-fase berikut:

   (a) **Preprocessing**: Memproses directive khusus seperti `#include`, `#define`

   (b) **Lexical Analysis**: Memecah source code menjadi token-token (identifiers, keywords, operators, literals)

   (c) **Syntax Analysis**: Menganalisis struktur grammar dan membangun parse tree atau Abstract Syntax Tree (AST)

   (d) **Semantic Analysis**: Memeriksa aturan semantik bahasa, type checking, scope resolution

   (e) **Intermediate Code Generation**: Mengubah AST menjadi intermediate representation (misalnya three-address code)

   (f) **Code Optimization**: Mengoptimasi kode intermediate untuk meningkatkan efisiensi

   (g) **Code Generation**: Menghasilkan target code (assembly atau machine code)

   (h) **Assembling**: Mengubah assembly code menjadi object code (file .o atau .obj)

   (i) **Linking**: Menyatukan object files dan library files menjadi satu executable file

3. **Identifikasi fase kompilasi:**

- **Menghapus whitespace dan comments**: **Lexical Analysis** - Fase ini membaca source code karakter demi karakter, mengelompokkan karakter menjadi token-token bermakna, dan mengeliminasi whitespace, comments, serta karakter yang tidak relevan.
- **Memeriksa apakah variabel dideklarasi sebelum digunakan**: **Semantic Analysis** - Fase ini melakukan scope resolution dan name resolution untuk memastikan setiap identifier merujuk ke deklarasi yang valid.
- **Mengoptimasi kode dengan menghilangkan kode yang tidak pernah dieksekusi**: **Code Optimization** - Fase ini melakukan dead code elimination untuk menghapus kode yang tidak memiliki efek pada perilaku program yang dapat diamati.
- **Mengkonversi ekspresi `a + b * c` menjadi three-address code**: **Intermediate Code Generation** - Fase ini mengubah AST menjadi intermediate representation seperti three-address code. Contoh hasilnya:

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

4. **Alasan kompilator modern menggunakan pendekatan multi-pass:**

   **Keuntungan multi-pass:**

   - **Pemisahan tanggung jawab**: Setiap pass memiliki tugas spesifik, membuat implementasi lebih modular dan mudah di-maintain
   - **Optimasi yang lebih baik**: Informasi yang dikumpulkan dalam pass pertama dapat digunakan untuk optimasi di pass berikutnya
   - **Portabilitas**: Front-end menghasilkan IR, back-end mengkonsumsi IR. Perubahan pada satu sisi tidak mempengaruhi sisi lain
   - **Retargeting**: Untuk menambahkan dukungan target baru, cukup menambahkan code generator untuk IR tersebut tanpa mengubah front-end
   - **Analisis yang lebih mendalam**: Multi-pass memungkinkan analisis lintas fase yang lebih komprehensif

   **Keuntungan dibanding single-pass:**

   - Single-pass memiliki keterbatasan dalam melakukan analisis yang memerlukan informasi dari seluruh program
   - Multi-pass memungkinkan optimasi yang lebih agresif dan akurat
   - Multi-pass memudahkan debugging karena setiap fase dapat diuji secara terpisah

5. **Contoh kompilator open source: GCC (GNU Compiler Collection)**

- **Bahasa sumber dan target yang didukung**:
  - Bahasa sumber: C, C++, Fortran, Ada, Go, dan lainnya
  - Target: x86, ARM, RISC-V, MIPS, dan banyak arsitektur lainnya
- **Tools untuk lexical dan syntax analysis**:
  - GCC menggunakan hand-written recursive descent parser untuk C dan C++ (bukan generator tools)
  - Alasan: Better error messages, easier maintenance, better handling of complex grammar
- **Format intermediate representation**:
  - GCC menggunakan GIMPLE (Generic Intermediate Representation) sebagai IR utama
  - Juga menggunakan RTL (Register Transfer Language) untuk optimasi tingkat rendah
  - GIMPLE adalah three-address code yang lebih high-level

### 17.2.2 Jawaban Latihan Bab 2: Regular Expression dan Finite Automata

1. **Regular expression untuk:**
   - **Email address sederhana** (`user@domain.com`):

     ```
     [a-zA-Z0-9_]+@[a-zA-Z0-9]+\.[a-zA-Z]{2,}
     ```

     Penjelasan: `[a-zA-Z0-9_]+` untuk user, `@` literal, `[a-zA-Z0-9]+` untuk domain, `\.` untuk titik literal, `[a-zA-Z]{2,}` untuk ekstensi domain (minimal 2 karakter).
   - **Phone number** (`+62-812-3456-7890`):

     ```
     \+62-[0-9]{3,4}-[0-9]{4}-[0-9]{4}
     ```

     Penjelasan: `\+` untuk tanda plus literal, `62` untuk kode negara, `-` untuk separator, `[0-9]{3,4}` untuk operator (3-4 digit), `[0-9]{4}` untuk nomor (4 digit).
   - **C-style comment** (`/* ... */`):

     ```
     /\*([^*]|\*+[^*/])*\*+/
     ```

     Penjelasan: `/\*` untuk awal comment, pola untuk konten berupa karakter selain asterisk atau sekuens asterisk yang tidak diikuti slash, `*+/` untuk akhir comment.

2. **Konstruksi NFA menggunakan algoritma Thompson:**

   Algoritma Thompson menggunakan template untuk setiap operasi regex:

   - **Literal**: State awal dengan transisi ke state akhir menggunakan karakter tersebut
   - **Concatenation (RS)**: Menghubungkan NFA untuk R dan S menggunakan $\epsilon$-transition
   - **Union (R|S)**: Menggunakan $\epsilon$-transitions untuk branching dari state awal ke NFA R dan S, kemudian menggabungkan ke state akhir
   - **Kleene Star (R*)**: Membuat loop dengan $\epsilon$-transitions yang memungkinkan nol atau lebih pengulangan R

   Untuk `a*b+`:

   - Buat NFA untuk `a*` dengan loop menggunakan $\epsilon$-transitions
   - Concatenate dengan NFA untuk `b+` (satu atau lebih b)

   Untuk `(a|b)*ab`:

   - Buat NFA untuk `(a|b)*` dengan union dan Kleene star
   - Concatenate dengan NFA untuk literal `a` dan `b`

   Untuk `[0-9]+(\.[0-9]+)?`:

   - Buat NFA untuk `[0-9]+` (satu atau lebih digit)
   - Buat optional group `(\.[0-9]+)?` dengan $\epsilon$-transition untuk kasus opsional

3. **Konversi NFA ke DFA menggunakan subset construction:**

   Algoritma subset construction:

   (a) Mulai dengan $\epsilon$-closure dari start state NFA sebagai state awal DFA

   (b) Untuk setiap state DFA dan setiap input symbol:
      - Hitung $\epsilon$-closure dari semua state yang dapat dicapai dengan symbol tersebut
      - Jika hasilnya non-empty dan belum ada, tambahkan sebagai state baru DFA

   (c) State DFA adalah accept state jika mengandung accept state dari NFA

   (d) Ulangi sampai tidak ada state baru

   State diagram DFA akan menunjukkan transisi deterministik (setiap state memiliki tepat satu transisi untuk setiap input symbol).

4. **Implementasi kelas NFA dan DFA dalam C++:**

   Struktur dasar:

- **Kelas NFA**: Menyimpan states, transitions (termasuk $\epsilon$-transitions), start state, dan accept states
- **Kelas DFA**: Menyimpan states, transition table (deterministik), start state, dan accept states
- **Fungsi konstruksi NFA dari regex**: Menggunakan algoritma Thompson secara rekursif
- **Fungsi konversi NFA ke DFA**: Mengimplementasikan subset construction algorithm
- **Fungsi simulasi DFA**: Mengikuti transisi berdasarkan input symbol, menerima jika berakhir di accept state

5. **Program recognizer untuk token-token:**

   Implementasi menggunakan DFA atau state machine:

   - **Identifier**: State machine yang mulai dengan huruf/underscore, kemudian menerima huruf/digit/underscore
   - **Integer**: State machine yang menerima satu atau lebih digit
   - **Float**: State machine yang menerima digit, titik desimal, kemudian digit
   - **Operator**: State machine yang mengenali operator single-character atau multi-character (==, !=)

6. **Mengapa DFA lebih efisien untuk simulasi dibanding NFA:**

   **DFA:**

   - Kompleksitas waktu: O(n) dimana n adalah panjang input string
   - Setiap state memiliki tepat satu transisi untuk setiap input symbol
   - Tidak memerlukan backtracking atau multiple states tracking
   - Implementasi sederhana: hanya perlu mengikuti transisi berdasarkan input symbol saat ini

   **NFA:**

   - Kompleksitas waktu: O(n * m) atau lebih buruk, dimana n adalah panjang input dan m adalah jumlah states
   - Memerlukan backtracking atau simulasi multiple states secara bersamaan
   - Perlu menangani $\epsilon$-transitions yang tidak mengonsumsi input
   - Implementasi lebih kompleks: perlu mempertahankan set of states yang mungkin

**Kesimpulan**: DFA lebih efisien untuk simulasi karena deterministik dan tidak memerlukan backtracking, meskipun konstruksi DFA dari regex lebih kompleks.

7. **Implementasi algoritma minimisasi DFA:**

   Algoritma minimisasi DFA (menggunakan partition refinement):

   (a) Partisi awal: Pisahkan accept states dan non-accept states

   (b) Untuk setiap partisi:

   - Cek apakah states dalam partisi memiliki transisi ke partisi yang berbeda untuk setiap input symbol
   - Jika ya, pisahkan menjadi partisi baru

   (c) Ulangi sampai tidak ada partisi yang dapat dipecah lagi

   (d) Gabungkan states dalam partisi yang sama menjadi satu state

   Hasilnya adalah DFA minimal yang ekuivalen dengan DFA asli tetapi dengan jumlah states yang lebih sedikit.

### 17.2.3 Jawaban Latihan Bab 3: Implementasi Lexer

1. **Implementasi lexer dasar:**

   Struktur dasar lexer:

   - **Token Types**: Enum untuk berbagai jenis token (IDENTIFIER, INTEGER_LITERAL, OPERATOR, PUNCTUATION)
   - **Token Structure**: Struktur data yang menyimpan type, lexeme, line, dan column
   - **State Machine**: Menggunakan finite state machine untuk mengenali token
   - **Methods**: `nextToken()`, `scanIdentifier()`, `scanNumber()`, `scanOperator()`, dll.

2. **Handling comments:**

   **Single-line comments (`//`):**

   - Ketika menemukan `//`, masuk ke state IN_COMMENT_LINE
   - Baca karakter sampai menemukan newline atau EOF
   - Skip semua karakter dalam comment (tidak menghasilkan token)

   **Multi-line comments (`/* */`):**

   - Ketika menemukan `/*`, masuk ke state IN_COMMENT_BLOCK
   - Baca karakter sampai menemukan `*/`

- Error handling: Jika mencapai EOF sebelum menemukan `*/`, report error "unclosed block comment"

3. **String literals dengan escape sequences:**

   Implementasi:

   - Ketika menemukan `"`, masuk ke state IN_STRING
   - Baca karakter sampai menemukan closing `"`
   - Handle escape sequences:
     - `\n`: Newline
     - `\t`: Tab
     - `\"`: Quote literal
     - `\\`: Backslash literal
   - Error handling: Jika mencapai EOF atau newline sebelum closing quote, report error "unclosed string"

4. **Float literals:**

   Extend number scanning:

   - Setelah membaca digit, cek apakah karakter berikutnya adalah titik desimal
   - Jika ya, masuk ke state IN_FLOAT
   - Baca digit setelah titik desimal
   - Untuk scientific notation (`1.5e10`, `2.3E-5`):
     - Setelah membaca float, cek apakah ada `e` atau `E`
     - Baca optional + atau −
     - Baca eksponen (digit)

5. **Unit testing:**

   Test cases yang harus dicakup:

   - **Valid tokens**: Semua kategori token (identifiers, numbers, operators, punctuation)
   - **Edge cases**:
     - Unclosed strings
     - Unclosed block comments
     - Invalid characters
     - Empty input

  - Whitespace-only input

- **Position tracking**: Verifikasi bahwa line dan column dihitung dengan benar untuk error reporting

6. **Error recovery:**

   Strategi error recovery:

   - **Skip invalid characters**: Ketika menemukan karakter yang tidak valid, skip karakter tersebut dan lanjutkan scanning
   - **Report multiple errors**: Kumpulkan semua error dalam satu pass, jangan berhenti pada error pertama
   - **Error messages yang informatif**: Sertakan line dan column number untuk setiap error

7. **Performance analysis:**

   Optimasi yang dapat dilakukan:

   - **Lookahead buffer**: Gunakan buffer untuk mengurangi jumlah system calls
   - **String interning**: Simpan lexeme sebagai interned strings untuk mengurangi memory usage
   - **Table-driven approach**: Gunakan lookup table untuk karakter classification
   - **Comparison dengan generator**: Benchmark dengan Flex-generated lexer untuk melihat perbedaan performa

### 17.2.4 Jawaban Latihan Bab 4: Lexer Generator

1. **Flex specification file:**

   Contoh specification untuk token-token yang diminta:

```
%{
#include "parser.tab.h"
%}

%%
"for"       { return FOR; }
"break"     { return BREAK; }
"continue"  { return CONTINUE; }

\"([^"\\]|\\.)*\"  {
    // String literal dengan escape sequences
    yylval.string = processString(yytext);
    return STRING_LITERAL;
```

```
}

'([^'\\]|\\.)'  {
    // Character literal
    yylval.char = processChar(yytext);
    return CHAR_LITERAL;
}

"++"        { return PLUS_PLUS; }
"--"        { return MINUS_MINUS; }
"+="        { return PLUS_EQ; }
"-="        { return MINUS_EQ; }
%%
```

2. **Implementasi dengan re2c:**

   Perbandingan kompleksitas:

   - **Flex**: Specification lebih declarative, menggunakan regular expressions
   - **re2c**: Specification lebih embedded dalam kode C/C++, menggunakan state machine yang lebih eksplisit
   - **Kompleksitas**: re2c biasanya menghasilkan kode yang lebih efisien tetapi specification lebih verbose

3. **Kapan menggunakan hand-written vs generator:**

   **Hand-written lexer cocok untuk:**

   - Error messages yang sangat informatif dan customizable
   - Kasus edge yang kompleks yang sulit diekspresikan dalam regex
   - Kontrol penuh atas implementasi
   - Project kecil yang tidak memerlukan generator overhead

   **Generator cocok untuk:**

   - Development yang cepat
   - Grammar yang well-defined dan standard
   - Project yang memerlukan proven algorithms
   - Ketika ingin fokus pada logic bukan implementasi detail

4. **Integrasi Flex dengan Bison:**

   Langkah-langkah:

(a) Buat Flex specification file (`lexer.l`) yang meng-include header dari Bison

(b) Buat Bison specification file (`parser.y`) yang mendefinisikan tokens

(c) Compile: `flex lexer.l`, `bison -d parser.y`

(d) Link: `gcc lex.yy.c parser.tab.c -o program`

Interface antara Flex dan Bison:

- Bison menghasilkan `parser.tab.h` dengan definisi token
- Flex menggunakan token definitions dari header tersebut
- Flex mengisi `yylval` dengan semantic value sebelum return token
- Parser memanggil `yylex()` untuk mendapatkan token berikutnya

5. **Performance benchmark:**

   Metodologi:

   - Siapkan input file besar (10MB)
   - Ukur waktu eksekusi untuk setiap implementasi
   - Ukur memory usage
   - Bandingkan hasil

     Hasil yang diharapkan:

     – Flex-generated lexer biasanya lebih cepat karena menggunakan optimized DFA

     – re2c-generated lexer juga sangat efisien

     – Hand-written lexer mungkin lebih lambat tetapi lebih fleksibel

6. **Longest match dan rule priority dalam Flex:**

   **Longest match**:

   - Flex selalu mencocokkan pattern terpanjang yang mungkin
   - Contoh: Input `"if"` akan cocok dengan keyword `"if"` bukan identifier `"i"` diikuti `"f"`

   **Rule priority**:

   - Jika beberapa pattern cocok dengan panjang yang sama, Flex menggunakan rule yang muncul pertama dalam specification
   - Contoh: Jika keyword `"if"` didefinisikan sebelum identifier pattern, maka `"if"` akan dikenali sebagai keyword
   - Urutan rule penting: keyword harus didefinisikan sebelum identifier pattern

### 17.2.5 Jawaban Latihan Bab 5: Context-Free Grammar dan Parsing

1. **Grammar dalam BNF:**

   **Ekspresi boolean:**

   ```
   <boolean_expr> ::= <boolean_expr> AND <boolean_term>
                    | <boolean_expr> OR <boolean_term>
                    | <boolean_term>

   <boolean_term> ::= NOT <boolean_term>
                   | <boolean_factor>

   <boolean_factor> ::= TRUE | FALSE | ( <boolean_expr> )
   ```

   **Assignment statement:**

   ```
   <assignment> ::= <identifier> = <expression> ;
   ```

   **For loop:**

   ```
   <for_loop> ::= for ( <init> ; <condition> ;
             <update> ) <statement>

   <init> ::= <declaration> | <expression> | epsilon
   <condition> ::= <expression>
   <update> ::= <expression>
   ```

2. **Konversi ke EBNF:**

   Grammar asli:

   ```
   <list> ::= <item> | <list> , <item>
   <item> ::= <number> | <string>
   ```

   Dalam EBNF:

   ```
   list = item { "," item }
   item = number | string
   ```

   Penjelasan: EBNF menggunakan `{ }` untuk repetition (nol atau lebih), sehingga `<list> , <item>` menjadi `{ "," item }`.

3. **Leftmost dan rightmost derivation untuk `(2 + 3) * 4`:**

   Grammar:

```
E → E + T | E - T | T
T → T * F | T / F | F
F → ( E ) | number
```

**Leftmost derivation:**

$$
\begin{aligned}
E &\Rightarrow T \\
&\Rightarrow T * F \\
&\Rightarrow F * F \\
&\Rightarrow (E) * F \\
&\Rightarrow (E + T) * F \\
&\Rightarrow (T + T) * F \\
&\Rightarrow (F + T) * F \\
&\Rightarrow (2 + T) * F \\
&\Rightarrow (2 + F) * F \\
&\Rightarrow (2 + 3) * F \\
&\Rightarrow (2 + 3) * 4
\end{aligned}
$$

**Rightmost derivation:**

$$
\begin{aligned}
E &\Rightarrow T \\
&\Rightarrow T * F \\
&\Rightarrow T * 4 \\
&\Rightarrow F * 4 \\
&\Rightarrow (E) * 4 \\
&\Rightarrow (E + T) * 4 \\
&\Rightarrow (E + F) * 4 \\
&\Rightarrow (E + 3) * 4 \\
&\Rightarrow (T + 3) * 4 \\
&\Rightarrow (F + 3) * 4 \\
&\Rightarrow (2 + 3) * 4
\end{aligned}
$$

4. **Parse tree untuk ekspresi:**

   Parse tree menunjukkan struktur hierarkis ekspresi berdasarkan grammar. Untuk `1 + 2 * 3`, parse tree menunjukkan bahwa `*` memiliki precedence lebih tinggi daripada `+`, sehingga `2 * 3` dievaluasi terlebih dahulu.

5. **Identifikasi ambiguity:**

   Grammar:

   ```
   S → a S a | b S b | a | b | epsilon
   ```

   Grammar ini menghasilkan bahasa palindrome dengan panjang ganjil. Contoh string yang dapat di-parse dengan lebih dari satu cara: `aba` dapat dihasilkan dengan:

   - `S → a S a → a b a` (menggunakan `S → b`)
   - Atau dengan cara lain tergantung pada urutan aplikasi rules

6. **Eliminasi left recursion:**

   Grammar asli:

   ```
   A → A + B | A - B | B
   B → B * C | B / C | C
   C → ( A ) | number
   ```

   Setelah eliminasi left recursion:

   ```
   A → B A'
   A' → + B A' | - B A' | epsilon

   B → C B'
   B' → * C B' | / C B' | epsilon

   C → ( A ) | number
   ```

7. **Left factoring:**

   Grammar asli:

   ```
   S → if E then S else S
     | if E then S
     | while E do S
     | id = E
   ```

   Setelah left factoring:

   ```
   S → if E then S S'
     | while E do S
     | id = E

   S' → else S | epsilon
   ```

8. **Grammar untuk bahasa sederhana:**

   Dalam EBNF:

```
program = { declaration | statement }

declaration = type identifier ";"

statement = assignment
          | if_statement
          | while_statement
          | block

assignment = identifier "=" expression ";"

if_statement = "if" "(" expression ")" statement
              [ "else" statement ]

while_statement = "while" "(" expression ")" statement

block = "{" { statement } "}"

expression = term { ("+" | "-") term }
term = factor { ("*" | "/") factor }
factor = identifier | number | "(" expression ")"

type = "int" | "float"
identifier = letter { letter | digit }
number = digit { digit }
```

   Contoh program valid:

```
int x;
x = 5;
if (x > 0) {
    int y;
    y = 10;
    while (y > 0) {
        y = y - 1;
    }
}
```

### 17.2.6 Jawaban Latihan Bab 6: Top-Down Parsing dan Recursive Descent

1. **Implementasi recursive descent parser:**

**Langkah 1: Eliminasi left recursion**

Grammar asli memiliki left recursion pada E dan T. Setelah eliminasi:

```
S → if E then S else S | id := E | while E do S
E → T E'
E' → + T E' | epsilon
T → F T'
T' → * F T' | epsilon
F → ( E ) | id | num
```

**Langkah 2: Implementasi dalam C++**

Setiap nonterminal menjadi fungsi yang sesuai:

- `parseS()` untuk S
- `parseE()`, `parseEPrime()` untuk E dan E'
- `parseT()`, `parseTPrime()` untuk T dan T'
- `parseF()` untuk F

2. **Modifikasi untuk operator unary minus:**

   Tambahkan production untuk unary minus di level factor:

   ```
   F → ( E ) | id | num | - F
   ```

   Implementasi dalam `parseF()`:

   - Cek apakah token berikutnya adalah −
   - Jika ya, consume token dan parse F secara rekursif
   - Return negated value

3. **Error recovery:**

   Strategi:

   - **Synchronization points**: Token-token yang dapat digunakan untuk recovery (misalnya `;`, `}`, keywords)
   - **Panic mode**: Skip token sampai menemukan synchronization point
   - **Error reporting**: Kumpulkan semua error, jangan berhenti pada error pertama
   - **Insertion/deletion recovery**: Coba insert token yang diharapkan atau skip token yang tidak diharapkan

4. **Parser untuk ekspresi boolean:**

   Grammar dengan precedence:

```
E → T E'          (OR level, lowest precedence)
E' → OR T E' | epsilon

T → F T'          (AND level)
T' → AND F T' | epsilon

F → NOT F | ( E ) | comparison
```

Precedence: NOT > AND > OR

5. **Perbandingan recursive descent vs table-driven LL:**

   **Recursive Descent:**

   - **Keuntungan**: Kode lebih readable, error recovery lebih baik, mudah di-debug
   - **Kerugian**: Hanya cocok untuk LL(1) grammar, lebih banyak kode manual

   **Table-driven LL:**

   - **Keuntungan**: Grammar dapat diubah tanpa mengubah kode parser, lebih compact
   - **Kerugian**: Error messages kurang informatif, lebih sulit di-debug, memerlukan table construction

6. **Parser untuk right-associative operator:**

   Untuk operator assignment = yang right-associative:

   ```
   assignment → identifier = assignment | expression
   ```

   Implementasi: Parse identifier dan =, kemudian parse assignment secara rekursif (right-associative).

### 17.2.7 Jawaban Latihan Bab 7: Bottom-Up Parsing dan Parser Generator

1. **Perbedaan top-down dan bottom-up parsing:**

   **Top-down parsing:**

   - Mulai dari start symbol, turun ke terminal
   - Menggunakan leftmost derivation
   - Contoh: Recursive descent, LL parsers

- Cocok untuk: Grammar yang tidak memiliki left recursion, grammar yang lebih sederhana

**Bottom-up parsing:**

- Mulai dari terminal, naik ke start symbol
- Menggunakan rightmost derivation dalam reverse
- Contoh: LR, LALR, SLR parsers
- Cocok untuk: Grammar yang lebih kompleks, grammar dengan left recursion

2. **Shift-reduce parsing manual untuk `id + id * id`:**

   Grammar:

   ```
   E → E + T | T
   T → T * F | F
   F → ( E ) | id
   ```

| Stack | Input | Action | Production |
|---|---|---|---|
| $ | id + id * id $ | Shift | |
| $ id | + id * id $ | Reduce | F → id |
| $ F | + id * id $ | Reduce | T → F |
| $ T | + id * id $ | Reduce | E → T |
| $ E | + id * id $ | Shift | |
| $ E + | id * id $ | Shift | |
| $ E + id | * id $ | Reduce | F → id |
| $ E + F | * id $ | Reduce | T → F |
| $ E + T | * id $ | Shift | |
| $ E + T * | id $ | Shift | |
| $ E + T * id | $ | Reduce | F → id |
| $ E + T * F | $ | Reduce | T → T * F |
| $ E + T | $ | Reduce | E → E + T |
| $ E | $ | Accept | |

3. **Perbedaan SLR(1), CLR(1), dan LALR(1):**

   **SLR(1) - Simple LR:**

   - Menggunakan LR(0) item sets
   - Reduce hanya jika lookahead dalam Follow set
   - Tabel lebih kecil, tetapi kurang powerful

   **CLR(1) - Canonical LR:**

- Menggunakan LR(1) items dengan lookahead spesifik
- Paling powerful, dapat menangani lebih banyak grammar
- Tabel sangat besar

**LALR(1) - Look-Ahead LR:**

- Merge states dari CLR(1) yang memiliki LR(0) core sama
- Kompromi praktis: tabel lebih kecil dari CLR(1), lebih powerful dari SLR(1)
- Digunakan oleh Bison dan Yacc

**Mengapa LALR(1) populer:**

- Balance yang baik antara power dan table size
- Dapat menangani sebagian besar grammar praktis
- Tabel cukup kecil untuk efisiensi

4. **File Bison untuk ekspresi aritmatika:**

```
%{
#include <stdio.h>
%}

%token NUMBER
%left '+' '-'
%left '*' '/'

%%
expr: expr '+' expr { printf("+ "); }
   | expr '-' expr { printf("- "); }
   | expr '*' expr { printf("* "); }
   | expr '/' expr { printf("/ "); }
   | '(' expr ')' { /* no action */ }
   | NUMBER { printf("%s ", $1); }
%%
```

5. **Integrasi Flex dan Bison untuk calculator:**

   Langkah-langkah:

   (a) Buat Flex lexer yang menghasilkan NUMBER token

   (b) Buat Bison parser dengan semantic actions untuk evaluasi

   (c) Link kedua file dan compile

   Semantic actions mengevaluasi ekspresi dan mencetak hasil.

6. **Error recovery dalam Bison:**

   Menggunakan token `error`:

   ```
   expr: expr '+' expr
      | error { yyerror("Syntax error in expression"); }
      | error ';' { /* recover by skipping to semicolon */ }
   ```

   Strategi: Skip token sampai menemukan synchronization point (misalnya `;`).

7. **Perbandingan performa:**

   **Recursive descent:**

   - Lebih cepat untuk grammar sederhana
   - Overhead function calls

   **Bison-generated:**

   - Table lookup overhead
   - Lebih efisien untuk grammar kompleks
   - Dapat dioptimasi dengan table compression

### 17.2.8 Jawaban Latihan Bab 8: Semantic Actions dan AST Construction

1. **Parser untuk ekspresi boolean:**

   Grammar dengan semantic actions untuk membangun AST:

   ```
   %union {
       ASTNode* node;
   }

   %type <node> expr

   %%
   expr: expr AND expr { $$ = createBinaryNode(AND, $1, $3); }
      | expr OR expr  { $$ = createBinaryNode(OR, $1, $3); }
      | NOT expr      { $$ = createUnaryNode(NOT, $2); }
      | comparison    { $$ = $1; }

   comparison: expr '<' expr
       { $$ = createBinaryNode(LT, $1, $3); }
            | expr '>' expr
       { $$ = createBinaryNode(GT, $1, $3); }
            | expr EQ expr
   ```

```
    { $$ = createBinaryNode(EQ, $1, $3); }
         | expr NE expr
    { $$ = createBinaryNode(NE, $1, $3); }
%%
```

2. **Modifikasi calculator:**

   Fitur tambahan:

   - **Variabel**: Symbol table untuk menyimpan nilai variabel
   - **Fungsi matematika**: Built-in functions (sin, cos, sqrt)
   - **Error handling**: Type checking, undefined variable detection

3. **Parser untuk konfigurasi file:**

   Grammar:

```
config: { /* init */ } config_items

config_items: config_items config_item | config_item

config_item: key '=' value '\n'
           | section

section: IDENTIFIER '{' config_items '}'

key: IDENTIFIER
value: STRING | NUMBER
```

   Semantic actions membangun struktur data konfigurasi.

4. **Semantic actions untuk AST lengkap:**

   Fungsi-fungsi yang diperlukan:

   - **Print AST**: Traverse AST dan print dalam format tree (pre-order atau in-order)
   - **Evaluate AST**: Jika semua nilai diketahui, evaluasi ekspresi
   - **Optimize AST**: Constant folding - ganti ekspresi konstanta dengan hasilnya

5. **Perbedaan LALR(1), GLR parsing:**

   **LALR(1) - Default Bison:**

   - Deterministic parsing
   - Satu parse tree untuk setiap input

- Tidak dapat menangani ambiguous grammar
- Lebih cepat dan efisien

**GLR - Generalized LR:**

- Dapat menangani ambiguous grammar
- Menjaga multiple parse trees aktif secara bersamaan
- Merge stack prefixes yang mungkin
- Lebih lambat tetapi lebih powerful

**Kapan digunakan:**

- LALR(1): Grammar yang unambiguous, sebagian besar kasus praktis
- GLR: Grammar yang ambiguous, natural language processing, extensible syntax

### 17.2.9 Jawaban Latihan Bab 9: Abstract Syntax Tree (AST)

1. **AST untuk program:**

   Program:

   ```
   int x = 10;
   int y = 20;
   int z = x + y * 2;
   ```

   Struktur AST:

   - Root: Program node dengan tiga children (declarations)
   - Setiap declaration: VarDecl node dengan type, identifier, dan optional initializer
   - Ekspresi `x + y * 2`: BinaryExpr node dengan operator `+`, left adalah Identifier `x`, right adalah BinaryExpr dengan operator `*`

2. **Implementasi AST node:**

   **For loop statement:**

   - Node: ForStmt dengan fields: init (optional), condition, update (optional), body

   **Array access expression:**

   - Node: ArrayAccess dengan fields: array (expression), index (expression)

   **String literal:**

   - Node: StringLiteral dengan field: value (string)

**Boolean literal:**

- Node: BoolLiteral dengan field: value (bool)

3. **Visitor untuk menghitung jumlah node:**

   Implementasi menggunakan visitor pattern:

   - Visitor memiliki counter
   - Setiap node memanggil `accept(visitor)`
   - Visitor increment counter dan traverse children
   - Hasil: total jumlah node dalam AST

4. **Visitor untuk constant folding:**

   Algoritma:

   - Traverse AST secara post-order
   - Jika menemukan BinaryExpr dengan kedua operan adalah konstanta:
     - Evaluasi ekspresi (misalnya 3 + 5 = 8)
     - Ganti BinaryExpr dengan Literal node yang berisi hasil
   - Ulangi sampai tidak ada perubahan

5. **Modifikasi PrettyPrinter:**

   Improvements:

   - Preserve whitespace dan formatting sebisa mungkin
   - Handle operator precedence dengan parentheses yang tepat
   - Format indentation yang konsisten
   - Preserve comments jika ada

6. **AST visualizer dengan Graphviz DOT:**

   Implementasi:

   - Traverse AST dan generate DOT format
   - Setiap node menjadi node dalam graph
   - Edges menunjukkan parent-child relationships
   - Output dapat di-render dengan `dot` command

7. **Mengapa AST lebih disukai daripada parse tree:**

   Alasan:

- **Lebih kompak**: Menghilangkan detail sintaksis yang tidak relevan
- **Fokus semantik**: Hanya menyertakan informasi yang diperlukan untuk fase selanjutnya
- **Mudah di-manipulasi**: Struktur yang lebih sederhana memudahkan transformasi
- **Efisien**: Ukuran lebih kecil, traversal lebih cepat

8. **Traversal orders untuk `(a + b) * (c - d)`:**

   **Pre-order**: `* + a b - c d` (operator sebelum operan)

   **In-order**: `(a + b) * (c - d)` (mirip dengan ekspresi asli)

   **Post-order**: `a b + c d - *` (operator setelah operan, cocok untuk evaluasi)

### 17.2.10 Jawaban Latihan Bab 10: Symbol Table dan Scope Management

1. **Implementasi symbol table lengkap:**

   Fitur-fitur:

   - **Insert**: Menambahkan symbol ke scope saat ini
   - **Lookup**: Mencari symbol mulai dari scope saat ini ke parent scopes
   - **Delete**: Menghapus symbol dari scope (biasanya saat exit scope)
   - **Nested scopes**: Stack of hash tables, setiap scope memiliki parent pointer
   - **Shadowing detection**: Warning ketika identifier shadow outer declaration
   - **Visualisasi**: Print symbol table dengan indentasi untuk menunjukkan hierarchy

2. **Test cases:**

   **Nested scopes dengan shadowing:**

```
int x = 1;
{
    int x = 2;  // Shadows outer x
    // x refers to inner x (2)
}
// x refers to outer x (1)
```

   **Duplicate declaration:**

```
int x = 1;
int x = 2;  // Error: duplicate declaration
```

**Undeclared identifier:**

```
x = 10;  // Error: x not declared
```

**Function parameters:**

```
void func(int x) {
    // x is in function scope
    int y = x + 1;  // OK: x is parameter
}
```

3. **Visualisasi symbol table untuk program:**

   Program:

```
int a = 1;        // Global scope
int b = 2;

void func1(int x) {  // Function scope
    int a = 10;      // Shadows global a
    int c = 20;

    {                // Block scope
        int b = 30;  // Shadows func1 b
        int d = 40;
    }
}

void func2() {       // Function scope
    int x = 100;
    int y = 200;
}
```

   Symbol table structure:

```
Level 0 (Global):
  a -> int (line 1)
  b -> int (line 2)
  func1 -> function
  func2 -> function

Level 1 (func1):
  x -> int (parameter)
  a -> int (line 4) [shadows global a]
  c -> int (line 5)

Level 2 (block in func1):
```

```
    b -> int (line 8) [shadows func1 b]
    d -> int (line 9)

  Level 1 (func2):
    x -> int (line 14)
    y -> int (line 15)
```

4. **Fitur tambahan:**

   **Tracking usages:**

   - Simpan list semua lokasi dimana identifier digunakan
   - Berguna untuk dead code elimination dan optimasi

   **Forward declaration:**

   - Simpan declaration tanpa body
   - Resolve saat body ditemukan

   **Scope numbering:**

   - Assign unique number untuk setiap scope
   - Memudahkan debugging dan visualisasi

5. **Perbandingan implementasi:**

   **Stack of hash tables:**

   - **Kelebihan**: Simple, efficient lookup dalam scope saat ini
   - **Kekurangan**: Lookup di parent scope memerlukan traversal

   **Single hash table dengan per-name stacks:**

   - **Kelebihan**: Fast lookup untuk name tertentu
   - **Kekurangan**: Lebih kompleks, perlu manage multiple stacks

### 17.2.11 Jawaban Latihan Bab 11: Type Checking dan Semantic Analysis

1. **Perbedaan static dan dynamic type checking:**

   **Static type checking:**

   - Type checking dilakukan pada waktu kompilasi
   - Error ditemukan sebelum program dijalankan

- Contoh bahasa: C, C++, Java, Rust
- Keuntungan: Early error detection, better performance

**Dynamic type checking:**

- Type checking dilakukan pada waktu runtime
- Error ditemukan saat program dijalankan
- Contoh bahasa: Python, JavaScript, Ruby
- Keuntungan: Lebih fleksibel, rapid prototyping

2. **Type checker untuk ekspresi aritmatika:**

   Implementasi:

   - **Operasi +, -, *, /**: Cek bahwa kedua operan adalah int atau float
   - **Type promotion**: Jika salah satu operan float, hasilnya float
   - **Operasi perbandingan**: Cek kompatibilitas tipe, hasilnya boolean

3. **Fungsi type compatibility:**

   Aturan:

   - Exact match selalu kompatibel
   - int dapat di-assign ke float (implicit conversion)
   - float tidak dapat di-assign ke int tanpa explicit cast

4. **Semantic error detection:**

   **Undeclared variable:**

   - Saat menemukan identifier, cek di symbol table
   - Jika tidak ditemukan, report error

   **Type mismatch pada assignment:**

   - Cek tipe left-hand side dan right-hand side
   - Gunakan fungsi compatibility check

   **Wrong number of arguments:**

   - Cek jumlah argument dengan jumlah parameter fungsi
   - Cek tipe setiap argument dengan parameter yang sesuai

5. **Test cases:**

   **Valid expressions:**

- `int x = 42;`
- `float y = 3.14;`
- `int z = x + 10;`
- `bool b = x > 0;`

**Invalid expressions:**

- `int x = "string";` // Type mismatch
- `x = 10;` // Undeclared variable
- `add(5);` // Wrong number of arguments

6. **Mengapa annotated AST diperlukan:**

   Alasan:

   - **Type information**: Setiap node menyimpan tipe hasil ekspresi
   - **Code generation**: Informasi tipe digunakan untuk memilih instruksi yang tepat
   - **Optimization**: Type information memungkinkan optimasi yang lebih baik

7. **Type checking untuk if dan while:**

   Implementasi:

   - Cek bahwa kondisi adalah boolean
   - Jika bukan boolean, report type error

8. **Perbandingan nominal dan structural typing:**

   **Nominal typing:**

   - Tipe kompatibel jika memiliki nama yang sama
   - Contoh: Java, C++
   - Keuntungan: Explicit, clear contracts

   **Structural typing:**

   - Tipe kompatibel jika memiliki struktur yang sama
   - Contoh: TypeScript, Go interfaces
   - Keuntungan: Lebih fleksibel, duck typing

### 17.2.12 Jawaban Latihan Bab 12: Intermediate Code Generation

1. **TAC untuk ekspresi:**

   **`a = b + c * d - e:`**

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

   **`x = (a + b) * (c - d):`**

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

   **`y = -a + b * -c:`**

   ```
   t1 = -a
   t2 = -c
   t3 = b * t2
   t4 = t1 + t3
   y = t4
   ```

2. **Generator TAC untuk operasi khusus:**

   **Unary operations:**

   - Negation: `t1 = -operand`
   - Logical NOT: `t1 = !operand`

   **Array access:**

   - `t1 = index * element_size`
   - `t2 = base + t1`
   - `t3 = mem[t2]`

   **Member access:**

   - `t1 = object + offset`
   - `t2 = mem[t1]`

3. **TAC untuk for loop:**

   Program:

```
int i, sum;
sum = 0;
for (i = 1; i <= 10; i = i + 1) {
    sum = sum + i;
}
```

TAC:

```
sum = 0
i = 1
L1: t1 = i <= 10
    jmpf t1, L2
    t2 = sum + i
    sum = t2
    t3 = i + 1
    i = t3
    jmp L1
L2:
```

4. **Common subexpression elimination:**

   Kode asli:

```
x = a + b * c
y = a + b * c
z = (a + b) * c
w = (a + b) * c
```

   Setelah CSE:

```
t1 = b * c
t2 = a + b
x = a + t1
y = x        // atau y = a + t1
z = t2 * c
w = z        // atau w = t2 * c
```

5. **Perbedaan TAC dan quadruples:**

   **Three-address code:**

   - Format text: `t1 = a + b`
   - Mudah dibaca manusia
   - Sequential representation

**Quadruples:**

- Format struktural: `(op: +, arg1: a, arg2: b, result: t1)`
- Mudah di-reorder untuk optimasi
- Lebih mudah untuk manipulasi programmatic

**Quadruples lebih menguntungkan untuk:**

- Optimasi yang memerlukan reordering
- Common subexpression elimination
- Code generation yang memerlukan manipulasi struktural

6. **Generator TAC untuk switch statement:**

   Strategi:

   - Generate comparison untuk setiap case
   - Gunakan jump table jika case values consecutive
   - Atau gunakan chain of if-else dengan labels

7. **Constant folding pada TAC generator:**

   Implementasi:

   - Saat generate TAC, cek apakah kedua operan adalah konstanta
   - Jika ya, evaluasi langsung dan generate assignment konstanta hasil
   - Contoh: `x = 3 + 5` langsung menjadi `x = 8`

8. **Perbandingan pendekatan TAC generation:**

   **Top-down (generate sambil traverse):**

   - Generate TAC langsung saat traverse AST
   - Keuntungan: Simple, langsung menghasilkan output
   - Kerugian: Sulit untuk optimasi lintas ekspresi

   **Bottom-up (build IR structure dulu):**

   - Build struktur IR lengkap dulu, kemudian generate TAC
   - Keuntungan: Memungkinkan optimasi sebelum code generation
   - Kerugian: Lebih kompleks, memerlukan lebih banyak memory

### 17.2.13 Jawaban Latihan Bab 13: Runtime Environment dan Memory Management

1. **Perbedaan static, stack, dan heap allocation:**

   **Static allocation:**

   - Alokasi pada waktu kompilasi
   - Lifetime: Selama program berjalan
   - Contoh: Global variables, static variables
   - Lokasi: Data segment

   **Stack allocation:**

   - Alokasi pada waktu runtime (function call)
   - Lifetime: Selama fungsi aktif
   - Contoh: Local variables, function parameters
   - Lokasi: Stack segment
   - Management: Automatic (dikelola oleh compiler)

   **Heap allocation:**

   - Alokasi dinamis pada waktu runtime
   - Lifetime: Sampai secara eksplisit di-deallocate
   - Contoh: Dynamic arrays, objects dengan lifetime fleksibel
   - Lokasi: Heap segment
   - Management: Manual (programmer) atau garbage collection

2. **Diagram activation records untuk `factorial(3)`:**

   Stack saat `factorial(3)` sedang dieksekusi (recursive call):

```
High Address
+---------------------+
| main() AR           |
|   result            |
+---------------------+
| factorial(3) AR     | <- Current (top)
|   n = 3             |
|   return address    |
|   control link      | -> main AR
+---------------------+
| factorial(2) AR     |
```

```
|   n = 2            |
|   return address   |
|   control link     | -> factorial(3) AR
+--------------------+
| factorial(1) AR    |
|   n = 1            |
|   return address   |
|   control link     | -> factorial(2) AR
Low Address
```

3. **Runtime stack simulator:**

   Implementasi:

   - **Stack structure**: Linked list atau array of activation records
   - **Push frame**: Saat function call, buat AR baru dan push ke stack
   - **Pop frame**: Saat function return, pop AR dari stack
   - **Parameter passing**: Simpan parameters dalam AR
   - **Local variables**: Simpan dalam AR
   - **Return value**: Simpan dalam AR sebelum pop

4. **Calling sequence untuk fungsi dengan 3 parameters:**

   Assembly code (simplified):

```
; Caller sequence
push param3      ; Push parameter 3
push param2      ; Push parameter 2
push param1      ; Push parameter 1
call function    ; Call function
add esp, 12      ; Clean up parameters (3 * 4 bytes)

; Callee sequence (function prologue)
push ebp         ; Save frame pointer
mov ebp, esp     ; Set new frame pointer
sub esp, N       ; Allocate space for locals

; Function body
...

; Return sequence (function epilogue)
mov eax, return_value  ; Set return value
mov esp, ebp     ; Restore stack pointer
pop ebp          ; Restore frame pointer
ret              ; Return
```

5. **Perbandingan manual memory management vs garbage collection:**

   **Manual memory management:**

   - **Keuntungan**: Kontrol penuh, predictable performance, no GC overhead
   - **Kekurangan**: Mudah terjadi memory leak, use-after-free bugs, lebih kompleks
   - Contoh: C, C++

   **Garbage collection:**

   - **Keuntungan**: Tidak ada memory leak, lebih aman, programmer tidak perlu manage memory
   - **Kekurangan**: GC pause, overhead, unpredictable timing
   - Contoh: Java, Python, Go

6. **Implementasi mark-and-sweep garbage collector:**

   Algoritma:

   (a) **Mark phase**: Traverse dari root set (stack, global variables, registers), mark semua reachable objects

   (b) **Sweep phase**: Scan semua objects, free yang tidak ter-mark

   (c) **Handle cyclic references**: Mark phase secara otomatis menangani cycles karena menggunakan graph traversal

7. **Memory layout untuk program:**

```
int global;           // Static/Global segment
static int static_var; // Static/Global segment

void func() {
    int local;              // Stack segment
    static int static_local; // Static (meskipun dalam fungsi)
    int* ptr = new int;     // Heap segment (yang ditunjuk ptr)
    // ptr sendiri ada di stack
}
```

8. **Recursive call yang sangat dalam:**

   Masalah yang mungkin terjadi:

   - **Stack overflow**: Stack segment terbatas, recursive call yang sangat dalam dapat melebihi batas
   - **Memory exhaustion**: Setiap call memerlukan activation record

- **Performance degradation**: Overhead function calls

Solusi:

- Tail call optimization (jika compiler mendukung)
- Iterative implementation
- Increase stack size (jika memungkinkan)

### 17.2.14 Jawaban Latihan Bab 14: Code Generation

1. **Code generator untuk operasi aritmatika RISC-V:**

   Implementasi:

   - **Addition**: `ADD rd, rs1, rs2`
   - **Subtraction**: `SUB rd, rs1, rs2`
   - **Multiplication**: `MUL rd, rs1, rs2`
   - **Division**: `DIV rd, rs1, rs2`

2. **Code generator untuk assignment:**

   **Constant ke variabel:**

   ```
   LI t1, 42        ; Load immediate
   SW t1, x         ; Store to variable
   ```

   **Variabel ke variabel:**

   ```
   LW t1, x         ; Load from x
   SW t1, y         ; Store to y
   ```

   **Ekspresi ke variabel:**

   ```
   LW t1, a
   LW t2, b
   ADD t3, t1, t2
   SW t3, result
   ```

3. **Local register allocation:**

   Algoritma sederhana:

   - Scan basic block, assign register untuk setiap variabel aktif
   - Jika register penuh, spill variabel yang paling lama tidak digunakan ke memory

- Load dari memory saat diperlukan kembali

4. **Code generator untuk function call:**

   RISC-V calling convention:

   - Parameters: a0-a7 (first 8 parameters), kemudian stack
   - Return value: a0
   - Caller-saved: t0-t6, a0-a7
   - Callee-saved: s0-s11

   Code generation:

```
; Caller
ADDI sp, sp, -16    ; Allocate stack space
SW ra, 12(sp)       ; Save return address
MV a0, param1       ; Pass parameter 1
MV a1, param2       ; Pass parameter 2
CALL function       ; Call function
LW ra, 12(sp)       ; Restore return address
ADDI sp, sp, 16     ; Deallocate
MV result, a0       ; Get return value

; Callee
function:
    ADDI sp, sp, -16  ; Allocate frame
    SW ra, 12(sp)     ; Save return address
    SW s0, 8(sp)      ; Save callee-saved
    ADDI s0, sp, 16   ; Set frame pointer
    ; Function body
    MV a0, return_val ; Set return value
    LW s0, 8(sp)      ; Restore
    LW ra, 12(sp)     ; Restore
    ADDI sp, sp, 16   ; Deallocate
    RET               ; Return
```

5. **Test workflow lengkap:**

   Program: `a * b + c`

   **Compile**:

```
TAC:
t1 = a * b
t2 = t1 + c
```

   **Assemble**:

```
LW t0, a
LW t1, b
MUL t2, t0, t1
LW t3, c
ADD t4, t2, t3
```

**Link**: Menghasilkan executable

**Run**: Eksekusi program

6. **Perbandingan dengan dan tanpa optimasi register allocation:**
   **Tanpa optimasi**:

```
LW t0, a          ; Load a
LW t1, b          ; Load b
MUL t2, t0, t1    ; t2 = a * b
SW t2, temp       ; Spill to memory
LW t3, c          ; Load c
LW t4, temp       ; Reload from memory
ADD t5, t4, t3    ; t5 = t2 + c
```

   Instruksi: 7, Memory access: 5
   **Dengan optimasi**:

```
LW t0, a
LW t1, b
MUL t2, t0, t1    ; Keep in register
LW t3, c
ADD t4, t2, t3    ; Use register directly
```

   Instruksi: 5, Memory access: 3

   **Improvement**: Mengurangi instruksi dan memory access.

7. **Code generator untuk if-else:**

```
; Condition evaluation
LW t0, x
LI t1, 0
BGT t0, t1, then_label  ; Branch if x > 0

; Else branch
else_label:
    LI t2, 0
    SW t2, y
    J end_label
```

```
; Then branch
then_label:
    LI t2, 1
    SW t2, y

end_label:
```

8. **Code generator untuk loop:**

```
loop_start:
    ; Condition check
    LW t0, i
    LI t1, 10
    BGE t0, t1, loop_end  ; Branch if i >= 10

    ; Loop body
    LW t2, sum
    ADD t2, t2, t0
    SW t2, sum

    ; Increment
    ADDI t0, t0, 1
    SW t0, i

    J loop_start          ; Jump back

loop_end:
```

### 17.2.15 Jawaban Latihan Bab 15: Optimasi Kompilator

1. **Identifikasi basic blocks:**

   Kode:

```
t1 = a + b
t2 = c * d
if t1 > t2 goto L1
t3 = t1 - t2
goto L2
L1: t4 = t1 * t2
L2: t5 = t3 + t4
return t5
```

   Basic blocks:

- **Block 1**: `t1 = a + b`, `t2 = c * d`, `if t1 > t2 goto L1`
- **Block 2**: `t3 = t1 - t2`, `goto L2`
- **Block 3 (L1)**: `t4 = t1 * t2`
- **Block 4 (L2)**: `t5 = t3 + t4`, `return t5`

2. **Constant folding:**

   Kode asli:

```
t1 = 5 + 3
t2 = t1 * 2
t3 = 10 / 2
x = t2 + t3
```

   Setelah constant folding:

```
t1 = 8          ; 5 + 3 = 8
t2 = 16         ; 8 * 2 = 16
t3 = 5          ; 10 / 2 = 5
x = 21          ; 16 + 5 = 21
```

   Atau lebih optimal:

```
x = 21          ; Semua di-fold menjadi satu nilai
```

3. **Constant propagation dan folding:**

   Kode asli:

```
x = 10
y = 20
t1 = x + 5
t2 = y * 2
z = t1 + t2
```

   Setelah constant propagation:

```
x = 10
y = 20
t1 = 10 + 5    ; x diganti dengan 10
t2 = 20 * 2    ; y diganti dengan 20
z = t1 + t2
```

   Setelah constant folding:

```
x = 10
y = 20
t1 = 15        ; 10 + 5 = 15
t2 = 40        ; 20 * 2 = 40
z = 55         ; 15 + 40 = 55
```

4. **Dead code elimination:**

   Kode asli:

```
x = 10
y = 20
t1 = x + y
t2 = 5 + 3      // Dead: t2 tidak digunakan
t3 = t2 - 2     // Dead: t3 tidak digunakan
z = t1 * 2
return z
```

   Setelah dead code elimination:

```
x = 10
y = 20
t1 = x + y
z = t1 * 2
return z
```

5. **Implementasi optimizer sederhana:**

   Algoritma:

   (a) Constant folding: Evaluasi ekspresi konstanta

   (b) Constant propagation: Ganti variabel konstanta dengan nilainya

   (c) Dead code elimination: Hapus assignment yang tidak digunakan

   (d) Iterate sampai tidak ada perubahan

6. **Perbedaan optimasi:**

   **Local vs Global:**

   - Local: Dalam satu basic block
   - Global: Lintas basic blocks, memerlukan data-flow analysis

   **Machine-independent vs Machine-specific:**

   - Machine-independent: Tidak bergantung pada target architecture (constant folding, dead code elimination)

- Machine-specific: Bergantung pada target (register allocation, instruction selection)

**Constant folding vs Constant propagation:**

- Constant folding: Evaluasi ekspresi konstanta langsung
- Constant propagation: Ganti penggunaan variabel konstanta dengan nilainya

7. **Benchmark untuk evaluasi optimasi:**

   Metrik:

   - **Code size**: Ukuran executable sebelum dan sesudah optimasi
   - **Execution time**: Waktu eksekusi program yang dikompilasi
   - **Compilation time**: Waktu kompilasi dengan optimasi

   Hasil yang diharapkan:

   - Code size: Biasanya lebih kecil setelah optimasi
   - Execution time: Biasanya lebih cepat setelah optimasi
   - Compilation time: Biasanya lebih lama karena optimasi memerlukan analisis tambahan

### 17.2.16 Jawaban Latihan Bab 16: Evaluasi dan Project Final

1. **Prepare Presentation:**

   **Outline presentasi:**

   (a) Introduction: Overview project

   (b) Architecture: Komponen utama compiler

   (c) Implementation: Tools dan teknik yang digunakan

   (d) Demo: Live demonstration dengan test cases

   (e) Evaluation: Benchmark results dan analisis

   (f) Challenges: Masalah yang dihadapi dan solusi

   (g) Conclusion: Lessons learned dan future work

   **Demo script dengan 5 test cases:**

   - Test case 1: Simple arithmetic expressions
   - Test case 2: Control flow (if-else, loops)
   - Test case 3: Function calls

- Test case 4: Error handling
- Test case 5: Complex program

2. **Tool Evaluation:**

   **Tabel perbandingan:**

   - Hand-written lexer vs Flex: Development time, maintainability, error messages
   - Recursive descent vs Bison: Grammar support, error recovery, performance
   - Trade-offs: Flexibility vs correctness, development time vs maintainability

   **Justifikasi pilihan:**

   - Pilih tools berdasarkan requirements project
   - Pertimbangkan complexity, performance, dan maintainability

3. **Benchmarking:**

   **Test suite:**

   - Small programs (10-100 lines)
   - Medium programs (100-1000 lines)
   - Large programs (1000+ lines)

   **Metrik:**

   - Compilation time
   - Code size (executable)
   - Execution time

   **Laporan:**

   - Tabel hasil benchmark
   - Analisis perbandingan
   - Identifikasi bottleneck

4. **Documentation:**

   **README.md:**

   - Overview project
   - Build instructions
   - Usage examples

- Test instructions

**Design document:**

- Architecture overview
- Component descriptions
- Data structures
- Algorithms used

**API documentation:**

- Public interfaces
- Function signatures
- Usage examples

5. **Reflection:**

   **Refleksi pembelajaran:**

   - Pemahaman tentang compiler phases
   - Pengalaman implementasi
   - Challenges yang dihadapi
   - Lessons learned

   **3 challenges utama:**

   (a) Error recovery yang baik

   (b) Optimasi yang efektif

   (c) Integrasi antar fase

   **5 lessons learned:**

   (a) Importance of good data structures

   (b) Error messages sangat penting untuk user experience

   (c) Testing adalah kunci untuk correctness

   (d) Documentation membantu maintenance

   (e) Trade-offs selalu ada dalam engineering decisions

   **3 area untuk improvement:**

   (a) Advanced optimizations

(b) Better error recovery

(c) Support untuk lebih banyak language features

6. **Peer Review:**

   **Aspects to review:**

   - Code quality dan organization
   - Correctness dari compiler
   - Performance dan optimizations
   - Documentation quality
   - Test coverage

   **Feedback konstruktif:**

   - Identifikasi strengths
   - Suggest improvements
   - Compare different approaches

# Bab 18

## Kumpulan Quiz Pilihan Ganda

### 18.1 Soal

#### 18.1.1 Quiz Bab 1: Pengenalan Kompilator dan Fase-Fase Kompilasi

1. Apa perbedaan utama antara kompilator dan interpreter?

    a.) Kompilator mengeksekusi baris per baris; interpreter menerjemahkan seluruh program sekaligus

    b.) Kompilator menerjemahkan seluruh program sekaligus sebelum dieksekusi; interpreter menerjemahkan dan mengeksekusi baris demi baris

    c.) Kompilator menghasilkan bytecode; interpreter menghasilkan machine code

    d.) Tidak ada perbedaan yang signifikan

2. Fase kompilasi manakah yang bertanggung jawab menghapus whitespace dan komentar?

    a.) Syntax analysis

    b.) Semantic analysis

    c.) Lexical analysis

    d.) Code generation

3. Manakah yang termasuk bahasa yang menggunakan pendekatan interpreter?

    a.) C dan C++

    b.) Python dan JavaScript

    c.) Rust dan Go

    d.) Pascal dan Fortran

4. Fase manakah yang mengonversi ekspresi `a + b * c` menjadi three-address code?

a.) Lexical analysis

b.) Syntax analysis

c.) Semantic analysis

d.) Intermediate code generation

5. Program yang menerjemahkan assembly code ke machine code disebut...

a.) Kompilator

b.) Interpreter

c.) Assembler

d.) Linker

### 18.1.2 Quiz Bab 2: Regular Expression dan Finite Automata

1. Algoritma Thompson digunakan untuk...

a.) Mengkonversi NFA ke DFA

b.) Mengkonversi regular expression ke NFA

c.) Minimisasi DFA

d.) Mensimulasikan DFA

2. Mengapa DFA lebih efisien untuk simulasi dibanding NFA?

a.) DFA lebih mudah dikonstruksi

b.) DFA tidak memiliki $\epsilon$-transition; setiap state tepat satu transition per input symbol

c.) NFA tidak dapat mengenali regular language

d.) DFA menggunakan lebih sedikit memori

3. Operasi Kleene star ($\star$) dalam regular expression menyatakan...

a.) Satu atau lebih pengulangan

b.) Nol atau lebih pengulangan

c.) Tepat satu kemunculan

d.) Pilihan antara dua pattern

4. Subset construction digunakan untuk...

a.) Mengkonversi regular expression ke NFA

b.) Mengkonversi NFA ke DFA

c.) Minimisasi DFA

d.) Membangun lexical analyzer

5. Pattern `[0-9]+` dalam regular expression cocok untuk...

a.) Identifier

b.) Integer literal

c.) String literal

d.) Keyword

### 18.1.3 Quiz Bab 3: Implementasi Lexer (Hand-Written)

1. Keuntungan hand-written lexer dibanding lexer generator adalah...

a.) Lebih cepat dalam development

b.) Kontrol penuh dan pemahaman mendalam terhadap proses tokenization

c.) Kode yang dihasilkan lebih pendek

d.) Tidak perlu menangani edge case

2. Proses utama yang dilakukan lexer adalah...

a.) Parsing struktur program

b.) Tokenization atau pengenalan token dari source code

c.) Type checking

d.) Code generation

3. Komentar multi-line dalam C menggunakan...

a.) `//`

b.) `#`

c.) `/* */`

d.) `--`

4. Escape sequence `\n` menyatakan...

a.) Karakter backslash

b.) Karakter newline

c.) Karakter tab

d.) Karakter null

5. Output dari lexer adalah...

   a.) Parse tree

   b.) Stream of tokens

   c.) Symbol table

   d.) Machine code

### 18.1.4 Quiz Bab 4: Lexer Generator (Flex/re2c)

1. File specification Flex memiliki ekstensi...

   a.) `.y`

   b.) `.l`

   c.) `.c`

   d.) `.flex`

2. Bagian Rules dalam file Flex berisi...

   a.) Definisi makro dan C code

   b.) Pasangan pattern–action (regular expression dan kode C)

   c.) Fungsi `main()` dan `yywrap()`

   d.) Deklarasi token

3. Ketika beberapa pattern cocok dengan input, Flex memilih...

   a.) Pattern yang pertama didefinisikan

   b.) Longest match

   c.) Shortest match

   d.) Pattern yang terakhir didefinisikan

4. Tool re2c adalah...

   a.) Parser generator

   b.) Lexer generator untuk C/C++

   c.) Optimizer

   d.) Linker

5. Keuntungan lexer generator dibanding hand-written lexer adalah...

a.) Kontrol lebih penuh atas implementasi

b.) Produktivitas lebih tinggi dan specification lebih mudah di-maintain

c.) Lebih cepat saat runtime

d.) Tidak memerlukan dependency eksternal

### 18.1.5 Quiz Bab 5: Context-Free Grammar dan Parsing

1. CFG $G = (V, \Sigma, R, S)$: $S$ menyatakan...

   a.) Himpunan terminal

   b.) Himpunan nonterminal

   c.) Start symbol

   d.) Himpunan production

2. Grammar disebut ambiguous jika...

   a.) Memiliki left recursion

   b.) Memiliki lebih dari satu production untuk satu nonterminal

   c.) Ada string yang dapat di-parse dengan lebih dari satu parse tree

   d.) Memiliki $\epsilon$-production

3. Eliminasi left recursion diperlukan agar grammar...

   a.) Dapat di-parse dengan bottom-up parser

   b.) Dapat di-parse dengan recursive descent (top-down) parser

   c.) Menjadi unambiguous

   d.) Mendukung semua fitur bahasa

4. Notasi BNF adalah singkatan dari...

   a.) Binary Normal Form

   b.) Backus-Naur Form

   c.) Boolean Notation Format

   d.) Base Notation Form

5. Fase kompilasi yang menggunakan CFG adalah...

   a.) Lexical analysis

   b.) Syntax analysis

   c.) Semantic analysis

   d.) Code optimization

### 18.1.6 Quiz Bab 6: Top-Down Parsing dan Recursive Descent

1. Top-down parsing membangun parse tree...

   a.) Dari leaves ke root

   b.) Dari root (start symbol) ke leaves

   c.) Secara paralel

   d.) Dari tengah ke kedua ujung

2. Recursive descent parser cocok untuk grammar...

   a.) Dengan left recursion

   b.) LL(1) yang unambiguous

   c.) LR(1) saja

   d.) Yang ambiguous

3. Dalam recursive descent, setiap nonterminal biasanya diimplementasikan sebagai...

   a.) Struct atau class

   b.) Fungsi atau prosedur

   c.) Macro

   d.) Global variable

4. Precedence operator dalam recursive descent untuk ekspresi aritmatika ditangani dengan...

   a.) Urutan aturan grammar (layering: expression $\rightarrow$ term $\rightarrow$ factor)

   b.) Tabel lookup

   c.) Register khusus

   d.) Stack terpisah

5. Top-down parsing menggunakan...

   a.) Rightmost derivation

   b.) Leftmost derivation

   c.) Bottom-up reduction

   d.) Reverse derivation

### 18.1.7 Quiz Bab 7: Bottom-Up Parsing dan Parser Generator

1. Bottom-up parsing membangun parse tree...

   a.) Dari root ke leaves

   b.) Dari leaves (input tokens) ke root (start symbol)

   c.) Hanya untuk ekspresi

   d.) Menggunakan recursive calls

2. Operasi **reduce** dalam shift-reduce parsing berarti...

   a.) Memasukkan token ke stack

   b.) Mengganti handle di stack dengan LHS production

   c.) Mengeluarkan token dari stack

   d.) Memindahkan pointer input

3. LALR(1) parser merupakan pilihan populer karena...

   a.) Lebih powerful dari CLR(1)

   b.) Tabel parsing lebih kecil daripada CLR(1) tapi tetap cukup powerful

   c.) Tidak memerlukan lookahead

   d.) Dapat menangani grammar ambiguous

4. Bison adalah...

   a.) Lexer generator

   b.) Parser generator (implementasi GNU dari Yacc)

   c.) Optimizer

   d.) Assembler

5. Handle dalam bottom-up parsing adalah...

   a.) Simbol paling kiri yang akan di-expand

   b.) Substring yang cocok dengan RHS production dan reduced menuju start symbol

   c.) Lookahead token

   d.) State awal parser

### 18.1.8 Quiz Bab 8: Parser Generator (Bison/Yacc) dan Praktikum

1. File grammar Bison biasanya berekstensi...

   a.) `.l`

   b.) `.y`

   c.) `.lex`

   d.) `.bnf`

2. Semantic action dalam Bison berfungsi untuk...

   a.) Mendefinisikan token

   b.) Membangun AST atau melakukan aksi saat production di-reduce

   c.) Menangani error saja

   d.) Mengatur precedence

3. Integrasi Flex dan Bison: Flex menghasilkan...

   a.) Parser; Bison menghasilkan lexer

   b.) Lexer; Bison menghasilkan parser

   c.) Keduanya menghasilkan parser

   d.) Keduanya menghasilkan lexer

4. %token dalam file Bison digunakan untuk...

   a.) Mendefinisikan nonterminal

   b.) Mendeklarasikan token yang dikirim lexer

   c.) Mendefinisikan semantic value

   d.) Menentukan start symbol

5. Bison default menggunakan...

   a.) LR(0) parsing

   b.) LALR(1) parsing

   c.) SLR(1) parsing

   d.) GLR parsing

### 18.1.9 Quiz Bab 9: Abstract Syntax Tree (AST)

1. Perbedaan utama AST dengan parse tree adalah...

   a.) AST lebih detail daripada parse tree

   b.) AST menghilangkan detail sintaksis yang tidak relevan; lebih kompak

   c.) Parse tree tidak memiliki root

   d.) AST hanya untuk ekspresi

2. Traversal **in-order** pada AST ekspresi `a + b` mengunjungi node dalam urutan...

   a.) +, a, b

   b.) a, +, b

   c.) a, b, +

   d.) +, b, a

3. Visitor pattern biasanya digunakan untuk...

   a.) Mem-parse token

   b.) Traverse dan memproses node AST (misalnya pretty-print, analisis)

   c.) Generate lexer

   d.) Allokasi register

4. Constant folding pada AST mengubah...

   a.) Variabel menjadi konstanta

   b.) Ekspresi konstanta (mis. `3+5`) menjadi satu nilai (mis. `8`)

   c.) Semua node menjadi leaf

   d.) AST menjadi parse tree

5. AST digunakan sebagai input oleh...

   a.) Hanya lexical analyzer

   b.) Semantic analysis dan code generation

   c.) Hanya parser

   d.) Hanya optimizer

### 18.1.10 Quiz Bab 10: Symbol Table dan Scope Management

1. Symbol table menyimpan informasi tentang...

   a.) Hanya keywords

   b.) Identifier (variabel, fungsi, tipe) beserta atribut

   c.) Hanya variabel global

   d.) Hanya nama fungsi

2. Nested scope biasanya diimplementasikan dengan...

   a.) Array linear

   b.) Stack of hash tables (atau struktur serupa)

   c.) Single global table

   d.) Linked list tanpa hierarchy

3. Shadowing terjadi ketika...

   a.) Variabel tidak dideklarasikan

   b.) Identifier di inner scope menutupi identifier sama di outer scope

   c.) Ada dua variabel dengan tipe berbeda

   d.) Fungsi dipanggil dengan argumen salah

4. Operasi **lookup** pada symbol table berarti...

   a.) Menambah entry baru

   b.) Mencari identifier dan mengembalikan atributnya (jika ada)

   c.) Menghapus scope

   d.) Mendekomposisi tabel

5. Symbol table digunakan oleh...

   a.) Hanya lexical analyzer

   b.) Semantic analysis, type checker, dan code generator

   c.) Hanya parser

   d.) Hanya optimizer

### 18.1.11 Quiz Bab 11: Type Checking dan Semantic Analysis

1. Static type checking dilakukan...

   a.) Saat runtime

   b.) Saat compile time

   c.) Hanya untuk fungsi

   d.) Hanya untuk variabel global

2. Type promotion `int` $\rightarrow$ `float` berarti...

   a.) float bisa di-assign ke int tanpa cast

   b.) int bisa di-assign ke float (implicit conversion)

   c.) int dan float tidak kompatibel

   d.) Hanya explicit cast yang diperbolehkan

3. Output semantic analysis biasanya berupa...

   a.) Token stream

   b.) Annotated AST (dengan informasi tipe) dan daftar error semantik

   c.) Machine code

   d.) Parse tree mentah

4. Contoh semantic error adalah...

   a.) Token tidak valid

   b.) Variabel tidak dideklarasikan, type mismatch, wrong number of arguments

   c.) Kurung tidak seimbang

   d.) Keyword salah eja

5. Dynamic type checking biasanya digunakan oleh bahasa...

   a.) C dan C++

   b.) Python dan JavaScript (tanpa static typing)

   c.) Rust

   d.) Java

### 18.1.12 Quiz Bab 12: Intermediate Code Generation

1. Three-address code biasanya berbentuk...

   a.) Satu operasi per instruksi (mis. `t = a op b`)

   b.) Satu operand per instruksi

   c.) Tree structure

   d.) Bytecode stack-based

2. IR (Intermediate Representation) bersifat machine-independent artinya...

   a.) Hanya untuk satu arsitektur

   b.) Tidak tergantung arsitektur target; memungkinkan portabilitas

   c.) Harus dioptimasi manual

   d.) Tidak bisa diubah

3. Common subexpression elimination adalah...

   a.) Menghapus semua ekspresi

   b.) Menggunakan hasil komputasi yang sama untuk ekspresi yang muncul berulang

   c.) Mengganti variabel dengan konstanta

   d.) Menghapus dead code

4. Quadruples merepresentasikan instruksi dalam bentuk...

   a.) (op, arg1, arg2, result)

   b.) Hanya (op, result)

   c.) Tree

   d.) Daftar token

5. Input generator TAC (three-address code) adalah...

   a.) Token stream

   b.) AST (biasanya hasil semantic analysis)

   c.) Source code mentah

   d.) Machine code

### 18.1.13 Quiz Bab 13: Runtime Environment dan Memory Management

1. Activation record (stack frame) digunakan untuk...

   a.) Menyimpan kode program

   b.) Setiap pemanggilan fungsi: parameter, return address, variabel lokal

   c.) Hanya variabel global

   d.) Hanya konstanta

2. Stack dalam memory layout biasanya tumbuh...

   a.) Ke atas (low $\rightarrow$ high address)

   b.) Ke bawah (high $\rightarrow$ low address)

   c.) Secara acak

   d.) Hanya untuk heap

3. Variabel global dan static biasanya dialokasikan di...

   a.) Stack

   b.) Heap

   c.) Static/global data region

   d.) Register saja

4. Garbage collection umumnya mengelola...

   a.) Stack frame

   b.) Objek di heap yang tidak lagi direferensi

   c.) Variabel lokal

   d.) Kode program

5. Calling convention mengatur...

   a.) Hanya urutan parameter

   b.) Cara mem-pass parameter, return value, siapa membersihkan stack

   c.) Hanya return value

   d.) Hanya alokasi heap

### 18.1.14 Quiz Bab 14: Code Generation

1. Instruction selection adalah...

   a.) Memilih instruksi mesin untuk setiap operasi IR

   b.) Memilih register saja

   c.) Memilih optimasi

   d.) Memilih target OS

2. Register allocation bertujuan...

   a.) Mengurangi jumlah instruksi

   b.) Memetakan variabel/temporary ke register (atau memory jika perlu)

   c.) Hanya menggunakan stack

   d.) Menghapus label

3. RISC-V adalah contoh...

   a.) Lexer generator

   b.) Instruction set architecture (ISA) / arsitektur target

   c.) Format IR

   d.) Optimizer

4. Input code generator biasanya...

   a.) Token stream

   b.) IR (mis. TAC), symbol table, spesifikasi target

   c.) Hanya source code

   d.) Hanya AST tanpa IR

5. Output code generator dapat berupa...

   a.) Hanya binary

   b.) Assembly code atau machine code

   c.) Hanya AST

   d.) Hanya symbol table

### 18.1.15 Quiz Bab 15: Optimasi Kompilator Dasar

1. Optimasi kompilator tidak boleh...

   a.) Mengubah semantik program

   b.) Mengurangi ukuran kode

   c.) Meningkatkan kecepatan eksekusi

   d.) Melakukan constant folding

2. Constant folding mengubah `x = 3 + 5` menjadi...

   a.) `x = 3 + 5`

   b.) `x = 8`

   c.) Menghapus seluruh baris

   d.) `x = 35`

3. Dead code elimination menghapus...

   a.) Semua konstanta

   b.) Kode yang tidak pernah dieksekusi atau hasilnya tidak dipakai

   c.) Semua komentar

   d.) Semua label

4. Basic block adalah...

   a.) Seluruh program

   b.) Urutan instruksi dengan satu entry, satu exit, tidak ada branch di tengah

   c.) Hanya satu instruksi

   d.) Hanya loop

5. Constant propagation mengganti...

   a.) Semua variabel dengan konstanta

   b.) Penggunaan variabel yang nilainya diketahui konstan dengan nilai tersebut

   c.) Semua operator

   d.) Semua label

## 18.2 Kunci Jawaban

**Cara membaca:** Untuk Quiz Bab 1, jawaban benar soal 1 = **b**, soal 2 = **c**, soal 3 = **b**, soal 4 = **d**, soal 5 = **c**. Demikian seterusnya untuk bab lainnya.

| Bab | Materi | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 1 | Pengenalan Kompilator | b | c | b | d | c |
| 2 | Regular Expression dan FA | b | b | b | b | b |
| 3 | Implementasi Lexer | b | b | c | b | b |
| 4 | Lexer Generator | b | b | b | b | b |
| 5 | CFG dan Parsing | c | c | b | b | b |
| 6 | Top-Down Parsing | b | b | b | a | b |
| 7 | Bottom-Up Parsing | b | b | b | b | b |
| 8 | Parser Generator (Bison) | b | b | b | b | b |
| 9 | AST | b | b | b | b | b |
| 10 | Symbol Table | b | b | b | b | b |
| 11 | Type Checking | b | b | b | b | b |
| 12 | Intermediate Code | a | b | b | a | b |
| 13 | Runtime Environment | b | b | c | b | b |
| 14 | Code Generation | a | b | b | b | b |
| 15 | Optimasi | a | b | b | b | b |

Tabel 18.1: Kunci jawaban quiz Bab 1–15. Kolom 1–5 = nomor soal; isi = pilihan benar (a, b, c, atau d).

# 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 $\epsilon$-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 $\epsilon$-closure dari state awal NFA sebagai state awal DFA.

ii. Untuk setiap state DFA dan setiap simbol input, hitung transisi dan $\epsilon$-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
```

# Bab 20

## Tutorial: Membuat Kompilator Sederhana untuk Bahasa C

### 20.1 Tujuan Pembelajaran dan Pendahuluan

Tutorial ini menyajikan **versi minimal yang dapat dijalankan** dari proyek compiler subset C. Konsep dan fase-fase (lexical analysis, parsing, AST, code generation, assembling, linking) mengikuti materi Bab 2–16; spesifikasi token dan grammar selaras dengan Bab 1 (Bagian 1.9). Karena fokus pada "build dari nol" yang singkat, subset di sini dibatasi pada `print("string");`; setelah menyelesaikan Bab 2–16, pembaca dapat memperluas ke deklarasi, assignment, dan ekspresi sesuai grammar proyek. Kode dalam bab ini dapat dipandang sebagai langkah pertama atau snapshot minimal dari codebase proyek di `proyek-compiler-subset-c/`.

#### 20.1.1 Tujuan Pembelajaran

Setelah mempelajari bab ini, mahasiswa diharapkan mampu:

1. Memahami arsitektur compiler sederhana dari awal hingga akhir
2. Mengimplementasikan lexer sederhana dalam bahasa C
3. Mengimplementasikan parser recursive descent dalam bahasa C
4. Mengimplementasikan code generator yang menghasilkan assembly code
5. Mengintegrasikan assembler dan linker untuk menghasilkan executable file
6. Membangun compiler lengkap yang dapat mengkompilasi program sederhana menjadi .exe

#### 20.1.2 Overview Compiler yang Akan Dibuat

Dalam tutorial ini, kita akan membuat compiler sederhana yang dapat mengkompilasi subset minimal bahasa C menjadi executable Windows (.exe). Compiler ini mengimplementasikan

fase-fase yang sama dengan proyek Bab 2–16 (lexer, parser, codegen, assemble, link), dengan subset terbatas pada `print("string");` agar tutorial singkat. Compiler ini akan mengimplementasikan semua fase kompilasi yang telah dipelajari:

1. **Lexical Analysis**: Memecah source code menjadi token-token
2. **Syntax Analysis**: Membangun AST dari token stream
3. **Code Generation**: Menghasilkan assembly code dari AST
4. **Assembling**: Mengubah assembly menjadi object file
5. **Linking**: Menghasilkan executable file

### 20.1.3 Fitur yang Didukung

Compiler sederhana ini akan mendukung:

- **Print statement**: `print("string");`
- **String literals**: String dalam tanda kutip ganda
- **Minimal syntax**: Semicolon dan parentheses

Meskipun sangat sederhana, compiler ini akan menunjukkan semua fase kompilasi secara lengkap dan dapat menghasilkan executable yang benar-benar dapat dijalankan.

### 20.1.4 Contoh Program Target

Program yang akan kita kompilasi adalah:

Listing 20.1: Program hello.c

```
print("hello world !!!");
```

Program ini akan dikompilasi menjadi `hello.exe` yang ketika dijalankan akan menampilkan:

```
hello world !!!
```

### 20.1.5 Tools yang Diperlukan

Untuk mengikuti tutorial ini, Anda memerlukan:

1. **C Compiler**:
    - GCC (MinGW untuk Windows)
    - Atau Microsoft Visual C++ Compiler (cl.exe)

- Atau TCC (Tiny C Compiler) - lebih sederhana

2. **NASM (Netwide Assembler)**:
   - Download dari: https://www.nasm.us/
   - Versi untuk Windows (nasm.exe)
   - Digunakan untuk meng-assemble kode assembly menjadi object file
3. **Linker**:
   - Microsoft Linker (link.exe) - biasanya sudah termasuk dengan Visual Studio
   - Atau MinGW linker (ld.exe) - sudah termasuk dengan GCC
   - Digunakan untuk mengubah object file menjadi executable
4. **Text Editor**: Editor teks apapun untuk menulis kode C

### 20.1.6 Struktur Project

Struktur project compiler yang akan kita buat:

```
compiler/
 lexer.h          # Header file untuk lexer
 lexer.c          # Implementasi lexer
 parser.h         # Header file untuk parser
 parser.c         # Implementasi parser
 codegen.h        # Header file untuk code generator
 codegen.c        # Implementasi code generator
 main.c           # Driver program utama
 build.bat        # Script untuk build compiler
 test.bat         # Script untuk test compiler
 hello.c          # Program test: print("hello world !!!");
```

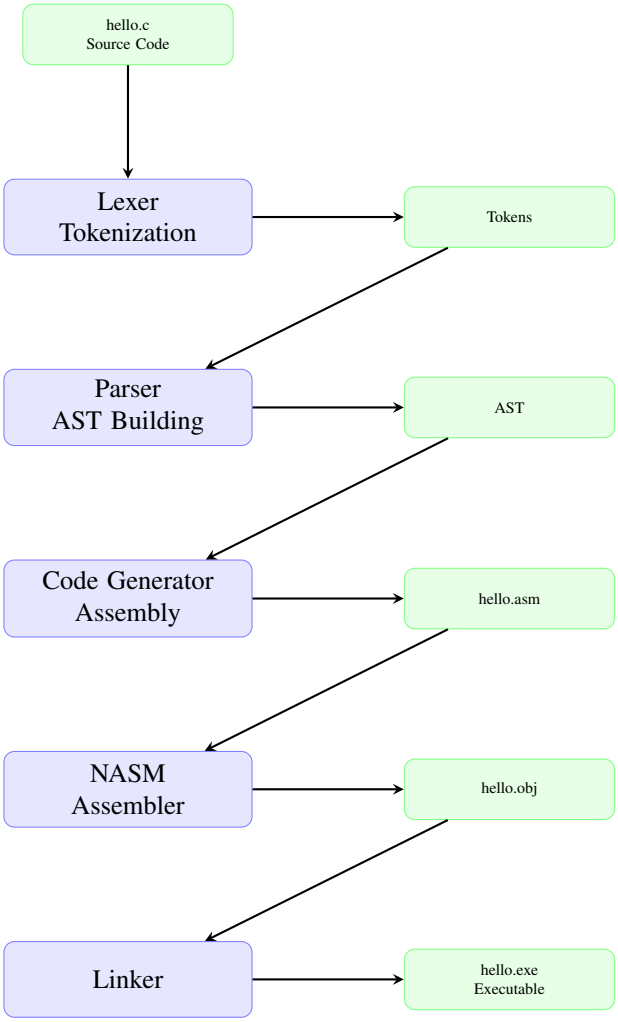
### 20.1.7 Arsitektur Compiler

Diagram berikut menunjukkan arsitektur compiler sederhana yang akan kita buat:

### 20.1.8 Langkah-Langkah Pembuatan

Tutorial ini akan dibagi menjadi beberapa bagian:

1. **Bagian 1**: Implementasi Lexer - mengenali token dari source code
2. **Bagian 2**: Implementasi Parser - membangun AST dari token stream
3. **Bagian 3**: Code Generation - menghasilkan assembly code




Gambar 20.1: Arsitektur compiler sederhana

4. **Bagian 4**: Assembling dan Linking - menghasilkan executable

5. **Bagian 5**: Testing dan Verifikasi - memastikan compiler bekerja dengan benar

6. **Bagian 6**: Extensions - menambahkan fitur tambahan

Setiap bagian akan dilengkapi dengan kode lengkap yang dapat langsung digunakan dan di-compile.

## 20.2 Implementasi Lexer

Lexer adalah komponen pertama dalam compiler yang bertugas memecah source code menjadi token-token. Untuk compiler sederhana kita, lexer hanya perlu mengenali beberapa jenis token.

### 20.2.1 Token Types

Kita akan mendefinisikan token types berikut:

Listing 20.2: Token types dalam lexer.h

```
#ifndef LEXER_H
#define LEXER_H

typedef enum {
    TOKEN_EOF = 0,
    TOKEN_PRINT,
    TOKEN_STRING,
    TOKEN_LPAREN,       // (
    TOKEN_RPAREN,       // )
    TOKEN_SEMICOLON,    // ;
    TOKEN_ERROR
} TokenType;

typedef struct {
    TokenType type;
    char* value;        // Untuk string literal, berisi nilai string
    int line;
    int column;
} Token;

// Fungsi-fungsi lexer
void initLexer(const char* source);
Token nextToken(void);
void freeLexer(void);

#endif
```

### 20.2.2 Struktur Data Lexer

Lexer akan menyimpan state berikut:

- Source code yang sedang di-scan
- Posisi saat ini (current position)
- Line dan column number untuk error reporting

### 20.2.3 Implementasi Lexer

Berikut adalah implementasi lengkap lexer dalam bahasa C:

Listing 20.3: Implementasi lexer.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "lexer.h"

// State lexer
static const char* source = NULL;
static int pos = 0;
static int line = 1;
static int column = 1;
static int sourceLen = 0;

void initLexer(const char* src) {
    source = src;
    pos = 0;
    line = 1;
    column = 1;
    sourceLen = strlen(src);
}

static void skipWhitespace(void) {
    while (pos < sourceLen && isspace(source[pos])) {
        if (source[pos] == '\n') {
            line++;
            column = 1;
        } else {
            column++;
        }
        pos++;
    }
}

static Token makeToken(TokenType type, const char* value) {
    Token token;
    token.type = type;
    token.line = line;
    token.column = column;
```

```
39
40      if (value != NULL) {
41          token.value = (char*)malloc(strlen(value) + 1);
42          strcpy(token.value, value);
43      } else {
44          token.value = NULL;
45      }
46
47      return token;
48  }
49
50  static Token scanString(void) {
51      int start = pos;
52      pos++; // Skip opening quote
53      column++;
54
55      // Scan sampai menemukan closing quote
56      while (pos < sourceLen && source[pos] != '"') {
57          if (source[pos] == '\n') {
58              // Error: newline dalam string literal
59              return makeToken(TOKEN_ERROR, "Unclosed string literal");
60          }
61          pos++;
62          column++;
63      }
64
65      if (pos >= sourceLen) {
66          return makeToken(TOKEN_ERROR, "Unclosed string literal");
67      }
68
69      // Extract string value (without quotes)
70      int len = pos - start - 1;
71      char* str = (char*)malloc(len + 1);
72      strncpy(str, source + start + 1, len);
73      str[len] = '\0';
74
75      pos++; // Skip closing quote
76      column++;
77
78      Token token = makeToken(TOKEN_STRING, str);
79      free(str);
80      return token;
81  }
82
83  static Token scanIdentifier(void) {
84      int start = pos;
85
86      while (pos < sourceLen &&
87             (isalnum(source[pos]) || source[pos] == '_')) {
88          pos++;
89          column++;
90      }
91
92      int len = pos - start;
```

```
93      char* ident = (char*)malloc(len + 1);
94      strncpy(ident, source + start, len);
95      ident[len] = '\0';
96
97      // Check if it's a keyword
98      if (strcmp(ident, "print") == 0) {
99          free(ident);
100         return makeToken(TOKEN_PRINT, NULL);
101     }
102
103     // For now, we only support "print" keyword
104     free(ident);
105     return makeToken(TOKEN_ERROR, "Unknown identifier");
106 }
107
108 Token nextToken(void) {
109     skipWhitespace();
110
111     if (pos >= sourceLen) {
112         return makeToken(TOKEN_EOF, NULL);
113     }
114
115     char c = source[pos];
116
117     // String literal
118     if (c == '"') {
119         return scanString();
120     }
121
122     // Identifier or keyword
123     if (isalpha(c) || c == '_') {
124         return scanIdentifier();
125     }
126
127     // Single character tokens
128     switch (c) {
129         case '(':
130             pos++;
131             column++;
132             return makeToken(TOKEN_LPAREN, NULL);
133         case ')':
134             pos++;
135             column++;
136             return makeToken(TOKEN_RPAREN, NULL);
137         case ';':
138             pos++;
139             column++;
140             return makeToken(TOKEN_SEMICOLON, NULL);
141         default:
142             // Unknown character
143             char error[64];
144             snprintf(error, sizeof(error), "Unexpected character: %c", c)
    ↪ ;
145             pos++;
```

```
            column++;
            return makeToken(TOKEN_ERROR, error);
    }
}

void freeLexer(void) {
    // Cleanup jika diperlukan
    source = NULL;
    pos = 0;
    line = 1;
    column = 1;
}
```

### 20.2.4 Contoh Penggunaan Lexer

Berikut adalah contoh penggunaan lexer untuk tokenize program `print("hello world !!!");`:

Listing 20.4: Contoh penggunaan lexer

```
#include "lexer.h"
#include <stdio.h>

int main() {
    const char* source = "print(\"hello world !!!\");";

    initLexer(source);

    Token token;
    do {
        token = nextToken();

        printf("Token: ");
        switch (token.type) {
            case TOKEN_PRINT:
                printf("PRINT");
                break;
            case TOKEN_STRING:
                printf("STRING(\"%s\")", token.value);
                break;
            case TOKEN_LPAREN:
                printf("LPAREN");
                break;
            case TOKEN_RPAREN:
                printf("RPAREN");
                break;
            case TOKEN_SEMICOLON:
                printf("SEMICOLON");
                break;
            case TOKEN_EOF:
                printf("EOF");
                break;
            case TOKEN_ERROR:
```

```
                printf("ERROR: %s", token.value);
                break;
        }
        printf(" at line %d, column %d\n", token.line, token.column);

        if (token.value != NULL) {
            free(token.value);
        }
    } while (token.type != TOKEN_EOF && token.type != TOKEN_ERROR);

    freeLexer();
    return 0;
}
```

Output yang dihasilkan:

```
Token: PRINT at line 1, column 1
Token: LPAREN at line 1, column 6
Token: STRING("hello world !!!") at line 1, column 7
Token: RPAREN at line 1, column 26
Token: SEMICOLON at line 1, column 27
Token: EOF at line 1, column 28
```

### 20.2.5 Testing Lexer

Untuk menguji lexer, buatlah file test sederhana:

Listing 20.5: test$_lexer.c$

```
#include "lexer.h"
#include <stdio.h>
#include <assert.h>

void testLexer() {
    const char* source = "print(\"hello world !!!\");";
    initLexer(source);

    Token t1 = nextToken();
    assert(t1.type == TOKEN_PRINT);

    Token t2 = nextToken();
    assert(t2.type == TOKEN_LPAREN);

    Token t3 = nextToken();
    assert(t3.type == TOKEN_STRING);
    assert(strcmp(t3.value, "hello world !!!") == 0);

    Token t4 = nextToken();
    assert(t4.type == TOKEN_RPAREN);

    Token t5 = nextToken();
    assert(t5.type == TOKEN_SEMICOLON);

    Token t6 = nextToken();
```

```
    assert(t6.type == TOKEN_EOF);

    printf("Lexer test passed!\n");
    freeLexer();
}

int main() {
    testLexer();
    return 0;
}
```

Lexer ini sudah siap digunakan untuk tahap berikutnya: parsing.

## 20.3 Implementasi Parser

Parser bertugas menganalisis token stream dan membangun Abstract Syntax Tree (AST). Untuk compiler sederhana kita, kita akan menggunakan recursive descent parser.

### 20.3.1 Grammar Sederhana

Grammar untuk print statement:

```
program    → print_stmt
print_stmt  → PRINT LPAREN STRING RPAREN SEMICOLON
```

Grammar ini sangat sederhana karena kita hanya mendukung satu jenis statement: print statement.

### 20.3.2 Struktur AST

AST untuk compiler sederhana kita hanya perlu menyimpan informasi tentang print statement:

Listing 20.6: Struktur AST dalam parser.h

```
#ifndef PARSER_H
#define PARSER_H

#include "lexer.h"

typedef struct ASTNode {
    enum {
        AST_PRINT_STMT
    } type;

    union {
        struct {
            char* string_value;  // String yang akan di-print
        } print_stmt;
    } data;
} ASTNode;
```

```
17
18 // Fungsi-fungsi parser
19 ASTNode* parse(void);
20 void freeAST(ASTNode* node);
21 void printAST(ASTNode* node);
22
23 #endif
```

### 20.3.3 Implementasi Parser

Berikut adalah implementasi parser recursive descent:

Listing 20.7: Implementasi parser.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include "parser.h"
5  #include "lexer.h"
6
7  static Token currentToken;
8
9  static void advance(void) {
10     currentToken = nextToken();
11 }
12
13 static void expect(TokenType expected) {
14     if (currentToken.type != expected) {
15         fprintf(stderr, "Parse error at line %d, column %d: ",
16                 currentToken.line, currentToken.column);
17         fprintf(stderr, "Expected token type %d, got %d\n",
18                 expected, currentToken.type);
19         exit(1);
20     }
21     advance();
22 }
23
24 ASTNode* parsePrintStmt(void) {
25     ASTNode* node = (ASTNode*)malloc(sizeof(ASTNode));
26     node->type = AST_PRINT_STMT;
27
28     // Expect PRINT keyword
29     expect(TOKEN_PRINT);
30
31     // Expect LPAREN
32     expect(TOKEN_LPAREN);
33
34     // Expect STRING literal
35     if (currentToken.type != TOKEN_STRING) {
36         fprintf(stderr, "Parse error: Expected string literal\n");
37         exit(1);
38     }
39
40     // Copy string value
```

```
    node->data.print_stmt.string_value =
        (char*)malloc(strlen(currentToken.value) + 1);
    strcpy(node->data.print_stmt.string_value, currentToken.value);

    advance(); // Consume STRING token

    // Expect RPAREN
    expect(TOKEN_RPAREN);

    // Expect SEMICOLON
    expect(TOKEN_SEMICOLON);

    return node;
}

ASTNode* parse(void) {
    // Initialize lexer (should be done before calling parse)
    advance(); // Get first token

    // Parse print statement
    ASTNode* node = parsePrintStmt();

    // Expect EOF
    if (currentToken.type != TOKEN_EOF) {
        fprintf(stderr, "Parse error: Expected EOF\n");
        exit(1);
    }

    return node;
}

void freeAST(ASTNode* node) {
    if (node == NULL) return;

    switch (node->type) {
        case AST_PRINT_STMT:
            if (node->data.print_stmt.string_value != NULL) {
                free(node->data.print_stmt.string_value);
            }
            break;
    }

    free(node);
}

void printAST(ASTNode* node) {
    if (node == NULL) return;

    switch (node->type) {
        case AST_PRINT_STMT:
            printf("PrintStmt(\"%s\")\n",
                   node->data.print_stmt.string_value);
            break;
    }
```

```
95 }
```

### 20.3.4 Error Handling

Parser melakukan error handling dengan:

- Memeriksa token yang diharapkan dengan fungsi `expect()`
- Menampilkan pesan error yang jelas dengan informasi line dan column
- Menghentikan parsing jika terjadi error (untuk compiler sederhana)

### 20.3.5 Contoh Penggunaan Parser

Berikut adalah contoh penggunaan parser:

Listing 20.8: Contoh penggunaan parser

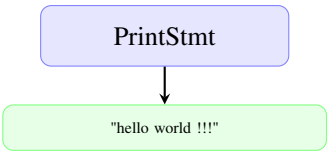
```c
#include "parser.h"
#include "lexer.h"
#include <stdio.h>

int main() {
    const char* source = "print(\"hello world !!!\");";

    // Initialize lexer
    initLexer(source);

    // Parse
    ASTNode* ast = parse();

    // Print AST
    printf("AST:\n");
    printAST(ast);

    // Cleanup
    freeAST(ast);
    freeLexer();

    return 0;
}
```

Output:

```
AST:
PrintStmt("hello world !!!")
```

### 20.3.6 Visualisasi AST

AST untuk program `print("hello world !!!");` dapat divisualisasikan sebagai:




Gambar 20.2: AST untuk print statement

### 20.3.7 Testing Parser

Untuk menguji parser, buatlah file test:

Listing 20.9: test$_parser.c$

```
#include "parser.h"
#include "lexer.h"
#include <stdio.h>
#include <assert.h>
#include <string.h>

void testParser() {
    const char* source = "print(\"hello world !!!\");";

    initLexer(source);
    ASTNode* ast = parse();

    assert(ast != NULL);
    assert(ast->type == AST_PRINT_STMT);
    assert(strcmp(ast->data.print_stmt.string_value,
                  "hello world !!!") == 0);

    printf("Parser test passed!\n");

    freeAST(ast);
    freeLexer();
}

int main() {
    testParser();
    return 0;
}
```

Parser sudah siap untuk tahap berikutnya: code generation.

## 20.4 Code Generation ke Assembly

Code generator adalah komponen yang mengubah AST menjadi assembly code. Untuk Windows, kita akan menghasilkan x86-64 assembly code yang menggunakan Windows API untuk output.

### 20.4.1 Target Assembly

Kita akan menghasilkan assembly code untuk Windows x86-64 yang menggunakan:

- Windows API: `GetStdHandle`, `WriteFile`, `ExitProcess`
- Format: NASM syntax (Intel syntax)
- Calling convention: Windows x64 calling convention

### 20.4.2 Struktur Code Generator

Listing 20.10: Header file codegen.h

```
#ifndef CODEGEN_H
#define CODEGEN_H

#include "parser.h"
#include <stdio.h>

// Generate assembly code dari AST
void generateCode(ASTNode* ast, FILE* output);

#endif
```

### 20.4.3 Implementasi Code Generator

Berikut adalah implementasi code generator yang menghasilkan assembly code untuk Windows:

Listing 20.11: Implementasi codegen.c

```
#include <stdio.h>
#include <stdarg.h>
#include <string.h>
#include "codegen.h"
#include "parser.h"

static void emit(FILE* out, const char* format, ...) {
    va_list args;
    va_start(args, format);
    vfprintf(out, format, args);
    va_end(args);
}

static void escapeString(const char* str, char* buffer, int bufferSize) {
    int j = 0;
    for (int i = 0; str[i] != '\0' && j < bufferSize - 1; i++) {
        if (str[i] == '\'') {
            buffer[j++] = '\'';
            buffer[j++] = '\'';
        } else if (str[i] == '\n') {
```

```
            buffer[j++] = '\\';
            buffer[j++] = 'n';
        } else if (str[i] == '\t') {
            buffer[j++] = '\\';
            buffer[j++] = 't';
        } else {
            buffer[j++] = str[i];
        }
    }
    buffer[j] = '\0';
}

void generateCode(ASTNode* ast, FILE* output) {
    if (ast == NULL) return;

    // Write data section
    emit(output, "section .data\n");
    emit(output, "    msg db '%s', 0\n", ast->data.print_stmt.
↪ string_value);
    emit(output, "    msg_len equ $ - msg - 1\n");
    emit(output, "\n");

    // Write text section
    emit(output, "section .text\n");
    emit(output, "    global _start\n");
    emit(output, "    extern ExitProcess\n");
    emit(output, "    extern GetStdHandle\n");
    emit(output, "    extern WriteFile\n");
    emit(output, "\n");

    // Write _start function
    emit(output, "_start:\n");
    emit(output, "    ; Get stdout handle\n");
    emit(output, "    mov rcx, -11        ; STD_OUTPUT_HANDLE\n");
    emit(output, "    sub rsp, 32         ; Shadow space\n");
    emit(output, "    call GetStdHandle\n");
    emit(output, "    add rsp, 32\n");
    emit(output, "    mov rbx, rax        ; Save handle in rbx\n");
    emit(output, "\n");

    emit(output, "    ; Write to stdout\n");
    emit(output, "    mov rcx, rbx        ; hFile (stdout handle)\n");
    emit(output, "    lea rdx, [msg]      ; lpBuffer (pointer to message)
↪ \n");
    emit(output, "    mov r8, msg_len     ; nNumberOfBytesToWrite\n");
    emit(output, "    lea r9, [rsp-8]     ; lpNumberOfBytesWritten (local
↪  var)\n");
    emit(output, "    mov qword [rsp-16], 0 ; lpOverlapped (NULL)\n");
    emit(output, "    sub rsp, 32         ; Shadow space\n");
    emit(output, "    call WriteFile\n");
    emit(output, "    add rsp, 32\n");
    emit(output, "\n");

    emit(output, "    ; Exit process\n");
```

```
72      emit(output, "    mov rcx, 0          ; Exit code\n");
73      emit(output, "    sub rsp, 32         ; Shadow space\n");
74      emit(output, "    call ExitProcess\n");
75      emit(output, "    add rsp, 32\n");
76  }
```

### 20.4.4 Penjelasan Assembly Code

Assembly code yang dihasilkan terdiri dari:

#### Data Section

```
section .data
    msg db 'hello world !!!', 0
    msg_len equ $ - msg - 1
```

- `section .data`: Section untuk data yang dapat diubah
- `msg db`: Define byte - menyimpan string literal

- $\texttt{msg}_{len}$ : $Panjang string (tanpa null terminator)$

#### Text Section

```
section .text
    global _start
    extern ExitProcess
    extern GetStdHandle
    extern WriteFile
```

- `section .text`: Section untuk kode executable

- `global`
  $_{s}tart$ : $Entry point program$`extern` : $Deklarasi fungsi eksternal dari Windows API$

#### Main Function

- `_start:`

```
    ; Get stdout handle
    mov rcx, -11        ; STD_OUTPUT_HANDLE
    call GetStdHandle
    mov rbx, rax        ; Save handle
```

  - `-11`: Konstanta `STD_OUTPUT_HANDLE` untuk stdout
  - `GetStdHandle`: Windows API untuk mendapatkan handle ke stdout
  - Handle disimpan di `rbx` untuk digunakan kemudian

```
    ; Write to stdout
    mov rcx, rbx        ; hFile
    lea rdx, [msg]      ; lpBuffer
    mov r8, msg_len     ; nNumberOfBytesToWrite
    lea r9, [rsp-8]     ; lpNumberOfBytesWritten
    mov qword [rsp-16], 0 ; lpOverlapped
    call WriteFile
```

- `WriteFile`: Windows API untuk menulis ke file/handle
- Parameter sesuai Windows x64 calling convention:
  - `rcx`: Parameter pertama (handle)
  - `rdx`: Parameter kedua (buffer)
  - `r8`: Parameter ketiga (length)
  - `r9`: Parameter keempat (bytes written)
  - `[rsp-16]`: Parameter kelima di stack (overlapped)
- Shadow space: 32 bytes yang harus disediakan sebelum call

```
    ; Exit process
    mov rcx, 0          ; Exit code
    call ExitProcess
```

- `ExitProcess`: Windows API untuk mengakhiri proses
- Exit code 0 berarti sukses

### 20.4.5 Contoh Output Assembly

Untuk program `print("hello world !!!");`, code generator akan menghasilkan:

```
section .data
    msg db 'hello world !!!', 0
    msg_len equ $ - msg - 1

section .text
    global _start
    extern ExitProcess
    extern GetStdHandle
    extern WriteFile

_start:
    ; Get stdout handle
    mov rcx, -11        ; STD_OUTPUT_HANDLE
```

```
    sub rsp, 32          ; Shadow space
    call GetStdHandle
    add rsp, 32
    mov rbx, rax         ; Save handle in rbx

    ; Write to stdout
    mov rcx, rbx         ; hFile (stdout handle)
    lea rdx, [msg]       ; lpBuffer (pointer to message)
    mov r8, msg_len      ; nNumberOfBytesToWrite
    lea r9, [rsp-8]      ; lpNumberOfBytesWritten (local var)
    mov qword [rsp-16], 0 ; lpOverlapped (NULL)
    sub rsp, 32          ; Shadow space
    call WriteFile
    add rsp, 32

    ; Exit process
    mov rcx, 0           ; Exit code
    sub rsp, 32          ; Shadow space
    call ExitProcess
    add rsp, 32
```

### 20.4.6 Windows x64 Calling Convention

Penting untuk memahami Windows x64 calling convention:

- **First 4 parameters**: `rcx`, `rdx`, `r8`, `r9` (untuk integer/pointer)
- **Additional parameters**: Di stack, dari kanan ke kiri
- **Shadow space**: 32 bytes harus dialokasikan sebelum call (bahkan jika tidak ada parameter di stack)
- **Return value**: `rax` untuk integer/pointer
- **Caller-saved registers**: `rax`, `rcx`, `rdx`, `r8`, `r9`, `r10`, `r11`
- **Callee-saved registers**: `rbx`, `rbp`, `rsi`, `rdi`, `r12-r15`

### 20.4.7 Testing Code Generator

Untuk menguji code generator:

Listing 20.12: test$_codegen.c$

```c
#include "codegen.h"
#include "parser.h"
#include "lexer.h"
#include <stdio.h>

```

```
int main() {
    const char* source = "print(\"hello world !!!\");";

    initLexer(source);
    ASTNode* ast = parse();

    FILE* output = fopen("hello.asm", "w");
    if (output == NULL) {
        fprintf(stderr, "Cannot open output file\n");
        return 1;
    }

    generateCode(ast, output);
    fclose(output);

    printf("Assembly code generated: hello.asm\n");

    freeAST(ast);
    freeLexer();
    return 0;
}
```

Code generator sudah siap. File assembly yang dihasilkan akan di-assemble dan di-link pada tahap berikutnya.

## 20.5 Assembling dan Linking menjadi Executable

Setelah code generator menghasilkan assembly code, langkah berikutnya adalah meng-assemble dan meng-link menjadi executable file (.exe).

### 20.5.1 Proses Assembling

Assembling adalah proses mengubah assembly code menjadi object file (.obj). Kita akan menggunakan NASM (Netwide Assembler).

#### Command NASM

```
nasm -f win64 hello.asm -o hello.obj
```

Penjelasan parameter:

- `-f win64`: Format output untuk Windows 64-bit
- `hello.asm`: File input (assembly code)
- `-o hello.obj`: File output (object file)

### 20.5.2 Proses Linking

Linking adalah proses mengubah object file menjadi executable. Kita akan menggunakan Microsoft Linker atau MinGW linker.

#### Menggunakan Microsoft Linker (link.exe)

```
link hello.obj /subsystem:console /entry:_start /out:hello.exe kernel32.lib
```

Penjelasan parameter:

- `hello.obj`: Object file yang akan di-link
- `/subsystem:console`: Subsystem untuk console application
- `/entry:_start`: Entry point program
- `/out:hello.exe`: Nama output executable
- `kernel32.lib`: Library yang berisi Windows API functions (GetStdHandle, WriteFile, ExitProcess)

#### Menggunakan MinGW Linker (ld.exe)

Jika menggunakan MinGW, command-nya sedikit berbeda:

```
ld hello.obj -o hello.exe -e _start -subsystem:console kernel32.lib
```

Atau dengan gcc (lebih mudah):

```
gcc hello.obj -o hello.exe -nostdlib -e _start kernel32.lib
```

### 20.5.3 Batch File untuk Automasi

Untuk memudahkan proses build, kita dapat membuat batch file:

Listing 20.13: build.bat - Build kompilator

```
@echo off
echo Building kompilator...

REM Compile kompilator
gcc -o compiler.exe main.c lexer.c parser.c codegen.c

if errorlevel 1 (
    echo Compilation failed!
    exit /b 1
)

echo Kompilator built successfully: compiler.exe
```

Listing 20.14: compile.bat - Compile hello.c menjadi hello.exe

```
@echo off
echo Compiling hello.c...

REM Step 1: Run compiler to generate assembly
compiler.exe hello.c hello.asm

if errorlevel 1 (
    echo Compilation failed at code generation!
    exit /b 1
)

REM Step 2: Assemble with NASM
nasm -f win64 hello.asm -o hello.obj

if errorlevel 1 (
    echo Assembly failed!
    exit /b 1
)

REM Step 3: Link with Microsoft Linker
link hello.obj /subsystem:console /entry:_start /out:hello.exe kernel32.
↪ lib

if errorlevel 1 (
    echo Linking failed!
    exit /b 1
)

REM Step 4: Cleanup temporary files
del hello.obj hello.asm

echo Compilation successful: hello.exe
```

### 20.5.4 Main Program - Driver

Berikut adalah main program yang mengintegrasikan semua komponen:

Listing 20.15: main.c - Driver program

```
#include <stdio.h>
#include <stdlib.h>
#include "lexer.h"
#include "parser.h"
#include "codegen.h"

int main(int argc, char* argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <input.c> <output.asm>\n", argv[0]);
        return 1;
    }

    // Read source file
    FILE* input = fopen(argv[1], "r");
```

```
    if (input == NULL) {
        fprintf(stderr, "Cannot open input file: %s\n", argv[1]);
        return 1;
    }

    // Get file size
    fseek(input, 0, SEEK_END);
    long size = ftell(input);
    fseek(input, 0, SEEK_SET);

    // Read source code
    char* source = (char*)malloc(size + 1);
    fread(source, 1, size, input);
    source[size] = '\0';
    fclose(input);

    // Initialize lexer
    initLexer(source);

    // Parse
    ASTNode* ast = parse();
    if (ast == NULL) {
        fprintf(stderr, "Parse error\n");
        free(source);
        freeLexer();
        return 1;
    }

    // Generate code
    FILE* output = fopen(argv[2], "w");
    if (output == NULL) {
        fprintf(stderr, "Cannot open output file: %s\n", argv[2]);
        freeAST(ast);
        free(source);
        freeLexer();
        return 1;
    }

    generateCode(ast, output);
    fclose(output);

    // Cleanup
    freeAST(ast);
    free(source);
    freeLexer();

    printf("Compilation successful: %s\n", argv[2]);
    return 0;
}
```

### 20.5.5 Testing Lengkap

Langkah-langkah untuk testing kompilator lengkap:

**1. Build Kompilator**

```
build.bat
```

Ini akan menghasilkan `compiler.exe`.

**2. Buat File Test**

Buat file `hello.c`:

```
print("hello world !!!");
```

**3. Kompilasi dengan Kompilator**

```
compiler.exe hello.c hello.asm
```

Ini akan menghasilkan `hello.asm`.

**4. Assemble dengan NASM**

```
nasm -f win64 hello.asm -o hello.obj
```

Ini akan menghasilkan `hello.obj`.

**5. Link dengan Linker**

```
link hello.obj /subsystem:console /entry:_start /out:hello.exe kernel32.lib
```

Ini akan menghasilkan `hello.exe`.

**6. Jalankan Executable**

```
hello.exe
```

Output yang diharapkan:

```
hello world !!!
```

### 20.5.6 Script Lengkap untuk Testing

Berikut adalah script lengkap yang mengotomasi semua proses:

Listing 20.16: test.bat - Script testing lengkap

```
@echo off
echo ========================================
echo   Testing Simple C Compiler
echo ========================================
echo.

REM Step 1: Build kompilator
```

```
echo [1/5] Building kompilator...
call build.bat
if errorlevel 1 (
    echo ERROR: Failed to build kompilator
    exit /b 1
)
echo.

REM Step 2: Create test file
echo [2/5] Creating test file...
echo print("hello world !!!"); > hello.c
echo Test file created: hello.c
echo.

REM Step 3: Compile with our kompilator
echo [3/5] Compiling hello.c to hello.asm...
compiler.exe hello.c hello.asm
if errorlevel 1 (
    echo ERROR: Compilation failed
    exit /b 1
)
echo Assembly generated: hello.asm
echo.

REM Step 4: Assemble
echo [4/5] Assembling hello.asm...
nasm -f win64 hello.asm -o hello.obj
if errorlevel 1 (
    echo ERROR: Assembly failed
    echo Make sure NASM is installed and in PATH
    exit /b 1
)
echo Object file created: hello.obj
echo.

REM Step 5: Link
echo [5/5] Linking hello.obj...
link hello.obj /subsystem:console /entry:_start /out:hello.exe kernel32.
↪ lib
if errorlevel 1 (
    echo ERROR: Linking failed
    echo Make sure Microsoft Linker is available
    exit /b 1
)
echo Executable created: hello.exe
echo.

REM Step 6: Run
echo ========================================
echo   Running hello.exe...
echo ========================================
hello.exe
echo.

```

```
REM Step 7: Cleanup
echo Cleaning up temporary files...
del hello.obj hello.asm 2>nul
echo.
echo ========================================
echo   Test completed successfully!
echo ========================================
```

### 20.5.7 Troubleshooting

**Error: NASM not found**

- Pastikan NASM sudah di-install
- Pastikan NASM ada di PATH environment variable
- Atau gunakan full path ke nasm.exe

**Error: Linker not found**

- Untuk Microsoft Linker: Pastikan Visual Studio atau Windows SDK terinstall
- Untuk MinGW: Pastikan MinGW ada di PATH
- Atau gunakan gcc untuk linking: `gcc hello.obj -o hello.exe -nostdlib -e _start kernel32.lib`

**Error: kernel32.lib not found**

- Pastikan Windows SDK terinstall
- Atau gunakan full path ke kernel32.lib
- Untuk MinGW, biasanya tidak perlu specify kernel32.lib secara eksplisit

### 20.5.8 Kesimpulan

Dengan langkah-langkah di atas, kita telah berhasil membuat compiler sederhana yang dapat:

1. Membaca source code (hello.c)
2. Melakukan lexical analysis
3. Melakukan syntax analysis
4. Menghasilkan assembly code
5. Meng-assemble menjadi object file

6. Meng-link menjadi executable

7. Menghasilkan program yang dapat dijalankan

Compiler sederhana ini menunjukkan semua fase kompilasi secara lengkap dan menghasilkan executable yang benar-benar dapat dijalankan di Windows.

## 20.6 Extensions dan Perbaikan

Kompilator sederhana yang telah kita buat sudah berfungsi dengan baik untuk program minimal. Namun, ada banyak perbaikan dan extensions yang dapat ditambahkan untuk membuat kompilator lebih robust dan powerful.

### 20.6.1 Menambahkan Support untuk Ekspresi Sederhana

Saat ini kompilator hanya mendukung print statement dengan string literal. Kita dapat memperluasnya untuk mendukung ekspresi sederhana.

#### Modifikasi Grammar

Grammar yang diperluas:

```
program     → stmt
stmt        → print_stmt
print_stmt  → PRINT LPAREN expr RPAREN SEMICOLON
expr        → STRING | INTEGER | IDENTIFIER
```

#### Modifikasi AST

Tambahkan node untuk ekspresi:

Listing 20.17: AST yang diperluas

```
typedef struct ASTNode {
    enum {
        AST_PRINT_STMT,
        AST_STRING_EXPR,
        AST_INTEGER_EXPR,
        AST_IDENTIFIER_EXPR
    } type;

    union {
        struct {
            ASTNode* expr;  // Expression yang akan di-print
        } print_stmt;

        struct {
            char* value;
        } string_expr;
```

```
17
18          struct {
19              int value;
20          } integer_expr;
21
22          struct {
23              char* name;
24          } identifier_expr;
25      } data;
26  } ASTNode;
```

### Modifikasi Parser

Tambahkan fungsi parsing untuk ekspresi:

Listing 20.18: Parser untuk ekspresi

```
1  ASTNode* parseExpr(void) {
2      if (currentToken.type == TOKEN_STRING) {
3          ASTNode* node = (ASTNode*)malloc(sizeof(ASTNode));
4          node->type = AST_STRING_EXPR;
5          node->data.string_expr.value =
6              (char*)malloc(strlen(currentToken.value) + 1);
7          strcpy(node->data.string_expr.value, currentToken.value);
8          advance();
9          return node;
10     }
11
12     if (currentToken.type == TOKEN_INTEGER) {
13         ASTNode* node = (ASTNode*)malloc(sizeof(ASTNode));
14         node->type = AST_INTEGER_EXPR;
15         node->data.integer_expr.value = atoi(currentToken.value);
16         advance();
17         return node;
18     }
19
20     // Error
21     fprintf(stderr, "Expected expression\n");
22     exit(1);
23 }
24
25 ASTNode* parsePrintStmt(void) {
26     ASTNode* node = (ASTNode*)malloc(sizeof(ASTNode));
27     node->type = AST_PRINT_STMT;
28
29     expect(TOKEN_PRINT);
30     expect(TOKEN_LPAREN);
31
32     node->data.print_stmt.expr = parseExpr();
33
34     expect(TOKEN_RPAREN);
35     expect(TOKEN_SEMICOLON);
36
37     return node;
```

```
}
```

### Modifikasi Code Generator

Generate code untuk berbagai jenis ekspresi:

Listing 20.19: Code generator untuk ekspresi

```
void generateExpr(ASTNode* expr, FILE* output) {
    switch (expr->type) {
        case AST_STRING_EXPR:
            emit(output, "    lea rdx, [msg]\n");
            emit(output, "    mov r8, msg_len\n");
            break;
        case AST_INTEGER_EXPR:
            // Convert integer to string (simplified)
            emit(output, "    lea rdx, [int_str]\n");
            emit(output, "    mov r8, int_str_len\n");
            break;
        default:
            fprintf(stderr, "Unsupported expression type\n");
            exit(1);
    }
}

void generateCode(ASTNode* ast, FILE* output) {
    // ... setup code ...

    generateExpr(ast->data.print_stmt.expr, output);

    // ... rest of code ...
}
```

## 20.6.2 Error Handling yang Lebih Baik

Saat ini, kompilator langsung exit ketika menemukan error. Kita dapat memperbaikinya dengan:

### Error Recovery

Listing 20.20: Error recovery dalam parser

```
static int errorCount = 0;
static int maxErrors = 10;

static void reportError(const char* message) {
    fprintf(stderr, "Error at line %d, column %d: %s\n",
            currentToken.line, currentToken.column, message);
    errorCount++;

    if (errorCount >= maxErrors) {
        fprintf(stderr, "Too many errors, stopping compilation\n");
```

```
        exit(1);
    }
}

static void synchronize(void) {
    // Skip tokens until we find a statement boundary
    while (currentToken.type != TOKEN_EOF) {
        if (currentToken.type == TOKEN_SEMICOLON) {
            advance();
            return;
        }
        advance();
    }
}
```

#### Error Messages yang Informatif

Tambahkan informasi lebih detail dalam error messages:

Listing 20.21: Error messages yang lebih baik

```
static void expect(TokenType expected) {
    if (currentToken.type != expected) {
        fprintf(stderr, "Error at line %d, column %d: ",
                currentToken.line, currentToken.column);
        fprintf(stderr, "Expected ");
        printTokenType(expected);
        fprintf(stderr, ", but got ");
        printTokenType(currentToken.type);
        fprintf(stderr, "\n");

        // Try to recover
        synchronize();
    } else {
        advance();
    }
}
```

### 20.6.3 Optimasi Dasar

Beberapa optimasi sederhana yang dapat ditambahkan:

#### Constant Folding

Jika ekspresi adalah konstanta, evaluasi pada compile time:

Listing 20.22: Constant folding

```
int evaluateConstant(ASTNode* node) {
    switch (node->type) {
        case AST_INTEGER_EXPR:
            return node->data.integer_expr.value;
        case AST_ADD_EXPR:
```

```
            return evaluateConstant(node->data.add_expr.left) +
                   evaluateConstant(node->data.add_expr.right);
        // ... other cases ...
        default:
            return -1; // Not a constant
    }
}
```

#### Dead Code Elimination

Hapus kode yang tidak pernah dieksekusi (jika menambahkan control flow).

#### Register Allocation

Gunakan register dengan lebih efisien dalam code generation.

### 20.6.4 Menambahkan Fitur Baru

#### Multiple Statements

Support untuk multiple print statements:

```
print("Hello");
print("World");
```

#### Variables

Support untuk variabel sederhana:

```
int x = 42;
print(x);
```

#### Arithmetic Expressions

Support untuk operasi aritmatika:

```
print(10 + 20);
print(5 * 3);
```

#### Comments

Support untuk komentar:

```
// This is a comment
print("Hello"); // Another comment
```

### 20.6.5 Testing Framework

Buat testing framework untuk memudahkan pengujian:

Listing 20.23: test$_framework.c$

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef struct {
    const char* name;
    const char* source;
    int shouldFail;
} TestCase;

void runTest(TestCase* test) {
    printf("Running test: %s\n", test->name);

    // Compile
    // Run
    // Check output

    printf("  PASSED\n");
}

int main() {
    TestCase tests[] = {
        {"Basic print", "print(\"hello\");", 0},
        {"Missing semicolon", "print(\"hello\")", 1},
        // ... more tests ...
    };

    int numTests = sizeof(tests) / sizeof(tests[0]);
    int passed = 0;

    for (int i = 0; i < numTests; i++) {
        runTest(&tests[i]);
        passed++;
    }

    printf("\n%d/%d tests passed\n", passed, numTests);
    return 0;
}
```

### 20.6.6 Documentation

Tambahkan dokumentasi yang lengkap:

- README.md dengan instruksi build dan usage
- Comments dalam kode yang menjelaskan setiap fungsi
- Contoh-contoh penggunaan

- Troubleshooting guide

### 20.6.7 Saran untuk Pengembangan Lebih Lanjut

1. **Type System**: Tambahkan type checking untuk memastikan type safety
2. **Symbol Table**: Implementasikan symbol table untuk variabel dan fungsi
3. **Control Flow**: Tambahkan support untuk if/else, loops
4. **Functions**: Support untuk definisi dan pemanggilan fungsi
5. **Arrays**: Support untuk array dan indexing
6. **Structs**: Support untuk struktur data
7. **Standard Library**: Implementasikan standard library functions
8. **Optimization Passes**: Tambahkan lebih banyak optimasi
9. **Debugging Support**: Tambahkan informasi debugging dalam output
10. **Cross-platform**: Support untuk Linux dan macOS selain Windows

### 20.6.8 Kesimpulan

Kompilator sederhana yang telah kita buat adalah fondasi yang baik untuk memahami proses kompilasi. Dengan extensions dan perbaikan yang telah dijelaskan, kompilator ini dapat berkembang menjadi kompilator yang lebih powerful dan robust.

Penting untuk diingat bahwa:

- Pengembangan kompilator adalah proses iteratif
- Mulai dari yang sederhana, kemudian tambahkan fitur secara bertahap
- Testing sangat penting untuk memastikan kompilator bekerja dengan benar
- Dokumentasi membantu dalam maintenance dan pengembangan lebih lanjut

Dengan memahami dasar-dasar yang telah dipelajari dalam tutorial ini, Anda dapat mengembangkan kompilator yang lebih kompleks sesuai kebutuhan.

# Daftar Pustaka


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

[2] Diznr. *Six Phases of Compiler*. Online educational resource. 2024. URL: https://diznr.com/six-phases-of-compiler-lexical-syntax-semantic-intermediate-code-generation-optimization-code/.

[3] UC San Diego CSE Department. *CSE 231: Compiler Construction / Advanced Compiler Design*. Course materials and syllabus. 2024. URL: https://catalog.ucsd.edu/archive/2024-25/courses/CSE.html.

[4] Northeastern University. *CS 4410/6410: Compiler Design*. Course syllabus and materials, taught by Benjamin Lerner. 2024. URL: https://course.ccs.neu.edu/cs4410sp25/.

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

[6] Johns Hopkins University. *EN.601.428/628: Compilers and Interpreters*. Course syllabus and materials, taught by David Hovemeyer. 2024. URL: https://jhucompilers.github.io/fall2025/syllabus.html.

[7] Aoyama Gakuin University. *Compiler Lecture 5: Lexical Analysis*. Lecture notes. 2024. URL: https://www.sw.it.aoyama.ac.jp/2025/Compiler/lecture5.html.

[8] OpenGenus. *Build Lexer*. Tutorial on hand-written lexers. 2024. URL: https://iq.opengenus.org/build-lexer/.

[9] Wikipedia. *re2c*. Encyclopedia entry. 2024. URL: https://en.wikipedia.org/wiki/Re2c.

[10] John R. Levine. *flex & bison: Text Processing Tools*. O'Reilly Media, 2009.

[11] IT Trip. *C Parser Flex Bison*. Tutorial. 2024. URL: https://en.ittrip.xyz/c-language/c-parser-flex-bison.

[12] GNU Project. *GNU Bison Manual*. Official Bison documentation. 2024. URL: https://www.gnu.org/software/bison/manual/.

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

[14] StudyLib. *Outcomes-Based Education Curricula*. Materials on runtime environment and activation records. 2024. URL: https://studylib.net/doc/14111770/outcomes-based-education-curricula--academic-year-2015-.

[15] University of Oxford. *Compilers Course*. Course materials, Michaelmas Term 2024, taught by Matty Hoban. 2024. URL: https://www.cs.ox.ac.uk/teaching/courses/2024-2025/com/.