

# 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":

“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).

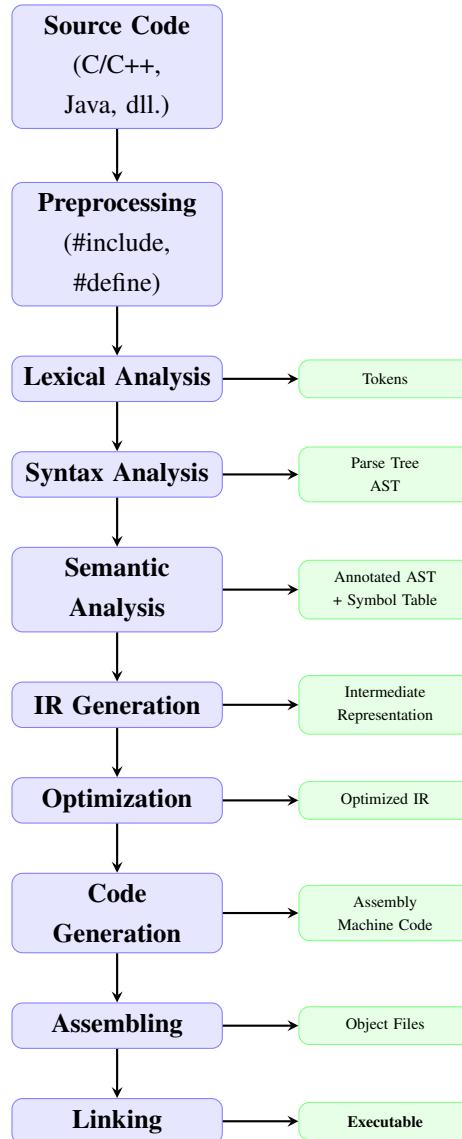
### 1.2.4 Peran Kompilator dalam Pengembangan Software

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):** Lexical, Syntax, dan Semantic Analysis
3. **Sintesis (Back-end):** IR Generation, Optimization, dan 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

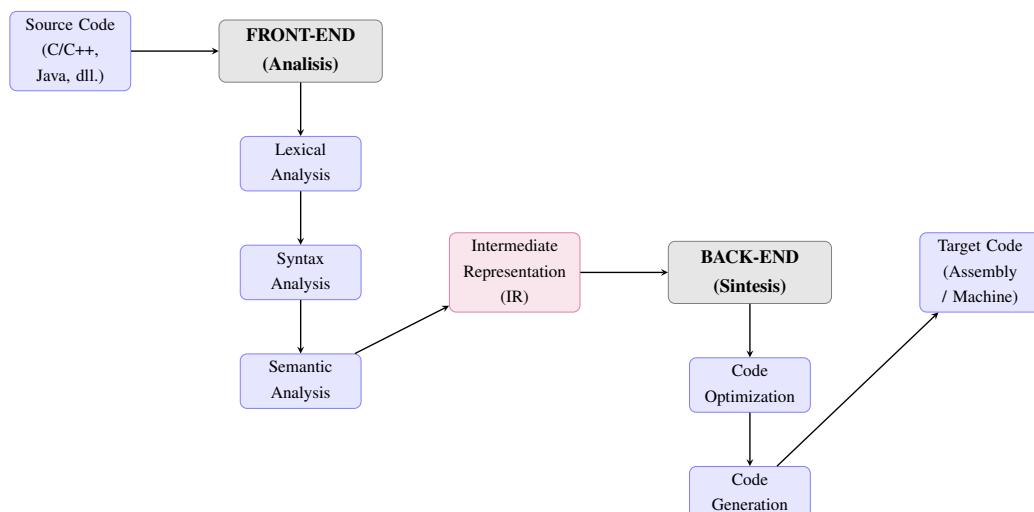
Setelah code generation, assembler mengubah assembly code menjadi object code (file .o atau .obj). Linker kemudian menyatukan:

- 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 1.2 menunjukkan struktur arsitektur kompilator secara keseluruhan.



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

### 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.”?

Fase-fase dalam front-end meliputi:

1. **Lexical Analysis (Tokenization):** Memecah source code menjadi token-token (identifiers, keywords, operators, literals, dll.)
2. **Syntax Analysis (Parsing):** Menganalisis struktur grammar dan membangun parse tree atau Abstract Syntax Tree (AST)
3. **Semantic Analysis:** Memeriksa aturan semantik bahasa, seperti type checking, scope resolution, dan name resolution

### 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. **Intermediate Code Generation:** Mengubah AST menjadi intermediate representation (misalnya three-address code, quadruples, atau bytecode)
2. **Code Optimization:** Mengoptimasi kode intermediate untuk meningkatkan efisiensi tanpa mengubah semantik
3. **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?:

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

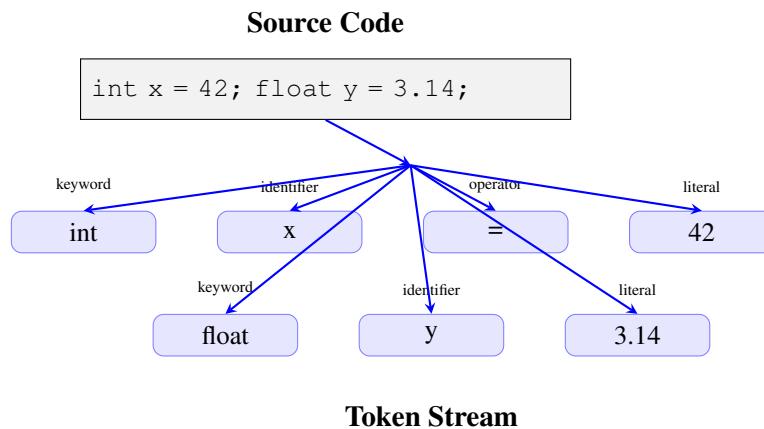
Fase pertama dalam proses kompilasi adalah lexical analysis atau scanning. Fase ini merupakan langkah awal yang mengubah source code (berupa string karakter) menjadi stream token yang bermakna. Tujuan fase ini adalah:

- 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.



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

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

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

Setelah lexical analysis menghasilkan stream token, fase berikutnya adalah 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.

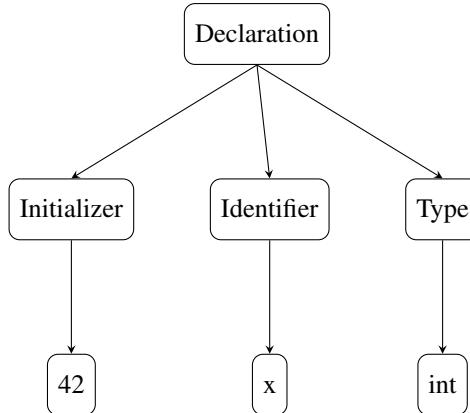
Hasil dari parsing biasanya berupa:

- **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: Semantic Analysis (Analisis Semantik)

Setelah syntax analysis memastikan bahwa struktur program valid secara grammar, 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 dideklarasikan?
- Apakah operasi aritmatika dilakukan pada tipe yang kompatibel?
- Apakah fungsi dipanggil dengan jumlah dan tipe parameter yang benar?

### 1.5.4 Fase 4: Intermediate Code Generation

Setelah semantic analysis selesai dan memastikan program valid secara semantik, kompilator menghasilkan intermediate representation (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)

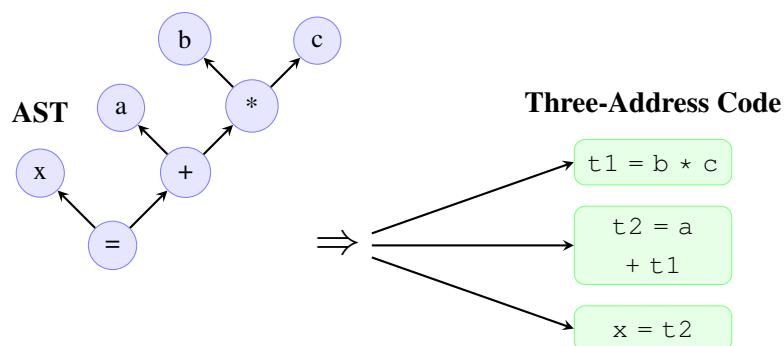
Bentuk IR yang umum digunakan:

- **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: Code Optimization

Setelah IR dihasilkan, fase optimasi 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?, optimasi harus menyeimbangkan antara:

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

### 1.5.6 Fase 6: Code Generation

Fase terakhir dalam proses kompilasi adalah 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
```

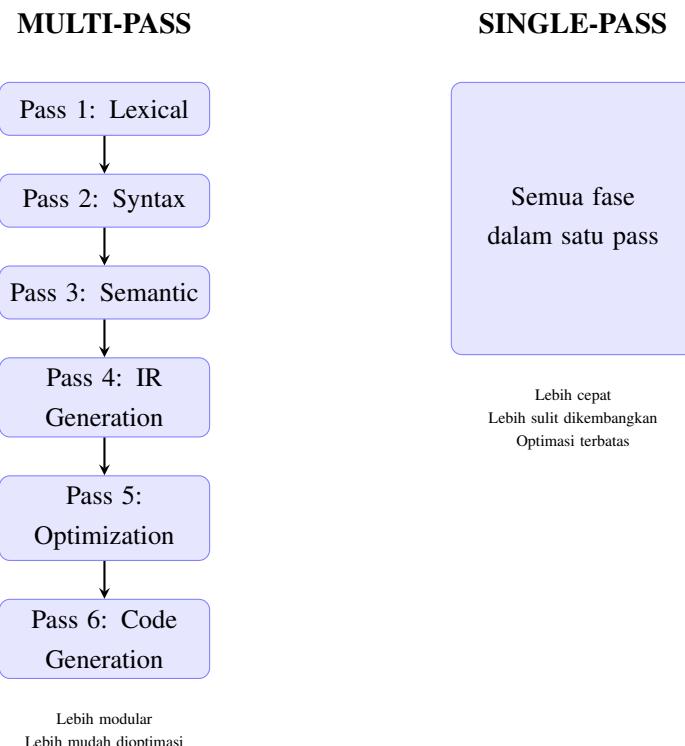
## 1.6 Kompilator Modern: Multi-Pass vs Single-Pass

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

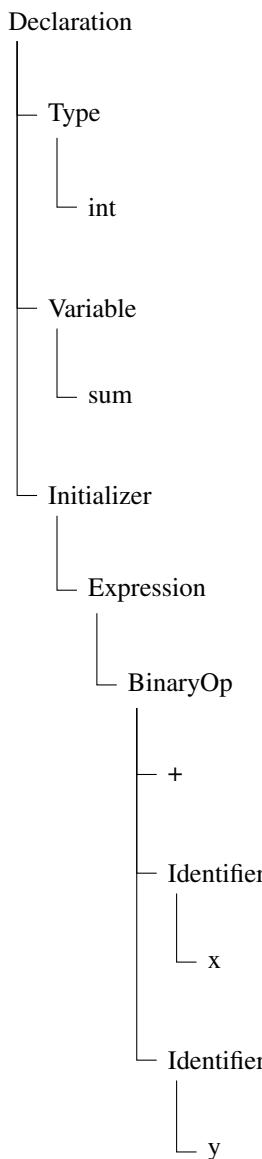
Dilanjutkan pada halaman berikutnya

**Tabel 1.1 – lanjutan dari halaman sebelumnya**

Token	Token Type
;	PUNCTUATION (SEMICOLON)
int	KEYWORD
y	IDENTIFIER
=	OPERATOR (ASSIGN)
20	INTEGER_LITERAL
;	PUNCTUATION (SEMICOLON)
int	KEYWORD
sum	IDENTIFIER
=	OPERATOR (ASSIGN)
x	IDENTIFIER
+	OPERATOR (PLUS)
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.



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

### 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
```

### 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 Kompiletor Sederhana

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

- **TinyCC (TCC)**: Kompiletor 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<sup>1</sup>**: 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 Kesimpulan

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

1. **Definisi Kompiletor**: Kompiletor adalah program yang menerjemahkan source code ke target code melalui beberapa fase yang terstruktur.
2. **Perbedaan dengan Translator Lainnya**: Kompiletor berbeda dengan interpreter dan translator lainnya (assembler, linker, loader, preprocessor, decompiler) dalam cara dan tujuan translasinya.

---

<sup>1</sup><https://github.com/AnjaneyaTripathi/c-compiler>

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

- **Front-end (Analisis):** Lexical Analysis, Syntax Analysis, dan Semantic Analysis
- **Back-end (Sintesis):** Intermediate Code Generation, Code Optimization, dan Code Generation

4. **Enam Fase Utama Kompilasi:** Setiap fase memiliki peran spesifik dalam transformasi source code menjadi executable:

- Lexical Analysis: Tokenization
- Syntax Analysis: Parsing dan pembangunan AST
- Semantic Analysis: Type checking dan scope resolution
- Intermediate Code Generation: Pembuatan IR
- Code Optimization: Optimasi 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. Setiap fase akan dibahas secara lebih mendalam dengan contoh implementasi menggunakan bahasa pemrograman C/C++.

## 1.10 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* ? - Bab 1: Introduction
- **Engineering a Compiler:** Cooper & Torczon (2011) ? - Bab 1: Overview of Compilation
- **UC San Diego CSE 231:** Course materials tentang compiler construction ?

- **Northeastern University CS 4410:** Comprehensive compiler design course ?
- **Johns Hopkins University EN.601.428:** Course tentang compilers dan interpreters ?