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 **??**). 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 !!!");
```

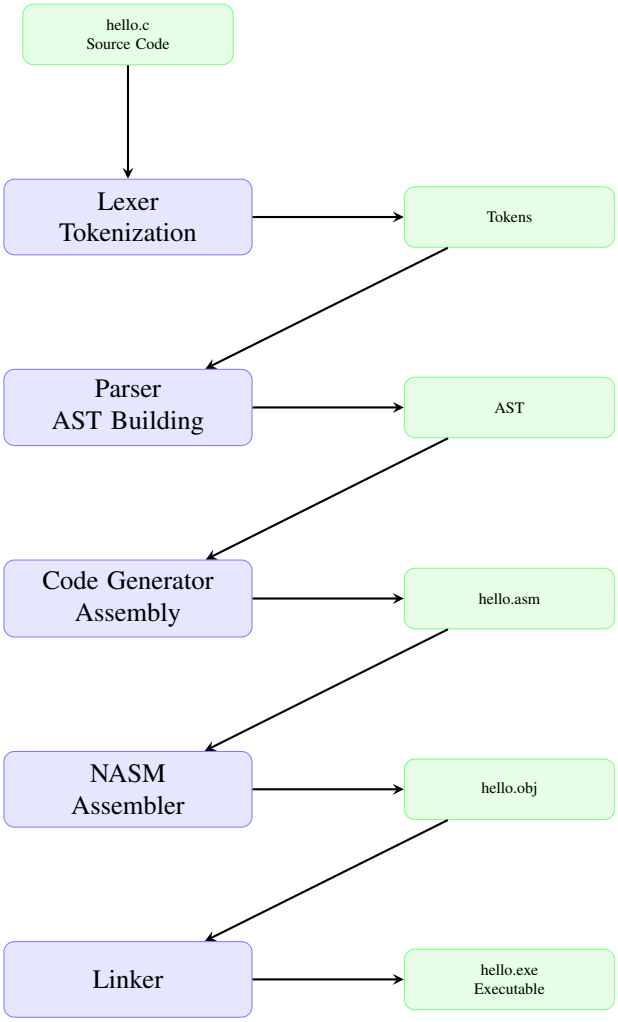
20.1.7 Arsitektur Compiler

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;
```

```
    char* ident = (char*)malloc(len + 1);
    strncpy(ident, source + start, len);
    ident[len] = '\0';

    // Check if it's a keyword
    if (strcmp(ident, "print") == 0) {
        free(ident);
        return makeToken(TOKEN_PRINT, NULL);
    }

    // For now, we only support "print" keyword
    free(ident);
    return makeToken(TOKEN_ERROR, "Unknown identifier");
}

Token nextToken(void) {
    skipWhitespace();

    if (pos >= sourceLen) {
        return makeToken(TOKEN_EOF, NULL);
    }

    char c = source[pos];

    // String literal
    if (c == '"') {
        return scanString();
    }

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

    // Single character tokens
    switch (c) {
        case '(':
            pos++;
            column++;
            return makeToken(TOKEN_LPAREN, NULL);
        case ')':
            pos++;
            column++;
            return makeToken(TOKEN_RPAREN, NULL);
        case ';':
            pos++;
            column++;
            return makeToken(TOKEN_SEMICOLON, NULL);
        default:
            // Unknown character
            char error[64];
            snprintf(error, sizeof(error), "Unexpected character: %c", c)
↪ ;
            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
```

```
41      node->data.print_stmt.string_value =
42          (char*)malloc(strlen(currentToken.value) + 1);
43      strcpy(node->data.print_stmt.string_value, currentToken.value);
44
45      advance(); // Consume STRING token
46
47      // Expect RPAREN
48      expect(TOKEN_RPAREN);
49
50      // Expect SEMICOLON
51      expect(TOKEN_SEMICOLON);
52
53      return node;
54  }
55
56  ASTNode* parse(void) {
57      // Initialize lexer (should be done before calling parse)
58      advance(); // Get first token
59
60      // Parse print statement
61      ASTNode* node = parsePrintStmt();
62
63      // Expect EOF
64      if (currentToken.type != TOKEN_EOF) {
65          fprintf(stderr, "Parse error: Expected EOF\n");
66          exit(1);
67      }
68
69      return node;
70  }
71
72  void freeAST(ASTNode* node) {
73      if (node == NULL) return;
74
75      switch (node->type) {
76          case AST_PRINT_STMT:
77              if (node->data.print_stmt.string_value != NULL) {
78                  free(node->data.print_stmt.string_value);
79              }
80              break;
81      }
82
83      free(node);
84  }
85
86  void printAST(ASTNode* node) {
87      if (node == NULL) return;
88
89      switch (node->type) {
90          case AST_PRINT_STMT:
91              printf("PrintStmt(\"%s\")\n",
92                     node->data.print_stmt.string_value);
93              break;
94      }
```

```
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

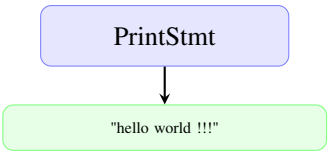
```c
#include "parser.h"
#include "lexer.h"
#include <stdio.h>

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
```

```
8  echo [1/5] Building kompilator...
9  call build.bat
10 if errorlevel 1 (
11     echo ERROR: Failed to build kompilator
12     exit /b 1
13 )
14 echo.
15
16 REM Step 2: Create test file
17 echo [2/5] Creating test file...
18 echo print("hello world !!!"); > hello.c
19 echo Test file created: hello.c
20 echo.
21
22 REM Step 3: Compile with our kompilator
23 echo [3/5] Compiling hello.c to hello.asm...
24 compiler.exe hello.c hello.asm
25 if errorlevel 1 (
26     echo ERROR: Compilation failed
27     exit /b 1
28 )
29 echo Assembly generated: hello.asm
30 echo.
31
32 REM Step 4: Assemble
33 echo [4/5] Assembling hello.asm...
34 nasm -f win64 hello.asm -o hello.obj
35 if errorlevel 1 (
36     echo ERROR: Assembly failed
37     echo Make sure NASM is installed and in PATH
38     exit /b 1
39 )
40 echo Object file created: hello.obj
41 echo.
42
43 REM Step 5: Link
44 echo [5/5] Linking hello.obj...
45 link hello.obj /subsystem:console /entry:_start /out:hello.exe kernel32.
   ↪ lib
46 if errorlevel 1 (
47     echo ERROR: Linking failed
48     echo Make sure Microsoft Linker is available
49     exit /b 1
50 )
51 echo Executable created: hello.exe
52 echo.
53
54 REM Step 6: Run
55 echo ========================================
56 echo   Running hello.exe...
57 echo ========================================
58 hello.exe
59 echo.
60
```

```
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.