BAB 8: PARSER GENERATOR (BISON/YACC) DAN PRAKTIKUM PARSER

Buku Ajar Teknik Kompilasi
Program Studi S1 Teknik Informatika

22 Januari 2026

1 Parser Generator (Bison/Yacc) dan Praktikum Parser

2 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

3 Pengenalan Parser Generator

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."**?**

3.1 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.

4 Struktur File Grammar Bison

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

4.1 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`)

4.2 Bagian 2: Grammar Rules

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

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

4.3 Bagian 3: Auxiliary Code

Bagian ini berisi kode C tambahan seperti:

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

5 Semantic Values dan Semantic Actions

5.1 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)**

5.2 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 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.

6 Typing Semantic Values dengan %union

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

Listing 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
```

7 Contoh Lengkap: Calculator dengan Bison

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

7.1 File Lexer (calc.l)

Listing 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; }
```

7.2 File Parser (calc.y)

Listing 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;
}
```

7.3 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 Integrasi Flex dan Bison

8.1 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.

8.2 Contoh Integrasi

Listing 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
%}
```

9 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.

9.1 Struktur AST Node

Pertama, kita definisikan struktur untuk AST node:

Listing 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;
}
```

9.2 Grammar dengan AST Building

Listing 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; }
  ;
```

10 Precedence dan Associativity

Untuk menangani precedence dan associativity tanpa membuat grammar yang ambigu, Bison menyediakan deklarasi khusus:

10.1 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.

10.2 Contoh Penggunaan

Listing 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 $2^{34} akandi - parsesebagai$

10.3 Error Token

Bison menyediakan token khusus error untuk error recovery. Ketika parser menemukan error, ia dapat mencoba recovery dengan menggunakan produksi yang mengandung error.

10.4 Contoh Error Recovery

Listing 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");
      }
  ;
```

10.5 Fungsi yyerror

Fungsi yyerror() dipanggil ketika terjadi syntax error. Kita dapat mengimplementas untuk memberikan pesan error yang informatif:

Listing 10: Implementasi yyerror yang lebih informatif

```
void yyerror(const char *s) {
    extern int yylineno;
    fprintf(stderr, "Error at line %d: %s\n", yylineno, s);
}
```

10.6 Location Tracking

Untuk memberikan informasi lokasi yang lebih akurat, kita dapat menggunakan location tracking:

Listing 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;
}
```

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 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.

12 Contoh Praktis: Parser untuk Subset C

Mari kita buat parser yang lebih kompleks untuk subset bahasa C yang mendukung:

- Deklarasi variabel (int, float)

- Assignment statements
- Ekspresi aritmatika
- Print statements

12.1 File Lexer (simplec.l)

Listing 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; }
```

12.2 File Parser (simplec.y)

Listing 14: Parser untuk subset C

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

```

```
6  extern int yylex();
7  void yyerror(const char *s);
8
9  // Symbol table sederhana
10 typedef struct {
11     char *name;
12     int type;  // 0 = int, 1 = float
13     union {
14         int intval;
15         double dval;
16     } value;
17 } Symbol;
18
19 Symbol symbols[100];
20 int sym_count = 0;
21 %}
22
23 %union {
24     int intval;
25     double dval;
26     char *strval;
27 }
28
29 %token <intval> NUMBER
30 %token <dval> FLOAT_LITERAL
31 %token <strval> IDENTIFIER
32 %token INT FLOAT PRINT ASSIGN SEMICOLON
33 %token PLUS MINUS MULTIPLY DIVIDE LPAREN RPAREN
34
35 %left PLUS MINUS
36 %left MULTIPLY DIVIDE
37
38 %type <dval> expr
39
40 %%
41 program:
42     /* empty */
43   | program statement
44   ;
45
46 statement:
47     declaration SEMICOLON
48   | assignment SEMICOLON
49   | print_stmt SEMICOLON
50   ;
51
52 declaration:
53     INT IDENTIFIER {
54         // Add to symbol table
```

```
55          symbols[sym_count].name = strdup($2);
56          symbols[sym_count].type = 0;
57          sym_count++;
58          printf("Declared int variable: %s\n", $2);
59      }
60    | FLOAT IDENTIFIER {
61          symbols[sym_count].name = strdup($2);
62          symbols[sym_count].type = 1;
63          sym_count++;
64          printf("Declared float variable: %s\n", $2);
65      }
66    ;
67
68  assignment:
69      IDENTIFIER ASSIGN expr {
70          // Find variable in symbol table and assign
71          printf("Assigning %g to %s\n", $3, $1);
72      }
73    ;
74
75  print_stmt:
76      PRINT expr {
77          printf("Print: %g\n", $2);
78      }
79    ;
80
81  expr:
82      NUMBER { $$ = (double)$1; }
83    | FLOAT_LITERAL { $$ = $1; }
84    | IDENTIFIER {
85          // Lookup in symbol table
86          $$ = 0.0; // Simplified
87      }
88    | expr PLUS expr { $$ = $1 + $3; }
89    | 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
```

```c
void yyerror(const char *s) {
    fprintf(stderr, "Syntax error: %s\n", s);
}

int main() {
    printf("Simple C Parser - Enter statements (Ctrl+D to exit)\n");
    yyparse();
    return 0;
}
```

13 Best Practices dan Tips

13.1 Organisasi File

- Pisahkan definisi AST node ke file header terpisah
- Gunakan Makefile untuk mengotomatisasi kompilasi
- Dokumentasikan grammar dengan baik

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

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

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.

15 Latihan

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?

16 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* ? - Buku lengkap tentang Flex dan Bison
- Dragon Book: Aho, Lam, Sethi, & Ullman (2006). *Compilers: Principles, Techniques, and Tools* ? - 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 ?
- Engineering a Compiler: Cooper & Torczon (2011) ? - Bab tentang parser generators