Bab 1

Bottom-Up Parsing, LR Parser, dan Parser Generator

1.1 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

1.2 Pendahuluan

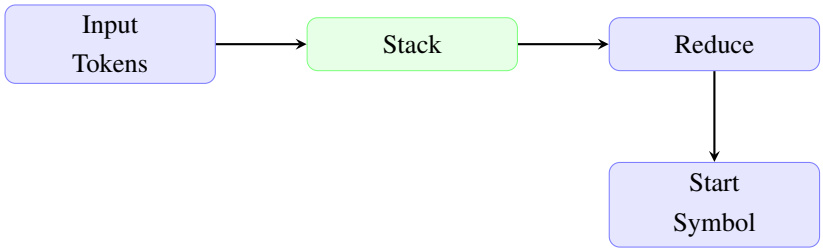
Setelah mempelajari top-down parsing pada bab sebelumnya, kita sekarang akan mempelajari pendekatan alternatif yang lebih powerful: bottom-up parsing. Menurut sumber terbuka:

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

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 1.1 menunjukkan alur bottom-up parsing.



Gambar 1.1: Alur bottom-up parsing

1.3 Konsep Bottom-Up Parsing

1.3.1 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

1.3.2 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]

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

Contoh: Jika kita memiliki grammar:

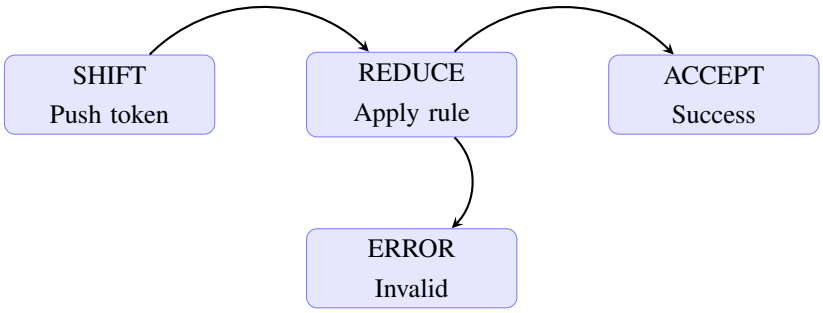
```
E -> E + T | T
T -> T * F | F
F -> ( E ) | id
```

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).

1.4 Shift-Reduce Parsing

1.4.1 Konsep Shift-Reduce

Gambar 1.2 menunjukkan empat operasi dasar dalam shift-reduce parsing.



Gambar 1.2: 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]

1.4.2 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]`.

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

1.4.3 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.

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]`.

1.4.4 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.

1.4.5 Operasi Error

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

1.4.6 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 1.1: Contoh shift-reduce parsing untuk `id + id`

1.5 LR Parsers

1.5.1 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

1.5.2 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

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

- 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

Menurut GeeksforGeeks:

> "SLR(1) uses LR(0) item sets, and reduction is allowed on lookahead symbols in Follow(A) for production A -> α when the item [A -> α •] 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:

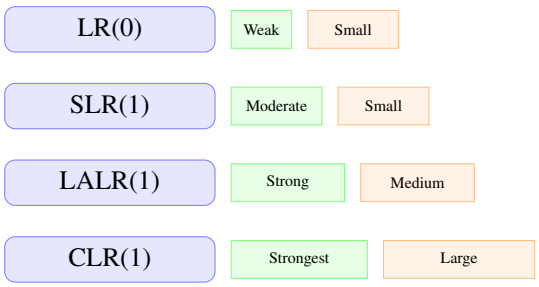
- Merge states dari CLR(1) yang memiliki LR(0) core yang sama
- Menggabungkan lookahead sets dari states yang di-merge
- Jumlah states sama atau mendekati SLR(1)
- Lebih powerful daripada SLR(1), hampir sekuat CLR(1)
- Digunakan oleh Yacc dan Bison (parser generator populer)

Menurut GeeksforGeeks:

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

> "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 1.3 menunjukkan perbandingan visual varian LR parser.



Gambar 1.3: Perbandingan varian LR parser: power vs table size

1.5.3 Perbandingan LR Parser Variants

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 1.2: Perbandingan varian LR parser

1.6 Konstruksi LR Parsing Table

1.6.1 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.

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

1.6.2 LR Items

LR item adalah production dengan dot (•) yang menandai posisi parsing saat ini. Format: `A -> α • β`

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

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.

1.6.3 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

1.6.4 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

1.6.5 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

1.6.6 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`

1.6.7 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

1.7 GLR Parsing (Generalized LR)

1.7.1 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.

1.7.2 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

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

1.8 Parser Generator: Bison dan Yacc

1.8.1 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."**?**

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

1.8.2 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.

1.8.3 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

1.8.4 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 */
```

1.8.5 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
```

1.8.6 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

1.8.7 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

1.9 Perbandingan Top-Down vs Bottom-Up Parsing

1.9.1 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 1.3: Perbandingan top-down dan bottom-up parsing

1.9.2 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

1.10 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.

1.11 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* **?** - Bab 4: Syntax-Directed Translation, Bab 5: Bottom-Up Parsers
- **Engineering a Compiler**: Cooper & Torczon (2011) **?** - Bab 3: Scanners and Parsers
- **flex & bison**: Levine (2009) **?** - Buku lengkap tentang Flex dan Bison
- **GeeksforGeeks**: Tutorial tentang shift-reduce parsing dan LR parsers[7]
- **IT Trip**: Tutorial tentang integrasi Flex dan Bison **?**
- **UC San Diego CSE 231**: Course materials tentang parser construction **?**
- **Northeastern University CS 4410**: Comprehensive compiler design course **?**

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