📘 Bottom-Up Parser — Full Topic Breakdown

**1️⃣ Syntax Analysis (Parsing)**

**🔹 Definition**

Syntax Analysis (Parsing) is the **second phase** of the compiler after lexical analysis. It checks whether the **token sequence** follows the grammatical structure of the language.

**🔹 Output:**

* **Parse Tree** or **Syntax Tree**
* Detects **syntax errors**

**🔹 Types:**

* **Top-Down Parsing** (Predictive, Recursive Descent)
* **Bottom-Up Parsing** (Shift-Reduce)

**2️⃣ Bottom-Up Parser**

**🔹 Definition:**

Parses input **from tokens to the start symbol**, constructing a parse tree **bottom-up**.

**🔹 Main Steps:**

1. **Shift**: Push the input symbol onto the stack
2. **Reduce**: Replace matching RHS of a production with its LHS

**🔹 Uses:**

* Handles **left-recursive grammars**
* Used in tools like **YACC**, **Bison**

**3️⃣ Bottom-Up Parsing Handle**

**🔹 Handle:**

A **handle** is a **substring** matching the **RHS of a production** rule that can be **reduced** to its LHS.

**🔹 Handle Pruning:**

Process of reducing a handle at each step to ultimately derive the **start symbol**.

Example:  
Given: id + id \* id  
Handles → id, then id \* id, then id + T

**4️⃣ Types of Bottom-Up Parsers**

|  |  |
| --- | --- |
| **Type** | **Description** |
| **Operator Precedence** | Based on precedence of operators |
| **LR Parsers** | Most powerful class of bottom-up parsers |

**5️⃣ Operator Precedence Parsing**

**🔹 Concept:**

Uses **precedence rules** of operators (+, \*, /, etc.) to decide shift or reduce.

**🔹 Requirements:**

* No ε-productions
* No adjacent non-terminals

**🔹 Parsing Table:**

Built using precedence relations:

* < → lower precedence
* = → same precedence
* > → higher precedence

**6️⃣ LR Grammar**

**🔹 Definition:**

A **context-free grammar** is called an **LR grammar** if it can be parsed **Left-to-right** with a **Rightmost derivation** in reverse using an LR parser.

**🔹 Conditions:**

* Deterministic
* Unambiguous

**7️⃣ LR Parsers**

**🔹 Components:**

1. **Input Buffer**
2. **Stack**
3. **Parsing Table (ACTION + GOTO)**
4. **Driver Program**

**🔹 Table:**

|  |  |
| --- | --- |
| ACTION | Tells shift/reduce/accept |
| GOTO | Tells which state to go to after reduction |

**8️⃣ LR Parsing Algorithm**

**🔹 Steps:**

1. Initialize stack with state 0
2. Read symbol from input
3. Use ACTION table to determine:
   * **Shift** → push symbol and new state
   * **Reduce** → pop stack, push non-terminal, go to new state
   * **Accept** → success
   * **Error** → failure
4. Repeat until accepted or rejected

**9️⃣ Construction of the LR Parsing Table**

**🔹 Steps:**

1. **Augment grammar** with new start symbol: S' → S
2. Compute **Canonical collection of LR items**
3. Construct **DFA** of item sets
4. Build **ACTION** and **GOTO** tables using:
   * Shifts
   * Reductions
   * Accept actions

**🔟 LR(0) Parser**

**🔹 No Lookahead**

* Items: A → α.β (dot indicates parser progress)
* Conflicts are common (no lookahead to resolve ambiguity)

**🔹 Steps:**

1. Augment grammar
2. Create item sets (I0, I1...)
3. Build parsing table

**🔹 Limitation:**

* **Cannot handle all deterministic grammars**

**1️⃣1️⃣ SLR(1) Parser (Simple LR)**

**🔹 Uses 1-token lookahead**

* Reduces conflicts seen in LR(0)

**🔹 Adds FOLLOW(A) to guide reductions**

**🔹 Construction:**

* Same LR(0) items
* Add **lookahead** using FOLLOW sets
* Modify ACTION table accordingly

**1️⃣2️⃣ Canonical LR(1) Parser (CLR(1))**

**🔹 Full LR parser**

* **Most powerful** variant
* Each item has a lookahead

Example:

csharp

CopyEdit

[A → α.β, a] → Reduce only if next symbol is `a`

**🔹 Advantage:**

* Eliminates shift/reduce conflicts
* Parses **almost all programming languages**

**🔹 Disadvantage:**

* **Very large parsing tables**

**1️⃣3️⃣ LALR(1) Parser (LookAhead LR)**

**🔹 Merges similar LR(1) states:**

* Combines states with identical LR(0) cores
* Reduces size of parsing table

**🔹 Used in:**

* **YACC**
* **Bison**

**🔹 Compromise:**

* Power of CLR(1)
* Size of SLR(1)

**1️⃣4️⃣ Comparison: Top-Down vs Bottom-Up Parsing**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Top-Down** | **Bottom-Up** |
| Start Symbol | Start → leaves | Leaves → Start |
| Grammar Type | LL | LR |
| Handles Left Rec | ❌ | ✅ |
| Uses Stack | ✅ | ✅ |
| Table Used | Predictive | ACTION/GOTO |

**1️⃣5️⃣ Error Recovery in LR Parsing**

**🔹 Techniques:**

1. **Panic Mode**  
   Skip tokens until a synchronizing token is found.
2. **Phrase-Level Recovery**  
   Insert/delete tokens to make valid phrases.
3. **Error Productions**  
   Add explicit productions for common mistakes.
4. **Global Correction**  
   Attempt least-cost fix (complex and rare)

**1️⃣6️⃣ Parser Construction with Ambiguous Grammars**

**🔹 Ambiguous Grammar:**

Grammar that has **more than one parse tree** for the same input.

**🔹 Handling:**

* Refactor grammar (eliminate ambiguity)
* Use precedence and associativity rules
* Introduce non-terminals for structure

**1️⃣7️⃣ Operator Precedence Parser**

* Special case of bottom-up parser
* Uses precedence table
* Doesn’t use full stack-based reduction

Example:

cpp

CopyEdit

+ < \* // + has lower precedence

\* > + // \* has higher precedence

Used in **expression evaluation**, calculators.

**1️⃣8️⃣ C Code to CFG Grammar**

**🔹 Goal:**

Translate constructs in C into context-free grammar rules.

**🔹 Examples:**

**If-Else:**

cfg

CopyEdit

stmt → if (expr) stmt else stmt

| if (expr) stmt

**While:**

cfg

CopyEdit

stmt → while (expr) stmt

**Assignment:**

cfg

CopyEdit

stmt → id = expr ;

**Expressions:**

cfg

CopyEdit

expr → expr + term | term

term → term \* factor | factor

factor → id | num | (expr)

**✅ Summary**

|  |  |
| --- | --- |
| **Topic** | **Covered Concepts** |
| Syntax Analysis | Parse trees, token validation |
| Bottom-Up Parser | Shift-reduce, handles |
| Types of Bottom-Up Parsers | Operator precedence, LR types |
| Operator Precedence Parsing | Parsing using precedence rules |
| LR Grammar | Deterministic grammar rules |
| LR Parsers | LR(0), SLR(1), CLR(1), LALR(1) |
| LR Parsing Algorithm | Shift, reduce, accept |
| LR Parsing Table Construction | DFA of item sets, ACTION/GOTO |
| LR(0)/SLR(1)/CLR(1)/LALR(1) | Variants with different power and table size |
| Top-Down vs Bottom-Up | Comparison of strategies |
| Error Recovery | Panic mode, phrase-level, error productions |
| Ambiguous Grammar Parsing | Solutions like grammar refactoring and precedence rules |
| Operator Precedence Parser | Simple parsing with tables |
| C to CFG Grammar | Translation of C syntax to CFG |

CS4031: Compiler Construction - Lecture 6

**Instructor:** Mahzaib Younas  
**Department of Computer Science, FAST NUCES CFD**

**Bottom-Up Parsing**

**Definition:** Bottom-up parsing constructs a parse tree starting from the input tokens and works upwards toward the start symbol.

**Key Concepts:**

* Start with tokens (terminals)
* **Shift:** Push the current input symbol to stack
* **Reduce:** Replace a handle (RHS of a production) on the stack with the LHS (non-terminal)

**Example:**

**Grammar:**

E → T | T \* F

T → id

F → T | id

**Input:** id \* id

Steps:

Input: id \* id

Stack operations:

Shift id → Reduce to F

Reduce F to T

Shift \*

Shift id → Reduce to F

Reduce F to T

Reduce T \* F to E

**Types of Bottom-Up Parsers**

* **LR(0)**
* **SLR(1)**
* **LALR(1)**
* **CLR(1) [Canonical LR(1)]**

**LR(0) Parser**

**LR:**

* L = Left-to-right scanning
* R = Rightmost derivation in reverse

**Steps:**

1. Augment the grammar (Add S' → .S)
2. Create LR(0) items using . (dot) notation
3. Build item sets (I0, I1, ..., In)
4. Construct parsing table (ACTION and GOTO)

**Example Grammar:**

E → T + E | T

T → id

**Augmented Grammar:**

E' → .E

E → .T + E | .T

T → .id

**Parsing Table Structure:**

State | id | + | $ | E | T

--------------------------------

0 | S3 | | | 1 | 2

1 | |S4 |acc| |

2 | |r2 |r2 | |

3 | |r3 |r3 | |

4 | S3 | | | 5 | 2

5 | |r1 |r1 | |

**LR(0) Example 2:**

S → (L) | a

L → S | L,S

**Augmented Grammar:**

S' → .S

S → .(L) | .a

L → .S | .L,S

**CLR(1) Parser (Canonical LR)**

* Enhances LR(0) by adding **lookahead symbols**.
* Uses **LR(1) items**, denoted like: A → α.β, a
* Helps resolve ambiguities by providing extra information for reductions

**Example:**

S → AA

A → aA | b

**Augmented Grammar:**

S' → .S, $

S → .AA, $

A → .aA, a/b

A → .b, a/b

**Parsing String: ccdd$ for Grammar:**

E → BB

B → cB | d

**Augmented Grammar:**

E' → .E

E → .BB

B → .cB | .d

**Item Sets:** I0, I1, ..., I6

**Parsing Table:**

Action Table:

State | c | d | $ | B | E

--------------------------

0 |S3|S4| | 2 | 1

1 | | |acc| |

2 |S3|S4| | 5 |

3 |S3|S4| | 6 |

4 |r3|r3|r3| |

5 |r1|r1|r1| |

6 |r2|r2|r2| |

**Stack Trace:**

$0 ccdd$

Shift c → $0 c3 cdd$

Shift c → $0 c3 c3 dd$

Shift d → $0 c3 c3 d4 d$

Reduce B → d

Reduce B → cB

Reduce B → cB

...

Accept

**Pros and Cons of LR(0)**

**Advantages:**

* Simpler to construct than CLR, LALR
* Unique parsing action in each table row

**Disadvantages:**

* Can parse only a small set of grammars
* No lookahead used; may reduce prematurely

**Summary:**

* Bottom-up parsers use **shift-reduce** mechanism.
* **LR(0)** is foundational and builds towards **SLR**, **LALR**, and **CLR**.
* CLR(1) is most powerful due to lookahead capability.

CS4031 Compiler Construction Lecture 7 –

**📘 CS4031 – Lecture 7: SLR(1) and CLR(1) Parsers**

**🔷 SLR(1) Parser (Simple LR Parser)**

**➤ What is SLR(1)?**

* **SLR** stands for **Simple LR**.
* The **“1”** indicates **1 symbol lookahead** is used.
* It is an **improvement over LR(0)** parsers because:
  + LR(0) blindly reduces using all terminals.
  + SLR(1) uses **FOLLOW sets** to **restrict reduce actions** only where appropriate.

**🔹 Key Concepts:**

|  |  |
| --- | --- |
| Term | Meaning |
| LR(0) Items | Productions with a dot . showing how much has been seen |
| SLR(1) | Uses LR(0) states but makes reduce decisions using FOLLOW sets |
| Lookahead | The next input symbol used to decide parser action |

**🔹 DFA Construction (States)**

* DFA = Deterministic Finite Automaton built using LR(0) items.
* A **final item** is one with dot at the end: A → α.

**🔹 SLR(1) Parsing Table Construction**

**Step-by-step:**

1. **Construct DFA** from LR(0) items.
2. **For each final item in a state** like A → α., place a **reduce action** r in that row **only** under columns from **FOLLOW(A)**.
3. **For non-final items** like A → α.Bβ, use **goto transitions** to compute **shift or goto** entries.

**🔹 Important Distinction:**

|  |  |
| --- | --- |
| Parser | Reduce Entries Placed In Columns |
| LR(0) | ALL terminal columns |
| SLR(1) | ONLY FOLLOW(A) columns |

**🧪 Example Grammar:**

r

CopyEdit

E → T + E | T

T → a

**📌 Follow Sets:**

* **Follow(E)** = { $ }
* **Follow(T)** = { +, $ }

**📋 Partial Parsing Table**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| State | a | + | $ | E | T |
| I0 | s2 |  |  | 1 | 3 |
| I1 |  | s4 | acc |  |  |
| I2 |  | r2 | r2 |  |  |
| I3 | s2 |  |  |  | 5 |
| ... | ... | ... | ... | ... | ... |

**📘 Example Grammar 2:**

less

CopyEdit

S → A a A b | B b B a

A → ε

B → ε

To check **if suitable for SLR(1)**:

* Construct DFA (states).
* Check FOLLOW sets.
* Build parsing table.
* If **no shift-reduce or reduce-reduce conflict**, then it **is SLR(1)**.

**🔎 SLR(1) Grammar with ε-productions**

less

CopyEdit

S → cAd

A → ab | ε

Augmented Grammar:

css

CopyEdit

S’ → S

S → cAd

A → ab

A → ε

**🧮 Parsing Table: (Input = ced$)**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| State | a | b | c | d | e | $ | A | S |
| I0 |  |  | S2 |  |  |  |  | 1 |
| I2 | S4 | S5 |  |  |  |  | 3 |  |
| I3 |  |  |  | S6 |  |  |  |  |
| I4 |  |  |  |  |  |  | r1 |  |
| I5 |  |  |  |  |  |  | r2 |  |
| I6 |  |  |  |  |  |  | r3 |  |

**Action for "ced$":**

|  |  |  |
| --- | --- | --- |
| Stack | Input | Action |
| $0 | ced$ | shift s2 |
| $0c2 | ed$ | shift s4... and so on |

**🔶 Canonical LR(1) Parser (CLR(1))**

**➤ What is CLR(1)?**

* **CLR(1)** = **Canonical LR with 1 symbol lookahead**.
* Much **more powerful** than SLR(1).
* Eliminates many **conflicts** that SLR(1) cannot.
* Uses **LR(1) Items**, which are more detailed than LR(0).

**🔹 Why CLR(1) Instead of SLR(1)?**

* SLR(1) uses **FOLLOW(A)** set → this can cause **shift-reduce conflicts**.
* CLR(1) uses **exact lookahead tokens** for each item to resolve conflicts properly.

**🔍 Example:**

SLR(1) might have:

css

CopyEdit

A → α. reduce on all FOLLOW(A)

CLR(1) has:

csharp

CopyEdit

[A → α., a] only reduce if lookahead is `a`

**🔹 Steps to Construct CLR(1) Parser:**

1. **Augment Grammar**: Add S’ → .S
2. **Construct LR(1) Items**:
   * Use Closure() and Goto() with lookahead.
3. **Build DFA with LR(1) Items**.
4. **Build Parsing Table**:
   * ACTION and GOTO.
   * Use lookahead for reduce.

**📘 Example Grammar:**

less

CopyEdit

S → AA

A → aA | b

Augmented:

less

CopyEdit

S’ → .S

S → .AA

A → .aA | .b

**📋 Sample LR(1) Items:**

csharp

CopyEdit

[A → .aA, a]

[A → .b, b]

Each state carries **lookahead** so reduce actions are more accurate than SLR.

**🧪 Example: Suitability Check for CLR(1)**

less

CopyEdit

S → A a A b | B b B a

A → ε

B → ε

* Construct LR(1) items
* Build DFA
* If **no conflicts** arise, the grammar is **CLR(1)**

Even if it's **not SLR(1)** (due to conflicts in FOLLOW), it may still be **CLR(1)** due to exact lookahead filtering.

**✅ Summary Comparison**

|  |  |  |
| --- | --- | --- |
| Feature | SLR(1) | CLR(1) (Canonical LR) |
| Lookahead | Uses FOLLOW sets | Uses specific lookaheads |
| States | Fewer | More |
| Reduce logic | Based on FOLLOW | Based on LR(1) lookahead |
| Conflicts | Higher chance | Lower chance |
| Table Size | Smaller | Larger |
| Accuracy | Less powerful | More powerful |

**📚 Important Terms**

|  |  |
| --- | --- |
| Term | Meaning |
| LR(0) item | Production with dot position |
| LR(1) item | LR(0) item + lookahead symbol |
| Closure() | Expand current state by ε-rules |
| Goto() | Transition to next state by shifting a symbol |
| FOLLOW(A) | Set of terminals that can appear after A |

📘 Lecture 8: Operator Precedence Parser & Symbol Table

**🔷 Operator Precedence Parser**

**➤ What is it?**

An Operator Precedence Parser is a **bottom-up shift-reduce parser** used to manually construct parsers for **a limited class of grammars**.

It relies on defining **precedence relationships** between terminal symbols using three relations:

* a ⋖ b (a has lower precedence than b → shift)
* a ≐ b (equal precedence → shift)
* a ⋗ b (a has higher precedence than b → reduce)

**➤ When Is It Used?**

Only when the grammar is:

* Free from **epsilon (ε) productions**
* Has **no adjacent non-terminals** on the RHS of any production (called an **Operator Grammar**)

**✅ Rules for Operator Precedence Grammar:**

* No epsilon ε in productions.
* No two adjacent non-terminals on RHS.

❌ Not Suitable:

mathematica

CopyEdit

E → E B E

B → + | - | =

✅ Suitable:

mathematica

CopyEdit

E → E + E | E \* E | E = E | id

**🔄 Shift and Reduce in Operator Precedence Parsing**

Let:

* a = top of stack (last terminal)
* b = current input symbol (lookahead)

Then:

* **Shift** if a ⋖ b or a ≐ b
* **Reduce** if a ⋗ b
* Accept if both a and b are $ (end of string)

**🔺 Operator Precedence Levels**

From **low to high**:

+, - < \*, / < id (terminal)

**🔤 Example: Precedence Relations in Input**

For grammar:

mathematica

CopyEdit

E → E + E | E \* E | id

Parsing:

bash

CopyEdit

id + id \* id

Inserting precedence:

shell

CopyEdit

$ < id > + < id > \* < id > $

Parsing uses the relations to decide when to shift or reduce.

**🔧 Constructing Operator Precedence Table**

Each terminal is placed along both rows and columns to define the ⋖, ⋗, or ≐ relationship.

**🧪 Example Grammar to Transform**

Original:

less

CopyEdit

S → AB

A → c | d

B → aAB | b

Converted Operator Precedence Grammar: must remove any ε, recursion, or adjacent non-terminals.

**✍️ Example 2: Parse a + b \* c**

Assuming:

mathematica

CopyEdit

E → E + E | E \* E | id

Precedence:

* + < \* (shift)
* \* > id (reduce)

**Parsing Trace:**

1. Shift a
2. Shift +
3. Shift b
4. Shift \*
5. Shift c
6. Reduce b \* c → E
7. Reduce a + E → E

**📊 Comparison: Top-Down vs Bottom-Up Parsers**

| **Feature** | **Top-Down Parser** | **Bottom-Up Parser** |
| --- | --- | --- |
| Strategy | Predictive, expands from S | Reductive, builds from input |
| Grammar Restriction | No left recursion | Can handle more grammars |
| Table | LL(1) | LR, Operator Precedence |
| Parser Type | Recursive Descent, LL(1) | LR(0), SLR, LALR, CLR |

**🗂️ Symbol Table**

**🔍 What is a Symbol Table?**

It’s a **data structure used by the compiler** to **store all identifiers** (names) and their associated properties.

**📚 Stored Items:**

* Keywords
* Data types
* Operators
* Functions
* Variables
* Constants
* Literals

**📥 Contents of Each Entry:**

1. Lexeme (name)
2. Corresponding token
3. Type (int, float, etc.)
4. Semantic info (function, variable, etc.)
5. Pointer to other entries if needed (e.g., function scopes)

**🔄 Compiler Phases Using Symbol Table**

|  |  |
| --- | --- |
| Phase | Usage Example |
| Initialization | Store keywords, operators |
| Scanner | Store user-defined names |
| Syntax Analyzer | Store token info |
| Semantic Analyzer | Check type, compatibility |
| Intermediate Code | Store variable location |
| Target Code | Track final addresses |

**🧠 Symbol Table Roles**

* Check whether a variable is **already defined**
* Maintain **scoping rules**
* Assist in **type checking**
* Store **storage locations**

**⚙️ Common Symbol Table Operations**

|  |  |
| --- | --- |
| Operation | Description |
| Insert | Add a symbol |
| Search | Find a symbol |
| Delete | Remove a symbol (on scope end) |

**🧰 Symbol Table Implementations**

|  |  |
| --- | --- |
| Type | Performance |
| Linear list | Simple, slow |
| Ordered list | Faster than linear |
| Tree-based (e.g., BST) | Balanced, scalable |
| Hash table | Fastest, preferred |

✅ Hash tables are most commonly used in real compilers.

**📦 What Is Stored in Symbol Table?**

* For type names: mapping (e.g., typedef int\* mytype)
* For variables: type, dimensions, scope info
* For constants: value and type
* For functions: parameter list, return type, call info

**🔄 Updating Symbol Table**

Symbol table is updated when:

* A new symbol is declared
* A new property is added to a symbol
* A new scope starts or ends

**🧪 C-Like Example for Symbol Table**

typedef struct {

char \*name;

int value;

} Symbol;

Symbol symbolTable[MAX\_SYMBOLS];

int symbolCount = 0;

int lookupSymbol(char \*name) {

for (int i = 0; i < symbolCount; i++) {

if (strcmp(symbolTable[i].name, name) == 0)

return i;

}

return -1;

}

void addSymbol(char \*name, int value) {

if (lookupSymbol(name) == -1) {

symbolTable[symbolCount].name = strdup(name);

symbolTable[symbolCount].value = value;

symbolCount++;

}

}

**🧱 Organization in Non-Block Structured Languages**

* Only one flat module, no submodules
* Performance depends on search length
* Each lookup involves a **search argument** (symbol name)
* Each variable has a **symbol table record**

**📂 Symbol Table Structures**

1. **Unordered Table** – Simple list
2. **Ordered Table** – Sorted list
3. **Tree-Based Table** – Balanced search
4. **Hash Table** – Fastest access

**✅ Summary**

**🔑 Operator Precedence Parser:**

* Requires operator grammar (no ε, no adjacent non-terminals)
* Shift-reduce based on precedence rules
* Used for small grammars

**🔑 Symbol Table:**

* Core data structure for all phases
* Stores every identifier’s properties
* Must support dynamic insert/delete/search
* Typically implemented using hash tables