Compiler Phases

A compiler translates a **high-level language** program into **machine code** (or intermediate code) in several phases. Each phase has a specific role.

---

1. Lexical Analysis (Scanner)

- **Purpose:** Break the input source code into **tokens** (smallest meaningful units).
- **Input:** Source program as a stream of characters.
- **Output:** Tokens (keywords, identifiers, literals, operators, punctuations).
- **Tasks:**
    - Remove whitespaces and comments.
    - Detect invalid characters.
- **Tools/Example:** Lex/Flex.
- **Example:**
- int a = b + 5;

Tokens: int, a, =, b, +, 5, ;

---

2. Syntax Analysis (Parser)

- **Purpose:** Check **syntactic correctness** of the token sequence according to grammar.
- **Input:** Tokens from lexical analysis.
- **Output: Parse tree** or **Concrete Syntax Tree (CST)**.
- **Tasks:**
    - Detect syntax errors.
    - Apply **context-free grammar rules**.
- **Example:** For a + b * c, builds a tree:
- E
- / \
- E *

- / \ / \
- a + b c

---

3. Semantic Analysis

- **Purpose:** Check **meaning of statements**.
- **Input:** Parse tree from syntax analysis.
- **Output: Annotated syntax tree** or **symbol table updates**.
- **Tasks:**
  - Type checking (int vs float).
  - Scope checking (undeclared variables).
  - Function parameter checking.
- **Example:** a = b + "hello" → type error.

---

4. Intermediate Code Generation

- **Purpose:** Generate **intermediate representation (IR)** that is independent of the machine.
- **Examples:**
  - **Three-Address Code (TAC)**
  - **Quadruples / Triples**
- **Advantages:** Makes optimization easier and portable.
- **Example:** x = a + b * c →
- t1 = b * c
- t2 = a + t1
- x = t2

---

5. Code Optimization

- **Purpose:** Improve IR for efficiency without changing meaning.

- **Tasks:**
    - **Peephole optimization**: optimize small instruction sequences.
    - **Common subexpression elimination**: reuse repeated calculations.
    - **Constant folding**: precompute constants.
    - **Dead code elimination**: remove unused code.
- **Example:**
- t1 = a * b
- t2 = a * b

Optimized → reuse t1 instead of computing twice.

---

6. Code Generation

- **Purpose:** Translate optimized IR to **target machine code**.
- **Tasks:**
    - Allocate registers.
    - Generate assembly instructions.
    - Map temporaries and variables to memory/registers.
- **Example:**
- t1 = b * c → MUL R1, b, c
- t2 = a + t1 → ADD R2, a, R1
- x = t2 → MOV x, R2

---

7. Symbol Table Management

- Maintained **throughout compilation**.
- Stores:
    - Identifier names, types, scope
    - Memory locations
    - Function info, parameters

---

8. Error Handling

- Done in **all phases**.
- Examples:
  - Lexical: illegal characters.
  - Syntax: missing semicolon.
  - Semantic: type mismatch.
- Compiler may **recover and continue** or halt with errors.

Summary Table

Phase	Input	Output	Task
Lexical Analysis	Source code	Tokens	Break into meaningful units
Syntax Analysis	Tokens	Parse tree	Grammar checking
Semantic Analysis	Parse tree	Annotated tree	Type & scope checking
Intermediate Code Gen	Annotated tree	IR (TAC, quads)	Machine-independent code
Code Optimization	IR	Optimized IR	Efficiency improvement
Code Generation	Optimized IR	Machine code	Target-specific translation
Symbol Table	Throughout	Symbol table	Manage identifiers & types
Error Handling	Throughout	Error messages	Detect & report errors

---

✅ **In short:**

- **Front-end:** Lexical, Syntax, Semantic → checks correctness.
- **Middle-end:** Intermediate code + Optimization → improves efficiency.
- **Back-end:**

Evaluation of Compilers

A compiler can be evaluated based on several **criteria**, generally divided into **efficiency**, **quality**, and **reliability**.

---

1. Correctness

- A compiler is correct if it **generates code that behaves exactly like the source program**.
- **Test:** Run programs on both source language specification and target machine → results should match.

---

2. Efficiency

a) Compiler Efficiency

- Measures **how fast the compiler runs**.
- A fast compiler improves developer productivity.

b) Generated Code Efficiency

- Measures **quality of machine code produced**:
  - Speed of execution
  - Memory usage
- **Example:** Optimized loops, minimized instructions, reduced memory access.

---

3. Code Optimization

- Compiler should **reduce execution time and memory usage** without changing program semantics.
- Techniques:
  - Common sub-expression elimination
  - Loop unrolling
  - Constant folding

---

4. Compiler Reliability

- Compiler should **handle errors gracefully** and **report meaningful messages**.
- Errors can occur during:
    - Lexical analysis (illegal characters)
    - Syntax analysis (missing tokens)
    - Semantic analysis (type mismatches)

---

5. Portability

- Ability to **run the compiler on different machines**.
- Useful if the compiler itself needs to target **different architectures**.

---

6. Error Detection and Recovery

- A good compiler **detects errors accurately** and **recovers to continue parsing**, instead of stopping at the first error.

---

7. Maintenance and Extensibility

- Easy to **modify** and **extend** for new language features.
- Modular design of compiler phases helps maintainability.

Applications of a Compiler

A **compiler** is not just for translating programs—it has several **practical applications** in computer science and software development.

---

1. Translation of High-Level Programs

- **Primary use** of a compiler is to **translate source code** in high-level languages (like C, Java, Python) into **machine code or intermediate code**.
- Example: C code → Assembly/Machine code.

---

2. Cross-Platform Development

- Compilers allow **source code written in one platform** to be **compiled on another platform**.
- Example: A Java compiler generates **bytecode**, which can run on any machine with a **JVM**.

---

3. Optimization of Code

- Compilers can **improve the efficiency** of programs automatically through:
  - Loop unrolling
  - Dead code elimination
  - Constant folding
- This helps reduce execution time and memory usage.

---

4. Debugging Support

- Compilers provide **error detection and reporting**, which helps in debugging programs:
  - Syntax errors
  - Type mismatches
  - Undefined variables or functions
- Many compilers also provide **line numbers and suggestions** for faster debugging.

---

5. Development of IDEs

- Compilers are a **core component of Integrated Development Environments (IDEs)**.
- IDEs use compilers for:
  - Syntax highlighting
  - Code completion
  - Refactoring tools
  - Real-time error checking

---

6. Intermediate Code Generation

- Compilers generate **intermediate code**, which is platform-independent.
- This intermediate code can be:
  - Interpreted (like Python bytecode)
  - Further compiled for multiple target machines
- Example: C → LLVM IR → machine code for multiple architectures.

---

7. Teaching and Research

- Compilers are widely used in **computer science education**:
  - Understanding programming language design
  - Studying parsing techniques (LL, LR parsers)
  - Experimenting with optimizations and code generation

1. Static Scope (Lexical Scope)

Definition

- The **scope of a variable is determined at compile time** based on the **program text**.
- The compiler decides which variable a name refers to **by looking at the blocks in which it is defined**.
- **Most modern languages (C, Java, Python)** use static scoping.

Characteristics

- Scope is fixed **lexically**, i.e., by the **location of the variable declaration** in the source code.
- Functions/blocks **see variables in outer scopes** if not redefined locally.

Example

```
int x = 10;  // Global variable


void func() {
```

```
    int x = 5;  // Local variable
    printf("%d", x); // prints 5 (local x)
}
```

- The compiler knows **at compile time** which x is being referred to.

Advantages

- Easier to understand and debug.
- Compiler can generate efficient code because the variable's location is known.

---

2. Block Scope

Definition

- A variable declared inside a **block** { … } is **visible only inside that block**.
- Also called **local scope** or **inner scope**.

Characteristics

- The variable **ceases to exist** once the block ends.
- Nested blocks can have variables with **same names** as outer blocks (inner variable **shadows** outer variable).

Example

```
int main() {
    int x = 10;       // outer block
    {
        int x = 5;    // inner block
        printf("%d", x);  // prints 5 (inner x)
    }
    printf("%d", x);      // prints 10 (outer x)
}
```

- The inner x exists **only within the inner block**.
- This is **block scope**.

Comparison: Static Scope vs Block Scope

Feature	Static Scope (Lexical Scope)	Block Scope
Determined at	Compile time	Compile time
Visibility	Depends on program text, can see outer variables	Only inside the block
Lifetime	Global/static depends on declaration	Exists only during block execution
Shadowing	Outer variable can be shadowed by inner declaration	Allowed in nested blocks
Example	x inside function refers to nearest declaration in text	x declared in { } is visible only inside { }

Parameter Passing Mechanisms

When a function is called, **arguments** are passed to **parameters**. The **mechanism of transferring values** from the caller to the callee determines the behavior.

---

1. Pass by Value

- **Copy of the actual parameter** is passed to the function.
- Changes inside the function **do not affect** the original variable.
- **Used in:** C, Java (primitive types).

Example:

```
void func(int x) {
   x = x + 5;
}
```

```
int main() {
    int a = 10;
    func(a);
    printf("%d", a);  // prints 10, original value unchanged
}
```

---

2. Pass by Reference

- **Address of the actual parameter** is passed.
- Changes inside the function **affect the original variable**.
- **Used in:** C++ (with references), languages like Fortran, Pascal.

Example:

```
void func(int &x) {
    x = x + 5;
}
int main() {
    int a = 10;
    func(a);
    printf("%d", a);  // prints 15, original variable changed
}
```

---

3. Pass by Value-Result (Copy-In Copy-Out)

- A **copy** of the argument is passed to the function.
- At the end, the **copy is written back** to the original variable.
- Acts like a **combination of pass by value and pass by reference**.
- **Used in:** Ada, some Fortran compilers.

Behavior:

- Changes inside function **appear in the caller** after function ends.

- Conflicts can occur if **same variable is passed multiple times**.

Input Buffering in Lexical Analysis

Purpose

- Lexical analyzers (scanners) read the **source program character by character**.
- **Input buffering** is used to **efficiently read characters** from the source file, minimizing **disk I/O**.

Why Input Buffering?

- Reading **one character at a time** from disk is **slow**.
- Using a **buffer** reduces the number of I/O operations.
- Helps **handle lookahead** characters efficiently (needed in some tokens, e.g., <= vs <).

Common Strategies

1. Single Buffer

- Use **one array** as a buffer (size N) to store characters.
- **Two pointers**:
    - lexemeBegin → start of current lexeme
    - forward → current scanning position
- When forward reaches the end of buffer → **read next block** of characters from file.

Drawbacks:

- Hard to handle lexemes **split across buffer boundaries**.

---

2. Double Buffering (Two-Buffer Scheme)

- Two buffers of size N each: **Buffer 1** and **Buffer 2**.
- Fill one buffer while scanning the other.
- **Pointers**:

    - lexemeBegin → start of current token
    - forward → current scanning position
- **EOF handling:**
    - Special **sentinel character** at end of each buffer to mark boundary.
- Efficient and commonly used.

Operation:

[Buffer1][Buffer2]

lexemeBegin → start of token

forward → moves through buffer

if forward reaches sentinel → load next buffer

Advantages:

- Handles **long tokens** across buffer boundaries.
- Minimizes I/O operations.

---

3. Sentinel Technique

- Add a **special character (EOF or $)** at the **end of buffer**.
- Helps **detect end-of-buffer** without checking every time.

Example

- Source: int x = 10;
- Lexical analyzer reads in **chunks**:
- Buffer1: int x =
- Buffer2: 10;
- Pointers move through buffers to identify tokens: int, x, =, 10, ;

Summary of Input Buffering Strategies

Strategy	Description	Pros	Cons
Single Buffer	One buffer for input	Simple	Hard to handle tokens across boundaries
Double Buffer	Two buffers alternately filled	Efficient, handles long tokens	Slightly complex
Sentinel Technique	Add special character at buffer end	Easy EOF detection	Needs careful placement

1. Prefix

- A **prefix** of a string S is a sequence of characters that occurs at the **start of the string**.
- Includes **empty string** (ε) and can include the full string itself.

Example:
String: S = "abc"

- Prefixes: ε, "a", "ab", "abc"

---

2. Suffix

- A **suffix** of a string S is a sequence of characters that occurs at the **end of the string**.
- Includes **empty string** and full string.

Example:
String: S = "abc"

- Suffixes: ε, "c", "bc", "abc"

---

3. Substring

- A **substring** is any **contiguous sequence of characters** within the string.
- Can start and end **anywhere** in the string.

Example:
String: S = "abc"

- Substrings: ε, "a", "b", "c", "ab", "bc", "abc"

---

4. Proper Prefix / Proper Suffix / Proper Substring

- **Proper Prefix:** Any prefix **except the string itself**
  - "a", "ab" are proper prefixes of "abc"
- **Proper Suffix:** Any suffix **except the string itself**
  - "c", "bc" are proper suffixes of "abc"
- **Proper Substring:** Any substring **except the string itself**
  - "a", "b", "ab", "bc" are proper substrings of "abc"

Summary Table

Term	Definition	Example for "abc"
Prefix	Starts at beginning	ε, "a", "ab", "abc"
Proper Prefix	Prefix excluding full string	"a", "ab"
Suffix	Ends at end	ε, "c", "bc", "abc"
Proper Suffix	Suffix excluding full string	"c", "bc"
Substring	Any contiguous sequence	ε, "a", "b", "c", "ab", "bc", "abc"
Proper Substring	Substring excluding full string	"a", "b", "c", "ab", "bc"

Algebraic Laws of Regular Expressions

Let ∅ = empty set, ε = empty string, and + = union (OR), · = concatenation, * = Kleene star.

---

1. Identity Laws

- $R + \emptyset = R$
- $R \cdot \varepsilon = R$
- $\varepsilon \cdot R = R$

---

2. Null Laws

- $R + \emptyset = R$ (already covered in identity)
- $R \cdot \emptyset = \emptyset$
- $\emptyset \cdot R = \emptyset$

---

3. Idempotent Law

- $R + R = R$

---

4. Commutative Law (for union)

- $R + S = S + R$

---

5. Associative Laws

- **Union:** $(R + S) + T = R + (S + T)$
- **Concatenation:** $(R \cdot S) \cdot T = R \cdot (S \cdot T)$

---

6. Distributive Laws

- $R \cdot (S + T) = (R \cdot S) + (R \cdot T)$
- $(R + S) \cdot T = (R \cdot T) + (S \cdot T)$

---

7. Closure Laws (Kleene Star)

- $\varepsilon^* = \varepsilon$
- $\emptyset^* = \varepsilon$
- $(R^*)^* = R^*$
- $R^* = \varepsilon + R \cdot R^*$
- $R^* = \varepsilon + R^* \cdot R$

---

8. Other Useful Laws

- $R \cdot \varepsilon = \varepsilon \cdot R = R$
- $R + R \cdot S = R \cdot S^*$ (sometimes useful in simplification)
- $(R + S)^* = (R^* \cdot S^*)^*$ (under certain conditions)

Summary Table

Law Type	Expression Example
Identity	$R + \emptyset = R$, $R \cdot \varepsilon = R$
Null	$R \cdot \emptyset = \emptyset$, $\emptyset \cdot R = \emptyset$
Idempotent	$R + R = R$
Commutative	$R + S = S + R$
Associative	$(R + S) + T = R + (S + T)$
Distributive	$R \cdot (S + T) = R \cdot S + R \cdot T$
Closure	$R^* = \varepsilon + R \cdot R^*$, $(R^*)^* = R^*$

1. Identifier

Rules

- Begins with a **letter** (a-z or A-Z).
- Followed by **letters or digits** (0-9).
- Examples: x, var1, temp123

Regular Expression (RE)

Let:

- letter = (a|b|...|z|A|B|...|Z)
- digit = (0|1|2|...|9)

Then identifier:

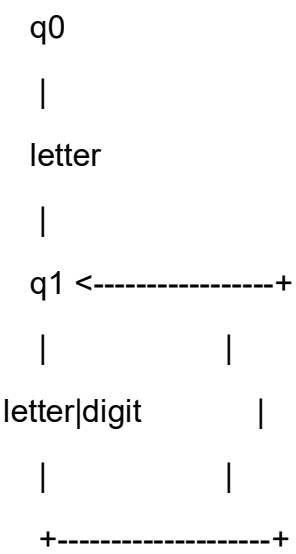
identifier = letter (letter | digit)*

- Explanation:
    - First character = letter
    - Remaining characters = zero or more letters/digits

Transition Diagram (Finite Automaton)

States:

- q0 = start
- q1 = accepting state (identifier recognized)



Explanation:

- From q0, if input is a letter, go to q1.
- From q1, on letter or digit, stay in q1 (loop).
- Accept when end of input is reached in q1.

---

2. Unsigned Number

Rules

- Sequence of digits (0-9)
- Can include **optional decimal point** for real numbers (unsigned float)

Examples:

- Integers: 0, 123, 4567
- Decimals: 0.5, 12.34, 123.0

Regular Expression (RE)

unsigned_integer = digit+

unsigned_real = digit+ . digit+

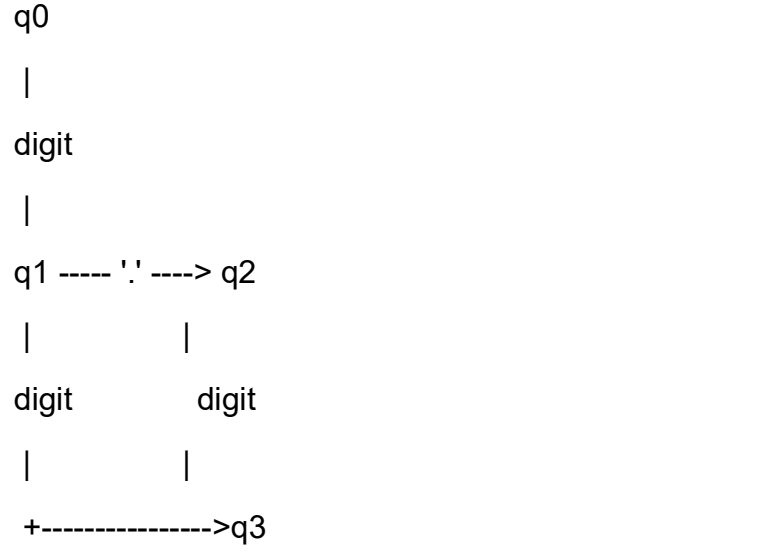
unsigned_number = digit+ (. digit+)?

- Explanation:
    - digit+ = one or more digits
    - Optional decimal part (. digit+)?

Transition Diagram (Finite Automaton)

States:

- q0 = start
- q1 = integer part recognized (accepting state)
- q2 = decimal point read
- q3 = fractional part recognized (accepting state)



Explanation:

- From q0, first digit → q1 (integer part).
- In q1, more digits → stay in q1.
- If . occurs → go to q2.
- From q2, digits → q3.
- Accept in q1 (integer) or q3 (decimal).

✅ **Summary Table**

Token	RE	Accepting State(s)
Identifier	`letter (letter | digit)*`	
Unsigned Number	digit+ (. digit+)?	q1 (int), q3 (real)

Consider the classic **expression grammar**:

E → T E'

E' → + T E' | ε

T → F T'

T' → * F T' | ε

F → (E) | id

Step 1: Construct FIRST sets

Rules for FIRST(X):

1. If X is a terminal → FIRST(X) = {X}
2. If X → ε → add ε
3. If X → Y1 Y2 … → add FIRST(Y1), if ε ∈ FIRST(Y1), also add FIRST(Y2), and so on

---

Compute FIRST sets

1. FIRST(F) = { (, id }
    - F → (E) → (
    - F → id → id
2. FIRST(T') = { *, ε }
    - T' → * F T' → *
    - T' → ε → ε
3. FIRST(T) = { (, id }
    - T → F T' → FIRST(F) = { (, id }
4. FIRST(E') = { +, ε }
    - E' → + T E' → +
    - E' → ε → ε
5. FIRST(E) = { (, id }
    - E → T E' → FIRST(T) = { (, id }

✅ **Summary of FIRST sets**

Non-terminal	FIRST
E	{ (, id }
E'	{ +, ε }
T	{ (, id }
T'	{ *, ε }
F	{ (, id }

Step 2: Construct FOLLOW sets

Rules for FOLLOW(A):

1. Start symbol → $ (end of input)
2. If A → α B β → everything in FIRST(β) except ε is in FOLLOW(B)

3. If ε ∈ FIRST(β) or B at end → add FOLLOW(A) to FOLLOW(B)

Compute FOLLOW sets

1. FOLLOW(E) = { $,) }
    - E is start symbol → $
    - Appears in F → (E) →)
2. FOLLOW(E') = { $,) }
    - E → T E' → FOLLOW(E) ⊆ FOLLOW(E')
3. FOLLOW(T) = { +, $,) }
    - E → T E' → FIRST(E') except ε → +
    - Also, ε ∈ FIRST(E') → FOLLOW(E) → $,)
4. FOLLOW(T') = { +, $,) }
    - T → F T' → FOLLOW(T) → +, $,)
5. FOLLOW(F) = { *, +, $,) }
    - T → F T' → FIRST(T') except ε → *
    - ε ∈ FIRST(T') → FOLLOW(T) → +, $,)

✅ **Summary of FOLLOW sets**

Non-terminal	FOLLOW
E	{ $,) }
E'	{ $,) }
T	{ +, $,) }
T'	{ +, $,) }
F	{ *, +, $,) }

Step 3: Construct Predictive Parsing Table

- Rows = Non-terminals

- Columns = Terminals (id, +, *, (,), $)
- Fill entries using **FIRST and FOLLOW** rules

Parsing Table

NT	id	+	*	(	)	$
E	E → T E'			E → T E'		
E'		E' → + T E'			E' → ε	E' → ε
T	T → F T'			T → F T'		
T'		T' → ε	T' → * F T'		T' → ε	T' → ε
F	F → id			F → (E)		

✅ Empty cells mean **error entries**.

Step 4: Predictive Parsing of id + id * id

Stack: E $
Input: id + id * id $

Steps:

Stack	Input	Action
E $	id + id * id $	E → T E'
T E' $	id + id * id $	T → F T'
F T' E' $	id + id * id $	F → id
id T' E' $	id + id * id $	Match id
T' E' $	+ id * id $	T' → ε
E' $	+ id * id $	E' → + T E'
+ T E' $	+ id * id $	Match +
T E' $	id * id $	T → F T'

Stack	Input	Action
F T' E' $	id * id $	F → id
id T' E' $	id * id $	Match id
T' E' $	* id $	T' → * F T'
* F T' E' $	* id $	Match *
F T' E' $	id $	F → id
id T' E' $	id $	Match id
T' E' $	$	T' → ε
E' $	$	E' → ε
$	$	Accept

Define the Grammar

Let's use a simple grammar suitable for arithmetic expressions with only a and +, * operators:

E → E + E

E → E * E

E → a

- Terminals: a, +, *
- Non-terminal: E

Step 1: Input String

Input: a a a * a a + + $

- We'll parse aaa*aa++ assuming **a as operand** and $ as end marker.

Step 2: Initialize Stack

- Stack starts empty.

- Input: aaa*aa++$

Step 3: Shift-Reduce Parsing Table (conceptual)

- **Shift:** Push next input symbol onto stack.
- **Reduce:** Apply grammar **rightmost derivation in reverse**:
    1. a → E
    2. E * E → E
    3. E + E → E

Step 4: Parsing Steps

Step	Stack	Input	Action
1		aaa*aa++$	Initial
2	a	aa*aa++$	Shift a
3	E	aa*aa++$	Reduce a → E
4	E a	a*aa++$	Shift a
5	E E	*aa++$	Reduce a → E
6	E E *	aa++$	Shift *
7	E * E	a++$	Shift a
8	E * E E	++$	Reduce a → E
9	E E	++$	Reduce E * E → E
10	E E +	+$	Shift +
11	E + E	$	Shift E
12	E	$	Reduce E + E → E
13	E	$	Accept

Step 5: Notes

1. **Shift:** Move next input to stack.
2. **Reduce:** Replace RHS of a production with LHS E.
3. **Accept:** Input consumed and stack has single E.

---

1. Shift-Reduce Parsing Overview

- **Shift-Reduce Parsing** is a **bottom-up parsing** technique.
- It tries to **reduce a string to the start symbol** by **reversing rightmost derivation**.
- **Actions:**
    1. **Shift:** Push the next input symbol onto the stack.
    2. **Reduce:** Replace a **handle** (RHS of a production) on the stack with its LHS.
    3. **Accept:** When stack contains start symbol and input is empty.
    4. **Error:** If no action possible.

Sample Grammar

We will use this classic arithmetic grammar:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow id$

- Terminals: id, +, *
- Non-terminal: E

---

2. Types of Shift-Reduce Parsers

There are **three main types** of shift-reduce parsers:

(A) Operator-Precedence Parser

Key Idea

- Uses **precedence relations** between operators to decide shift or reduce.
- Relations: <. (less), =. (equal), `>.' (greater)
- Works only for **operator-precedence grammars** (no ambiguity, no ε-productions, no two adjacent non-terminals).

Parsing Steps (Example: id + id * id)

1. Assign **precedence**: * > +
2. Start with **stack $** and input: id + id * id $
3. Shift id → reduce immediately to E.
4. Compare **stack top operator and next input operator**:
    - Stack top < input → **Shift**
    - Stack top > input → **Reduce**
5. Continue until input exhausted and stack = E.

✅ Produces correct parse using **operator precedence rules**.

(B) SLR(1) (Simple LR) Parser

Key Idea

- Uses **LR(0) items + FOLLOW sets** to construct **parsing table**.
- Table contains **shift, reduce, accept, and error** actions.
- Handles a **larger class of grammars** than operator-precedence.

Parsing Steps

1. Build **canonical collection of LR(0) items**.
2. Compute **ACTION** and **GOTO** table.
3. Stack stores **state numbers**, not just symbols.
4. Input: id + id * id $
5. At each step:
    - Check ACTION[state, input] → **shift** or **reduce**

- GOTO updates state after reduction.

6. Accept when stack = start symbol and input = $.

✅ More general and robust than operator-precedence parsing.

(C) LALR(1) Parser (Lookahead LR)

Key Idea

- **Lookahead version** of LR parser with **fewer states**.
- Combines states in SLR to reduce table size while keeping **1-symbol lookahead**.
- Commonly used in tools like **YACC, Bison**.

Steps

1. Construct **LR(1) items with 1 lookahead symbol**.
2. Merge compatible states to reduce size → **LALR(1) table**.
3. Parse input using **shift, reduce, accept** rules similar to SLR.

✅ More **space-efficient** than full LR(1) parser.

---

3. Comparison Table

Parser Type	Lookahead	Table Size	Grammar Supported	Notes
Operator-Precedence	1	Small	Operator-precedence only	Simple, fast, limited
SLR(1)	1	Medium	Simple LR(1) grammars	General, robust
LALR(1)	1	Small	Same as SLR(1)	Efficient, widely used
LR(1)	1	Large	All LR(1) grammars	Most powerful, but big table

Step 1: Choose Grammar

We will use a simple arithmetic expression grammar suitable for **recursive descent parsing**:

E → T E'

E' → + T E' | ε

T → F T'

T' → * F T' | ε

F → (E) | id

- Terminals: id, +, *, (,)
- Non-terminals: E, E', T, T', F

Note: Grammar must be **LL(1)** (no left recursion) to use recursive descent.

Step 2: Construct FIRST and FOLLOW sets (already done in previous discussion)

- FIRST(E) = { id, (}
- FIRST(E') = { +, ε }
- FIRST(T) = { id, (}
- FIRST(T') = { *, ε }
- FIRST(F) = { id, (}
- FOLLOW(E) = { $,) }
- FOLLOW(E') = { $,) }
- FOLLOW(T) = { +, $,) }
- FOLLOW(T') = { +, $,) }
- FOLLOW(F) = { *, +, $,) }

Step 3: Write Recursive Descent Procedures

Each **non-terminal** corresponds to a **function**:

// Assume 'token' is the current input token

```
void E() {
    T();
    E_prime();
}

void E_prime() {
    if (token == '+') {
        match('+');
        T();
        E_prime();
    } else if (token == '$' || token == ')') {
        // epsilon, do nothing
    } else {
        error();
    }
}

void T() {
    F();
    T_prime();
}

void T_prime() {
    if (token == '*') {
        match('*');
        F();
        T_prime();
```

```
    } else if (token == '+' || token == '$' || token == ')') {
        // epsilon, do nothing
    } else {
        error();
    }
}

void F() {
    if (token == 'id') {
        match('id');
    } else if (token == '(') {
        match('(');
        E();
        match(')');
    } else {
        error();
    }
}

// Match function moves to next token if current matches
void match(string expected) {
    if (token == expected) {
        token = nextToken();
    } else {
        error();
    }
}
```

Step 4: Parsing Input Example

Input: id + id * id $

Parsing sequence:

1. E() calls T() → F() → matches id
2. T'() → epsilon (next token +)
3. E'() → matches +, calls T()
4. T() → F() → matches id
5. T'() → matches *, calls F() → matches id
6. T'() → epsilon, return
7. E'() → epsilon, return
8. E() → done, stack empty → **accept**

Step 5: Notes

- Each non-terminal → **function**
- Each terminal → **match() function**
- **Epsilon** → just return
- Works **top-down**, parsing **LL(1) grammar**

✅ **Summary Table**

Non-terminal	Function
E	E() calls T(); E_prime();
E'	E_prime() handles `+ T E'
T	T() calls F(); T_prime();
T'	T_prime() handles `* F T'
F	F() handles `(E)

Step 1: Choose Grammar

Consider the classic grammar for arithmetic expressions:

1. E → E + T

2. E → T

3. T → T * F

4. T → F

5. F → (E)

6. F → id

Augmented Grammar

- Add a new start symbol E':

0. E' → E

- Terminals: id, +, *, (,)
- Non-terminals: E, T, F

Step 2: LR(1) Item Definition

An **LR(1) item** is:

[A → α • β, a]

- • indicates the **position of parser** in the production
- a is the **lookahead symbol** (terminal or $)
- Represents: “we have seen α, expect β, and next input should be a”

Step 3: Closure Operation

Closure(I):

1. Start with a set of items I.

2. For each item [A → α • B β, a] where B is non-terminal,
   add [B → • γ, b] for each production B → γ and **for each b ∈ FIRST(βa)**.
3. Repeat until no more items can be added.

Step 4: GOTO Operation

GOTO(I, X):

- Move • over symbol X in items of I
- Take **closure** of the resulting set

GOTO(I, X) = CLOSURE({ [A → α X • β, a] | [A → α • X β, a] ∈ I })

Step 5: Construct Canonical Collection of LR(1) Items

Step 5.1: Start with augmented start item

I0 = CLOSURE({ [E' → • E, $] })

- [E' → • E, $]
- E → E + T → add [E → • E + T, $]
- E → T → add [E → • T, $]
- T → T * F → add [T → • T * F, +] (because lookahead is FIRST of rest of production +)
- T → F → add [T → • F, +]
- F → (E) → add [F → • (E), +]
- F → id → add [F → • id, +]

I0 now contains **all items with • at beginning and appropriate lookahead**

Step 5.2: Compute GOTO sets from I0

- For each symbol X after • in I0, compute:

1. GOTO(I0, E) → I1
2. GOTO(I0, T) → I2
3. GOTO(I0, F) → I3

4. GOTO(I0, id) → I4
5. GOTO(I0, '(') → I5

- Repeat **closure** for each new set
- Continue computing **GOTO for all new sets** until no new sets appear

---

Step 5.3: Continue until Canonical Collection Complete

- After iterating, you will get a collection of sets:

I0, I1, I2, I3, I4, I5, ...

- Each set = **state of the LR(1) parser**

Step 6: Build LR(1) Parsing Table

- **Rows:** States (I0, I1, ...)
- **Columns:** Terminals + Non-terminals
- **Actions:** Shift, Reduce, Accept
- LR(1) parser uses **lookahead symbol** to decide which reduction to apply → avoids conflicts of SLR(1)

Step 7: Notes

1. LR(1) items = **LR(0) item + 1 lookahead symbol**
2. Closure operation = key step
3. GOTO determines **state transitions**
4. Canonical collection can be **large**; LALR(1) reduces table size

- This is a **synthesized attribute SDD** (bottom-up computation).
- Semantic rules **propagate values upward** in the parse tree.
- Can be implemented in a **recursive-descent parser** by evaluating val during parsing.

8.a) Consider the classic arithmetic grammar:

1. E → E + T

2. E → T

3. T → T * F

4. T → F

5. F → (E)

6. F → id

Augmented Grammar

- Add new start symbol E':

0. E' → E

- Terminals: id, +, *, (,)
- Non-terminals: E, T, F

Step 2: LR(0) Item Definition

An **LR(0) item** is:

[A → α • β]

- • indicates the **position in the production**
- Unlike LR(1), there is **no lookahead symbol** in LR(0)

Step 3: Closure Operation

Closure(I):

1. Start with a set of items I.
2. For each item [A → α • B β] where B is non-terminal,
   add [B → • γ] for **every production B → γ**
3. Repeat until no new items can be added

Step 4: GOTO Operation

GOTO(I, X):

GOTO(I, X) = CLOSURE({ [A → α X • β] | [A → α • X β] ∈ I })

- Move • over symbol X in items of I
- Take **closure** of resulting set

Step 5: Construct Canonical Collection

Step 5.1: Start with augmented start item

I0 = CLOSURE({ [E' → • E] })

- [E' → • E]
- Add items for E:

[E → • E + T]

[E → • T]

- Add items for T:

[T → • T * F]

[T → • F]

- Add items for F:

[F → • (E)]

[F → • id]

✅ **I0 now contains all items with • at beginning**

Step 5.2: Compute GOTO sets from I0

- For each symbol X after • in I0, compute GOTO:

1. GOTO(I0, E) → I1
2. GOTO(I0, T) → I2
3. GOTO(I0, F) → I3
4. GOTO(I0, id) → I4

5. GOTO(I0, '(') → I5

- Apply **closure** for each new set

Step 5.3: Continue GOTO for all new sets

- Compute GOTO for every symbol after • in new sets
- Continue until **no new sets** are produced
- Resulting sets form the **canonical collection of LR(0) items**:

I0, I1, I2, I3, I4, I5, ...

Each set corresponds to a **state of the LR(0) parser**

Step 6: Build SLR(1) Table (Optional Next Step)

- **Rows:** States (I0, I1, ...)
- **Columns:** Terminals + Non-terminals
- **Actions:** Shift, Reduce, Accept

Step 7: Notes

1. **LR(0) item** = LR(1) item without lookahead
2. Closure operation is the **key**
3. GOTO determines **state transitions**
4. Canonical collection forms the **state machine** for LR(0) parsing

---

8b. 1. Synthesized Attributes

Definition

- A **synthesized attribute** of a **non-terminal** is computed **from its children (or subtrees)** in the parse tree.
- Information **flows upward** from the leaves toward the root.

Characteristics

- Often used to **compute values of expressions**, types, or any property derived from subcomponents.
- Easy to implement in **bottom-up parsing**.

Example

Grammar:

E → E1 + T

E → T

T → T1 * F

T → F

F → num

- Attribute: val (value of expression)
- Semantic rules:

E.val = E1.val + T.val

T.val = T1.val * F.val

F.val = num.lexval

Explanation:

- E.val depends on **child nodes** E1 and T.
- So val is a **synthesized attribute**.

Flow: Leaf → Root

---

2. Inherited Attributes

Definition

- An **inherited attribute** of a **non-terminal** is computed **from its parent or siblings** in the parse tree.
- Information **flows downward or sideways** from parent/sibling toward the node.

Characteristics

- Often used to **pass types, symbol table info, or context**.
- Implemented in **top-down or recursive descent parsing**.

Example

Grammar:

S → L = R

L → id

R → expr

- Suppose L.type depends on **parent or right-hand side**:

L.inh = S.type

- Another example: passing inherited attributes for **operator precedence or offsets** in arrays.

Flow: Parent/Sibling → Child

---

3. Summary Table

Attribute Type	Computed From	Flow Direction	Typical Use
Synthesized	Children/Subtrees	Upward (bottom-up)	Expression value, type, code
Inherited	Parent/Siblings	Downward/Sideways	Type info, symbol table, offsets

---

4. Notes

- **SDDs** may use **only synthesized**, **only inherited**, or **both**.
- **L-attributed SDDs:** Only allow inherited attributes **from parent or left siblings** (suitable for top-down parsing).
- **S-attributed SDDs:** Only **synthesized attributes** (suitable for bottom-up parsing).