**Unit 3**

**Syntax Analysis**

**Introduction**

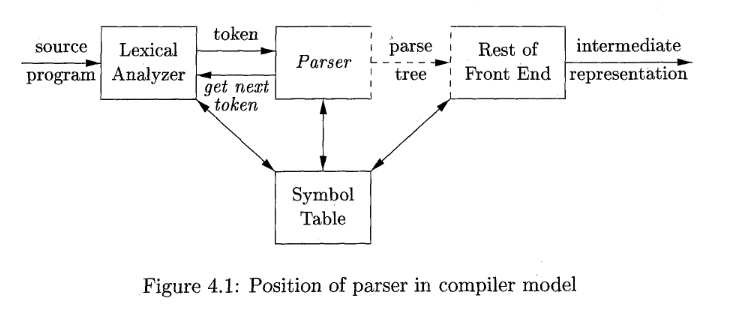
1. **Role of the Parser in a Compiler**:
   * The parser is responsible for analysing the structure of the source code according to the rules of a grammar.
   * It determines if the input program is syntactically correct.
2. **Grammars and Parsing**:
   * **Grammars**: A formal set of rules defining valid constructs in a language.
   * Example: Arithmetic expression grammars are commonly used to explain parsing concepts.
   * Techniques for parsing expressions are applicable to most programming language constructs.
3. **Error Handling in Parsing**:
   * The parser must identify and gracefully handle errors if the input doesn't conform to the grammar.
   * Effective error handling ensures the compiler remains robust and user-friendly.

**The Role of the Parser**

The parser is an important component of the compiler that receives a stream of tokens from the lexical analyzer and verifies if these tokens follow the grammar rules of the source language. If the input is valid, the parser constructs a parse tree that represents the program's structure and passes it to the next stage of the compiler. In practice, the parse tree may not always be explicitly built; instead, parsing actions can include tasks like checking the program or generating intermediate code alongside the parsing process.

The parser also plays a critical role in detecting and handling syntax errors. When an error is found, the parser ensures that the error is reported in a clear and understandable way. It also attempts to recover from common errors so that the rest of the program can still be processed without interruptions.

In the compiler model, the parser works alongside other components of the front end. It updates the **symbol table** with relevant information about tokens, performs **semantic analysis** like type checking, and may even assist in generating intermediate code. These additional tasks, along with parsing, are collectively managed by the rest of the front end, as shown in the figure.



There are three main types of parsers: **universal parsers**, **top-down parsers**, and **bottom-up parsers**. Universal parsers, like the Cocke-Younger-Kasami (CYK) algorithm and Earley's algorithm, can handle any grammar, but they are computationally expensive and are not suitable for real-world compilers. Instead, most compilers rely on either top-down or bottom-up parsing methods.

**Representative Grammars**

**What Are Grammars in Parsing?**

* Grammars are rules that define how programming constructs like expressions or loops are structured.
* Think of a grammar like a recipe: it tells the parser how to "build" valid code structures from tokens (pieces of code, like keywords, operators, etc.).

**Why Are Expressions Challenging to Parse?**

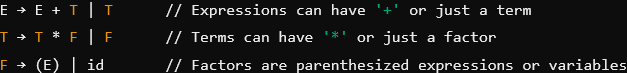
* Constructs like while or int are easy to parse because keywords clearly guide the parser.
* **Expressions**, on the other hand, involve:
  1. **Operator Precedence**: Which operator to evaluate first (e.g., \* before +).
  2. **Associativity**: How to group operators (e.g., left-to-right or right-to-left).

**1. Expression Grammar**

This grammar handles the precedence and associativity of operators (+, \*):

* E represents **expressions** like a + b.
* T represents **terms** (higher precedence, e.g., a \* b).
* F represents **factors**, the most basic building blocks (e.g., a, (b + c)).

**Rules**:



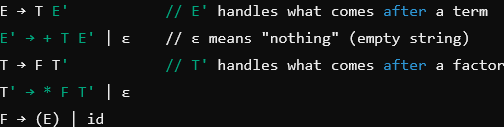
**Key Points**:

* This grammar **preserves precedence and associativity**:
  + \* has higher precedence than +.
  + + and \* are left-associative (a + b + c is grouped as (a + b) + c).
* **Limitation**: It is **left-recursive**, meaning the rule calls itself on the left side (e.g., E → E + T).  
  This makes it unsuitable for **top-down parsing**.

**2. Non-Left-Recursive Grammar**

To make the grammar work for **top-down parsing**, we eliminate left recursion. Here’s how the new rules look:

**Rules**:



**Key Points**:

* The grammar now works for **top-down parsers** like **LL parsers**.
* It still respects precedence (\* > +) and associativity (left-to-right).

**3. Grammar for Ambiguities**

Sometimes, we need a grammar to explore ambiguities in parsing. For example:

* **Ambiguity**: When a parser can produce more than one valid parse tree for the same input.  
  Example: a + b \* c could mean (a + b) \* c or a + (b \* c).

**Modified Grammar**:



* Here, all operators (+, \*) are treated equally (same precedence).
* This grammar allows multiple parse trees, making it useful for studying ambiguity resolution.

**Error Handling and Recovery**

**Why is Syntax Error Handling Important?**

* **Real-World Reality**: Programs are rarely written perfectly on the first attempt. Syntax errors, like missing semicolons or braces, are common.
* A **compiler** not only translates code but also helps the programmer by identifying and reporting errors so they can be fixed efficiently.
* **Challenge for Compiler Designers**: Programming language specifications rarely specify how errors should be handled. This means the **compiler designer** must plan and implement error-handling strategies.

**Types of Errors**

Errors in programming can occur at different levels. Let’s categorize them:

1. **Lexical Errors**:
   * Occur when **tokens** (basic code units like identifiers or keywords) are not valid.
   * Examples:
     + Misspelled keywords: elipsesize instead of ellipsesize.
     + Missing quotes around strings: Hello World instead of "Hello World".
2. **Syntactic Errors**:
   * Caused by violating the **grammar rules** of a language.
   * Examples:
     + Missing or misplaced semicolons, braces, or parentheses (if (x > 0 { instead of if (x > 0) {).
     + A case statement without a switch.
3. **Semantic Errors**:
   * These involve violations of the **meaning** or **type rules** of the language.
   * Examples:
     + Using a return statement in a method with a void return type in Java.
4. **Logical Errors**:
   * The program runs but produces unintended results due to **incorrect reasoning**.
   * Example:
     + Using = instead of == in a condition (if (x = 5) assigns 5 to x instead of comparing it to 5).

**Goals of an Error Handler in a Parser**

The **error handler** in a parser has three primary goals:

1. **Report Errors Clearly and Accurately**:
   * Tell the programmer exactly what went wrong and where.
   * Example: "Missing ) at line 10."
2. **Recover Quickly to Detect More Errors**:
   * After finding an error, the parser should **resume parsing** to find other issues instead of stopping entirely.
3. **Minimal Overhead for Correct Programs**:
   * For programs without errors, the error-handling mechanism should not slow down compilation.

**How Do Parsing Methods Detect Errors?**

Parsing methods (like **LL** and **LR**) are very precise in detecting syntax errors. They have the **viable-prefix property**, which means:

* As soon as the input **cannot be completed** into a valid string based on the grammar, they detect an error.
* This allows errors to be caught **as early as possible**.

**Error-Recovery Strategies**

Error recovery in a compiler helps it handle errors gracefully and allows further processing to detect additional issues. While no universal strategy fits all cases, several methods are commonly used. Below is a breakdown of these strategies:

**1. Panic-Mode Recovery**

* **How It Works**:  
  When an error is detected, the parser discards input tokens one by one until it finds a **synchronizing token**. These tokens are usually delimiters like semicolons (;) or closing braces (}) that clearly mark the end of a code block.
* **Advantages**:
  + Simple to implement.
  + Ensures the parser does not enter an **infinite loop**.
  + Quickly resumes parsing after an error.
* **Disadvantages**:
  + Skips a large portion of the input, possibly missing multiple errors in that section.

**2. Phrase-Level Recovery**

* **How It Works**:  
  The parser makes **local corrections** to the input, such as:
  + Replacing a comma with a semicolon.
  + Inserting a missing token like a closing parenthesis.
  + Deleting an extraneous token like an extra semicolon.
* **Advantages**:
  + Allows the parser to continue parsing without skipping large sections of the input.
  + Can handle any input string.
* **Disadvantages**:
  + Difficult to handle errors that occur **before the point of detection**.
  + Poorly chosen corrections can lead to **infinite loops** (e.g., if the parser keeps inserting tokens indefinitely).

**3. Error-Productions**

* **How It Works**:  
  The grammar is **augmented with error productions** that represent common mistakes.
  + When the parser encounters an error production, it recognizes the erroneous construct and generates a specific error message.
* **Advantages**:
  + Provides **precise diagnostics** for anticipated errors.
* **Disadvantages**:
  + Limited to **only those errors** explicitly accounted for in the grammar.
  + Designing and implementing these error productions is tedious and language-dependent.

**4. Global Correction**

* **How It Works**:  
  The compiler finds a **minimal sequence of changes** (insertions, deletions, or replacements of tokens) to transform the incorrect input string into a valid one.
  + Algorithms are used to determine the closest correct program (string) to the input program.
* **Advantages**:
  + Provides the **most accurate correction**.
  + Serves as a benchmark for evaluating other recovery strategies.
* **Disadvantages**:
  + Computationally expensive in terms of **time and memory**.
  + The "closest" program may not match the programmer's intent.
  + Used primarily for theoretical purposes due to high cost.

import { useEffect, useState } from "react";

import { useNavigate, useParams } from "react-router-dom";

import { useGetProductByIdQuery } from "../utils/userServices";

import { Rating } from "@mui/material";

import LocalMallOutlinedIcon from "@mui/icons-material/LocalMallOutlined";

import Counter from "../components/Counter";

import { isUserAuthenticated } from "../utils/userAuth";

import { useSnackbar } from "notistack";

import { useDispatch, useSelector } from "react-redux";

import { addToCartAsync, updateRetailerCart } from "../utils/cartSlice";

import RetailerCounter from "../components/RetailerCounter";

const ProductView = () => {

const { id } = useParams();

const { data, isLoading, refetch } = useGetProductByIdQuery(id);

const [product, setProduct] = useState(null);

const [quantityAvailable, setQuantityAvailable] = useState(null);

const [retailerCartQuantity, setRetailerCartQuantity] = useState(100);

const [userRating, setUserRating] = useState(0); // User's rating input

const [userComment, setUserComment] = useState(""); // User's comment input

const [isRatingSubmitted, setIsRatingSubmitted] = useState(false); // Track if rating is submitted

const { enqueueSnackbar } = useSnackbar();

const role = isUserAuthenticated();

const navigate = useNavigate();

const dispatch = useDispatch();

const { cartItems } = useSelector((state) => state.cart);

const userId = useSelector((state) => state.auth.user?.\_id); // Get logged-in user ID

// Fetch product data and pre-fill user's existing rating

useEffect(() => {

if (data) {

setProduct(data?.product);

setQuantityAvailable(data?.product?.quantityAvailableInKg);

// Pre-fill user's existing rating and comment

if (userId) {

const userRating = data?.product?.ratings?.find(

(r) => r.userId === userId

);

if (userRating) {

setUserRating(userRating.rating);

setUserComment(userRating.comment || "");

setIsRatingSubmitted(true);

}

}

// Retailer cart

if (role === "retailer" && cartItems) {

setRetailerCartQuantity(cartItems[data?.product.\_id] || 100);

}

}

}, [data, cartItems, role, userId]);

// Handle adding or updating a rating

const handleRatingSubmit = async () => {

if (!role) {

enqueueSnackbar("Please Sign In to submit a rating", {

variant: "warning",

});

return;

}

if (userRating < 1 || userRating > 5) {

enqueueSnackbar("Please select a rating between 1 and 5", {

variant: "error",

});

return;

}

try {

const response = await fetch(

`${import.meta.env.VITE\_API\_SERVER\_URL}/api/products/${id}/ratings`,

{

method: "POST",

headers: {

"Content-Type": "application/json",

Authorization: `Bearer ${localStorage.getItem("token")}`,

},

body: JSON.stringify({

userId,

rating: userRating,

comment: userComment,

}),

}

);

if (!response.ok) {

throw new Error("Failed to submit rating");

}

const result = await response.json();

enqueueSnackbar("Rating submitted successfully!", {

variant: "success",

});

setIsRatingSubmitted(true);

refetch(); // Refresh product data to show updated ratings

} catch (error) {

enqueueSnackbar(error.message || "Failed to submit rating", {

variant: "error",

});

}

};

// Calculate average rating

const averageRating = product?.ratings?.length

? product.ratings.reduce((sum, r) => sum + r.rating, 0) /

product.ratings.length

: 0;

if (!product) {

return (

<div className="flex justify-center items-center h-screen">

<div className="animate-spin rounded-full h-12 w-12 border-t-2 border-b-2 border-green-500"></div>

</div>

);

}

return (

<div className="min-h-screen bg-gray-50 py-10 px-4 sm:px-6 lg:px-8">

<div className="max-w-7xl mx-auto">

{/\* Product Details Section \*/}

<div className="grid grid-cols-1 lg:grid-cols-2 gap-8">

{/\* Product Image \*/}

<div className="w-full lg:w-[80%]">

<img

src={`${import.meta.env.VITE\_API\_SERVER\_URL}/uploads/${

product?.images[0]

}`}

alt={product?.name}

className="w-full h-auto rounded-lg shadow-lg"

/>

</div>

{/\* Product Information \*/}

<div className="flex flex-col space-y-6">

<h1 className="text-4xl font-bold text-gray-900">

{product?.name}

</h1>

{/\* Rating \*/}

<div className="flex items-center space-x-2">

<Rating value={averageRating} readOnly precision={0.5} />

<p className="text-sm text-gray-600">

({product?.ratings?.length || 0} ratings)

</p>

</div>

{/\* Price \*/}

<p className="text-3xl font-semibold text-green-600">

₹{product?.pricePerKg} / kg

</p>

{/\* Description \*/}

<div className="text-gray-700">

<p className="text-lg font-medium">Description</p>

<p className="text-sm">{product?.description}</p>

</div>

{/\* Stock Status \*/}

<div className="text-gray-700">

<p className="text-lg font-medium">Availability</p>

{!quantityAvailable || quantityAvailable < 1 ? (

<p className="text-red-600">Out of Stock</p>

) : (

<p className="text-green-600">

{quantityAvailable} kg(s) available

</p>

)}

</div>

{/\* Add to Cart Section \*/}

<div className="flex items-start space-x-2">

{role === "retailer" ? (

<div className="flex flex-col">

<RetailerCounter

cartQuantity={retailerCartQuantity}

setCartQuantity={setRetailerCartQuantity}

/>

<p className="text-sm mt-2 text-gray-500">

Minimum order quantity:{" "}

<span className="font-medium text-black">100 KG</span>

</p>

</div>

) : (

<Counter

productId={id}

count={cartItems[id]}

cartFarmerId={product.farmer.\_id}

quantityAvailable={quantityAvailable}

setQuantityAvailable={setQuantityAvailable}

/>

)}

{/\* Add to cart button \*/}

<button

onClick={handleCart}

className="flex items-center justify-center px-8 py-3 bg-green-600 text-white rounded-full font-medium hover:bg-green-700 transition-colors"

>

{!cartItems[id]

? "Add to Cart"

: role === "retailer"

? "Update Cart"

: "Go to Cart"}

<LocalMallOutlinedIcon className="ml-2" />

</button>

</div>

{/\* Additional Details \*/}

<div className="text-gray-700">

<p className="text-lg font-medium">Product Details</p>

<ul className="list-disc list-inside text-sm">

<li>Category: {product?.category}</li>

<li>

Harvest Date:{" "}

{new Date(product?.harvestDate).toLocaleDateString()}

</li>

<li>Farm: {product?.farmer?.farmName}</li>

<li>Farmer: {product?.farmer?.name}</li>

</ul>

</div>

</div>

</div>

{/\* Reviews Section \*/}

<div className="mt-12">

<h2 className="text-2xl font-bold text-gray-900 mb-6">

Customer Reviews

</h2>

{/\* Rating Input Section \*/}

<div className="mb-8">

<h3 className="text-lg font-medium mb-4">

{isRatingSubmitted ? "Update Your Rating" : "Add a Rating"}

</h3>

<div className="flex flex-col space-y-4">

<Rating

value={userRating}

onChange={(\_, newValue) => setUserRating(newValue)}

precision={0.5}

/>

<textarea

value={userComment}

onChange={(e) => setUserComment(e.target.value)}

placeholder="Write a review..."

className="w-full p-2 border rounded-lg focus:outline-none focus:ring-2 focus:ring-green-500"

rows={3}

/>

<button

onClick={handleRatingSubmit}

className="px-6 py-2 bg-green-600 text-white rounded-lg hover:bg-green-700 transition-colors"

>

{isRatingSubmitted ? "Update Rating" : "Submit Rating"}

</button>

</div>

</div>

{/\* Display Reviews \*/}

{product?.ratings?.length > 0 ? (

<div className="space-y-4">

{product.ratings.map((rating, index) => (

<div key={index} className="bg-white p-4 rounded-lg shadow-sm">

<div className="flex items-center space-x-2">

<Rating value={rating.rating} readOnly precision={0.5} />

<p className="text-sm text-gray-600">

by {rating.user?.name || "Anonymous"}

</p>

</div>

<p className="text-gray-700 mt-2">{rating.comment}</p>

</div>

))}

</div>

) : (

<p className="text-gray-600">No reviews yet.</p>

)}

</div>

</div>

</div>

);

};

export default ProductView;