BAHIR DAR UNIVERSITY

INSTITUTE OF TECHNOLOGY (BIT)

FACULITY OF COMPUTING

DEPARTMENT OF SOFTWARE ENGINEERING

PRINCIPLES OF COMPILER DESIGN: INDIVIDUAL ASSIGNMENT.

STUDENT NAME: ID

BIRTUKAN NIGUSSIE 1505500

SUBMITTED TO: LEC...

SUBMISION DATE: 27/04/2018 E.C

Table of Contents

INTRODUCTION:

1. **Assignment 01 – Syntax Analysis** Focuses on the foundational concepts of context-free grammars, terminals and non-terminals, parsing techniques, and practical implementation of a stack-based validator for balanced parentheses. It also includes analysis of parse trees for a given grammar, reinforcing the understanding of how compilers verify the structural correctness of source code.
2. **Assignment 02 – Presentation on Parsing Techniques** Explores predictive parsing in depth, specifically the construction and role of the parse table in LL(1) parsers. This topic bridges theoretical syntax analysis with practical table-driven parsing, a technique widely used in compiler front-ends for deterministic and efficient top-down parsing.
3. **Assignment 03 – Semantic Analysis** Delves into the semantic phase, where the compiler performs type checking to ensure meaningful and type-safe operations. The selected task implements a type checker for assignment statements, utilizing a symbol table to track variable types and enforce compatibility rules with support for safe implicit conversions.

Together, these assignments cover critical aspects of the compiler front-end:

- Recognizing tokens and structure (lexical and syntax analysis),
- Building predictive parsing mechanisms,
- Enforcing semantic correctness through type checking.

The solutions include detailed theoretical explanations, complete C++ implementations where required, illustrative diagrams, symbol table usage, and clear examples. All work is grounded in standard compiler design principles as presented in classic references such as the "Dragon Book" and course materials.

This compilation demonstrates a progressive understanding of how compilers transform source code into semantically valid and syntactically correct representations, laying the foundation for subsequent optimization and code generation phases.

Assignment 01: Question Number 06 (STUDENT 06)

Syntax Analysis

Terminals and Non-Terminals in a Grammar

What are terminals and non-terminals in a grammar?

In the context of **context-free grammars (CFGs)**, which form the foundation of syntax analysis in compiler design, symbols are classified into two categories: **terminals** and **non-terminals**.

- **Terminals** are the literal symbols or tokens that appear in the final strings of the language. They cannot be further expanded by any production rule. Terminals typically represent keywords, operators, identifiers, or punctuation scanned by the lexical analyzer (lexer). They are usually denoted in lowercase letters or as explicit symbols.
- **Non-terminals** are abstract symbols representing syntactic constructs (e.g., expressions, statements). They can be replaced by production rules during derivation and do not appear in the final output string. Non-terminals are conventionally written in uppercase, with one designated as the start symbol (often S).

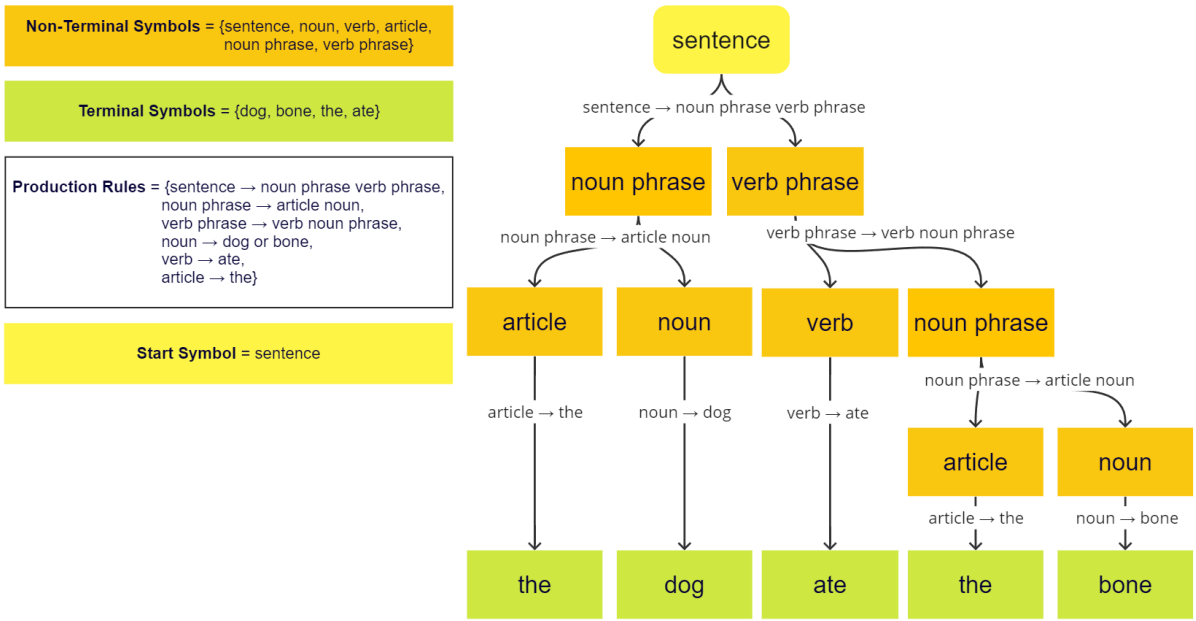


Illustration of Terminals and Non-terminals (from Wikipedia example showing derivation steps highlighting terminals in green and non-terminals in blue).

Example Grammar (simple arithmetic expressions): Productions: E → E + T | T T → T * F | F F → (E) | id

- **Non-terminals:** E (expression), T (term), F (factor)

- **Terminals**: +, *, (,), id

In the string "id + id * id", the terminals are the actual tokens matched from input, while non-terminals guide the hierarchical structure built during parsing.

This separation is critical in compilers: the syntax analyzer (parser) uses non-terminals to validate structure, while terminals come directly from the lexer.

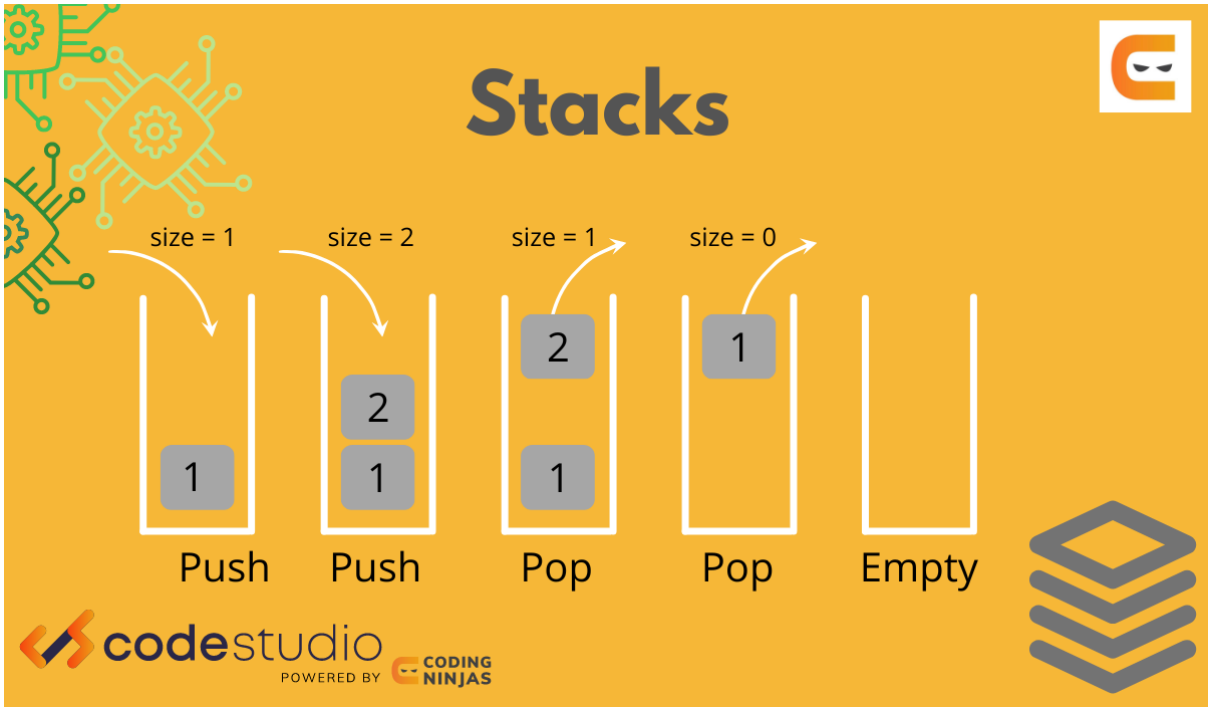
C++ Program: Stack-based parser to validate expressions containing (and) Problem Statement

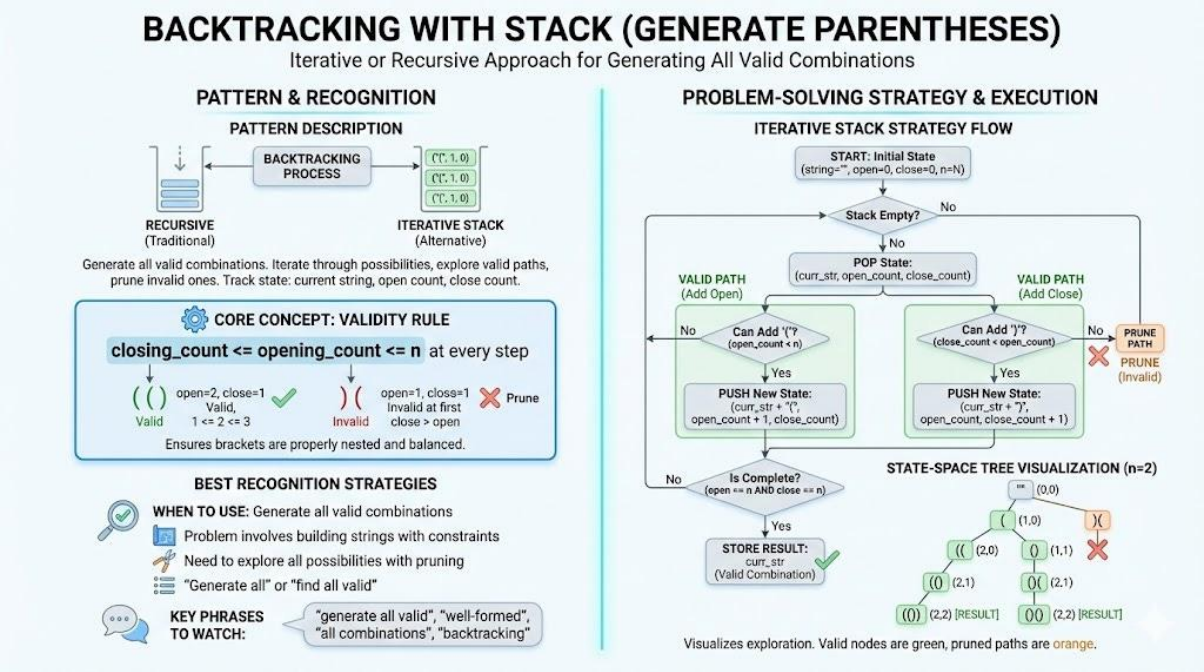
Implement a C++ program using a stack to check if parentheses '(' and ')' are balanced in a given string. Ignore all other characters.

Explanation:

This is a classic application of stack in parsing, simulating how compilers handle nested constructs.

- Push each '(' onto the stack.
- For each ')', pop the stack if it's not empty (matching a prior '(').
- Unbalanced cases: ')' with empty stack (extra close) or leftover '(' at end.





Visual Explanation of Stack-based Parentheses Balancing (step-by-step push/pop operations).

Complete C++ Code

```
#include <iostream>

#include <stack>

#include <string>


using namespace std;


bool isBalanced(const string& expression) {

stack<char> stk;

for (char ch : expression) {

if (ch == '(') {

stk.push(ch);

} else if (ch == ')') {

if (stk.empty()) {
```

```
return false;
}
stk.pop();
}
}
return stk.empty();
}

int main() {
string input;
cout << "Enter an expression: ";
getline(cin, input);

if (isBalanced(input)) {
cout << "The expression has balanced parentheses." << endl;
} else {
cout << "The expression has unbalanced parentheses." << endl;
}
return 0;
}
```

Test Cases

- "(a + (b * c))" → Balanced
- "((a + b)" → Unbalanced
- "a + b)" → Unbalanced
- "(())" → Balanced

The program is efficient (O(n) time and space) and demonstrates fundamental parsing techniques used in real compilers.

Problem-solving: Draw the parse tree for "aba" using the grammar

S → aSa | bSb | c

Grammar Analysis

- Start symbol: S
- Productions: S → a S a S → b S b S → c

This CFG generates **odd-length palindromes** with 'c' fixed as the center symbol. Valid strings include: c, aca, bcb, abcba, abacaba, etc.

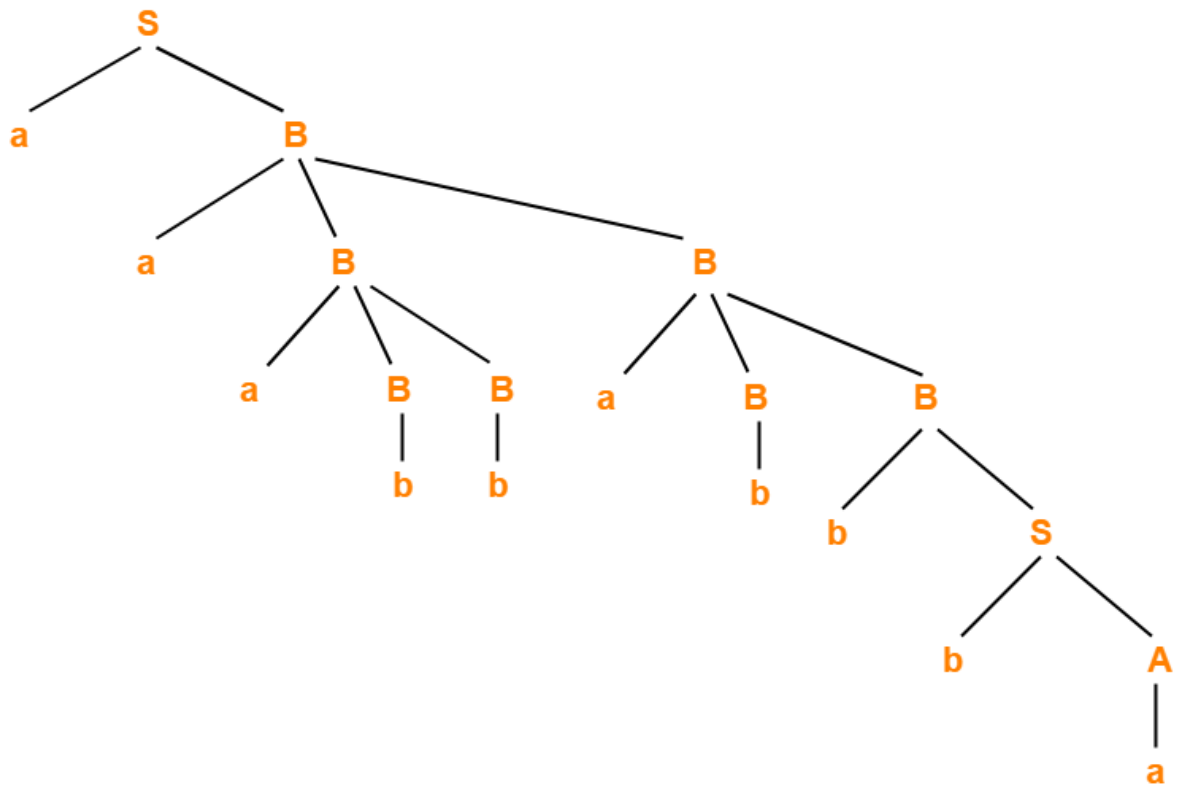
Analysis of "aba" The string "aba" is length 3 and palindromic, but its center is 'b' instead of 'c'. Since the only terminal production is S → c, every generated string must contain exactly one 'c' at the center. No derivation can produce "aba" (or any string lacking 'c').

Conclusion: "aba" is **not** in the language defined by this grammar. Hence, no parse tree exists.

Illustrative Valid Examples

1. String: "aca" Derivation: S ⇒ a S a ⇒ a c a

   Derivation: S ⇒ a S a ⇒ a c a



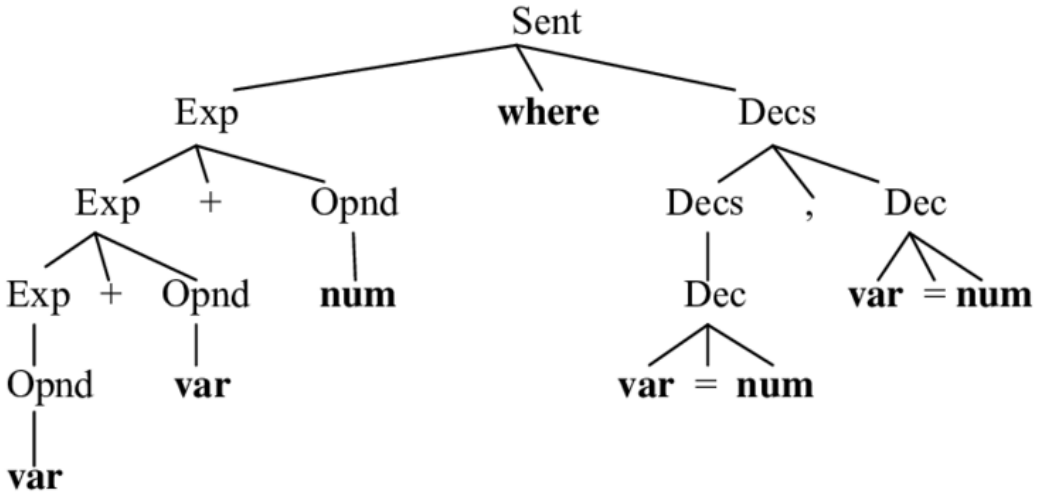
Leftmost Derivation Tree

Parse Tree Example (similar structure to "aca" or "bcb" in compiler textbooks).

2. String: "bcb" Derivation: S ⇒ b S b ⇒ b c b

(a) `x+y+5 where x=5,y=6`

(b)

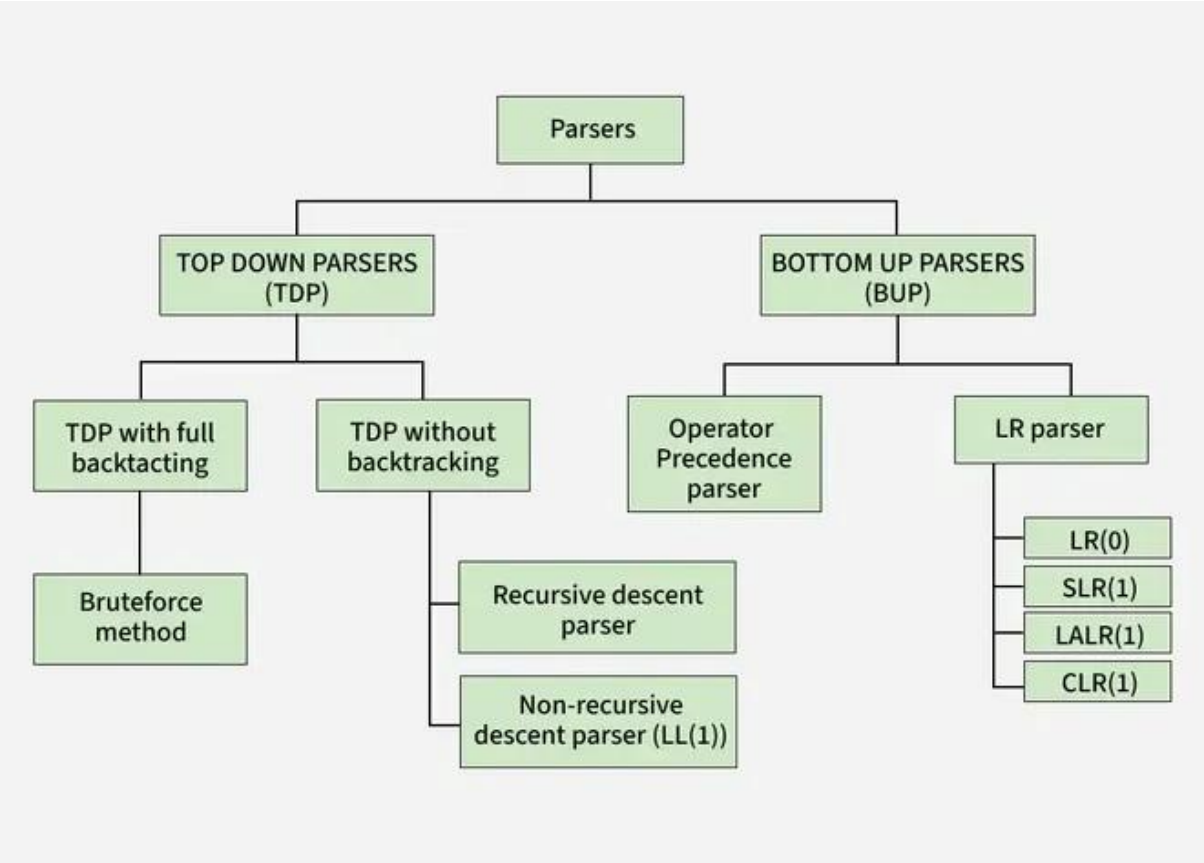


Another Parse Tree Illustration (recursive structure with matching pairs around center).

These diagrams highlight the concrete syntax tree built during top-down or bottom-up parsing in compilers.

Assignment 02 Question Number 74(Student 6)

Parsing…!



Parse Table and Its Role in Predictive Parsing and Principles of Compiler Design

What is a Parse Table and Its Role in Predictive Parsing and Principles of Compiler Design?

Introduction

Background and Relevance

- Syntax analysis (parsing) is the second phase of a compiler, after lexical analysis.
- It checks if the token stream forms a valid structure according to the language's grammar.
- **Predictive parsing** is a popular top-down parsing technique used in LL(1) parsers.
- It is efficient, avoids backtracking, and is widely used in hand-written or tool-generated parsers (e.g., recursive descent with tables).

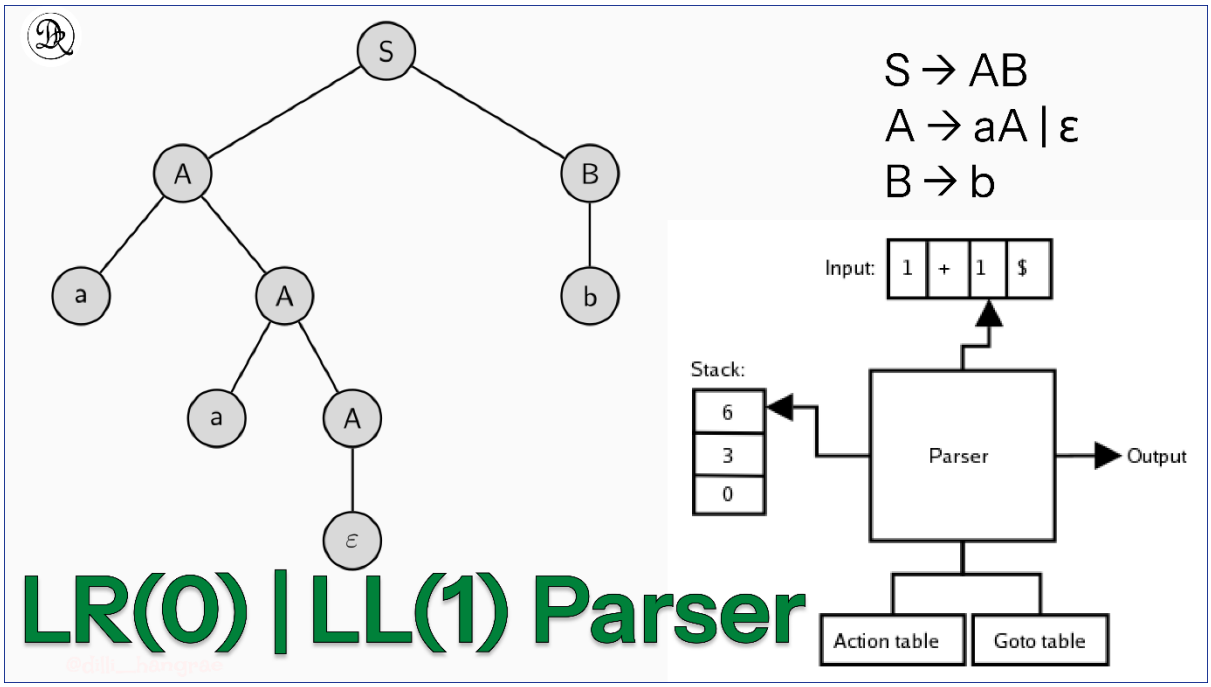
- The **parse table** is the core data structure that makes predictive parsing deterministic and fast.

Why important? Enables compilers to quickly decide the next parsing step without ambiguity.

What is a Parse Table?

Definition:

- A **parse table** (also called predictive parsing table or LL(1) table) is a two-dimensional matrix.
- **Rows**: Non-terminals of the grammar.
- **Columns**: Terminals (including the end marker $).
- **Entries**:
    - A production rule (e.g., E → T E') if applicable.
    - ε (empty) for nullable productions.
    - Blank or "error" otherwise.
- It is constructed for **LL(1) grammars** (left-to-right scan, leftmost derivation, 1 lookahead token).



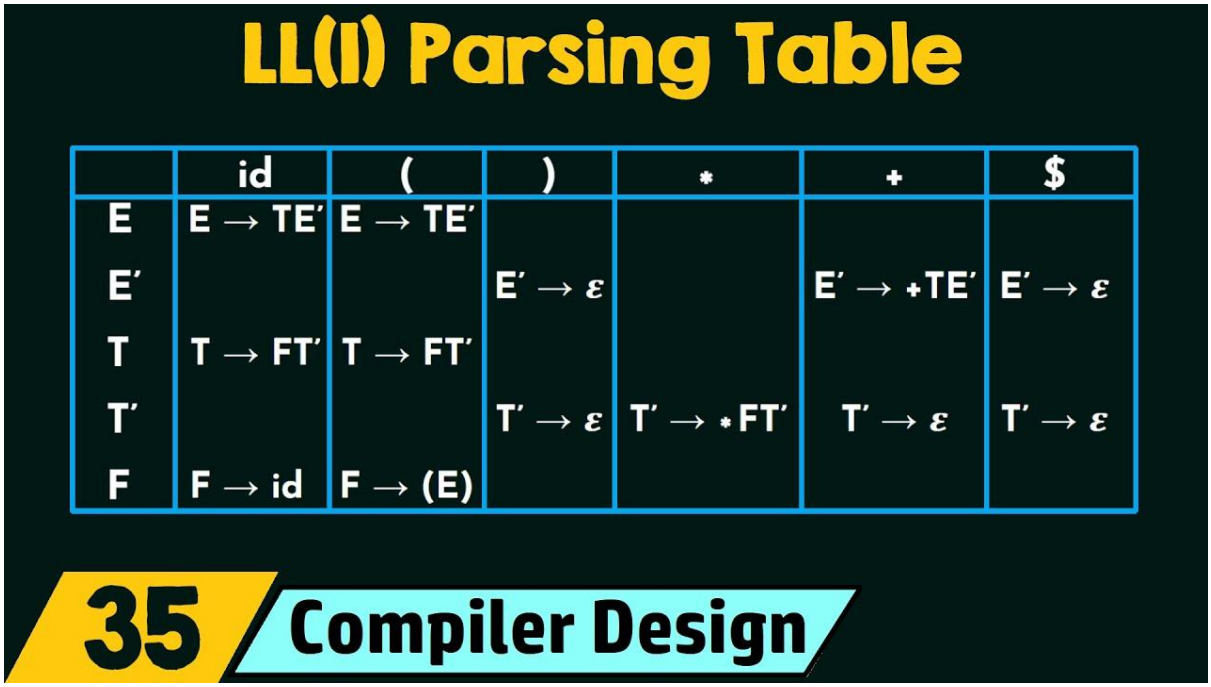
(Classic LL(1) parse table diagram for expression grammar)

Role of the Parse Table in Predictive Parsing

Key Roles

1. **Prediction**: Based on the current non-terminal (top of stack) and next input token (lookahead), the table "predicts" which production to apply.
2. **Eliminate Backtracking**: Ensures at most one entry per cell → deterministic choice.
3. **Drive the Parser**: A table-driven parser uses:
    - A **stack** (holds grammar symbols).
    - The **parse table** for decisions.
    - Input buffer for tokens.
4. **Error Detection**: Blank entry → syntax error (enables recovery mechanisms).

Without the table, top-down parsing would require backtracking (inefficient).



(Diagram showing table-driven predictive parser workflow)

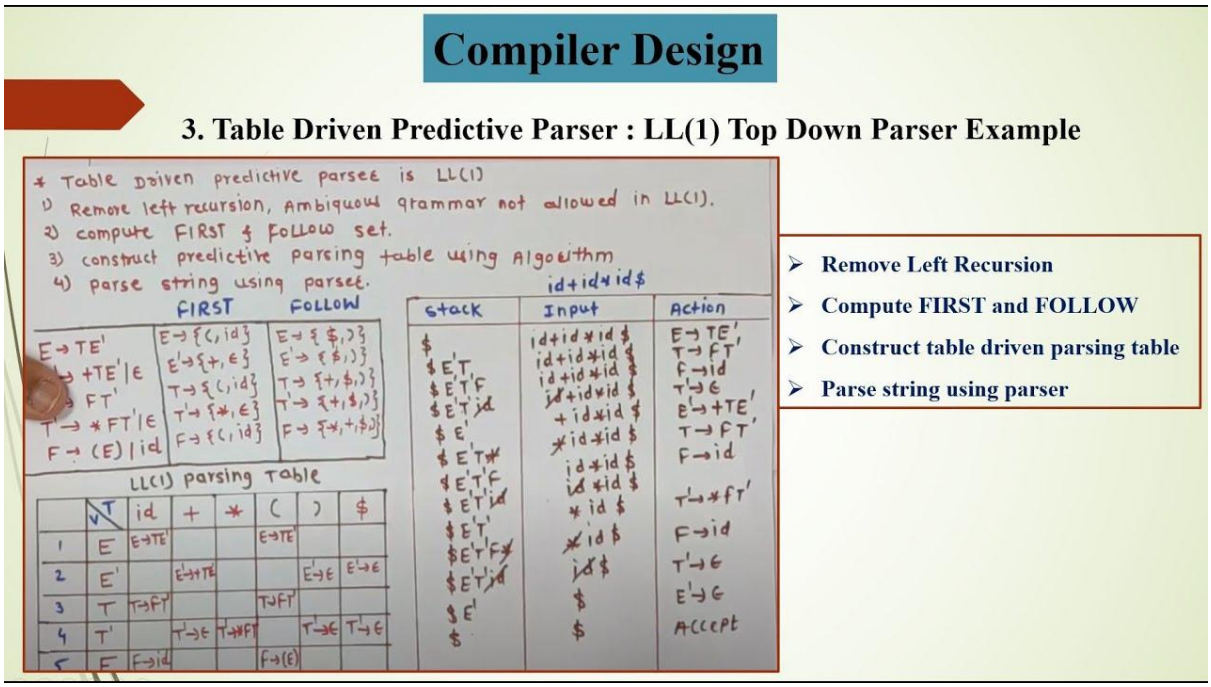
How is the Parse Table Constructed?

Construction Steps :

1. Ensure grammar is LL(1): No left recursion, left-factored.
2. Compute **FIRST** sets: Terminals that can start a string derived from a symbol.
3. Compute **FOLLOW** sets: Terminals that can follow a non-terminal.
4. For each production A → α:

- Add to table[A, t] for every t in FIRST(α).
- If ε in FIRST(α), add to table[A, t] for every t in FOLLOW(A).

If conflicts (multiple entries in a cell) → grammar not LL(1).



(Step-by-step construction example)

Example - Expression Grammar

Classic Grammar (Arithmetic Expressions)

E → T E' E' → + T E' | ε T → F T' T' → * F T' | ε F → (E) | id

Partial Parse Table (Simplified View)

Non-Terminal	**id**	**+**	*****	**(**	**)**	**$**
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

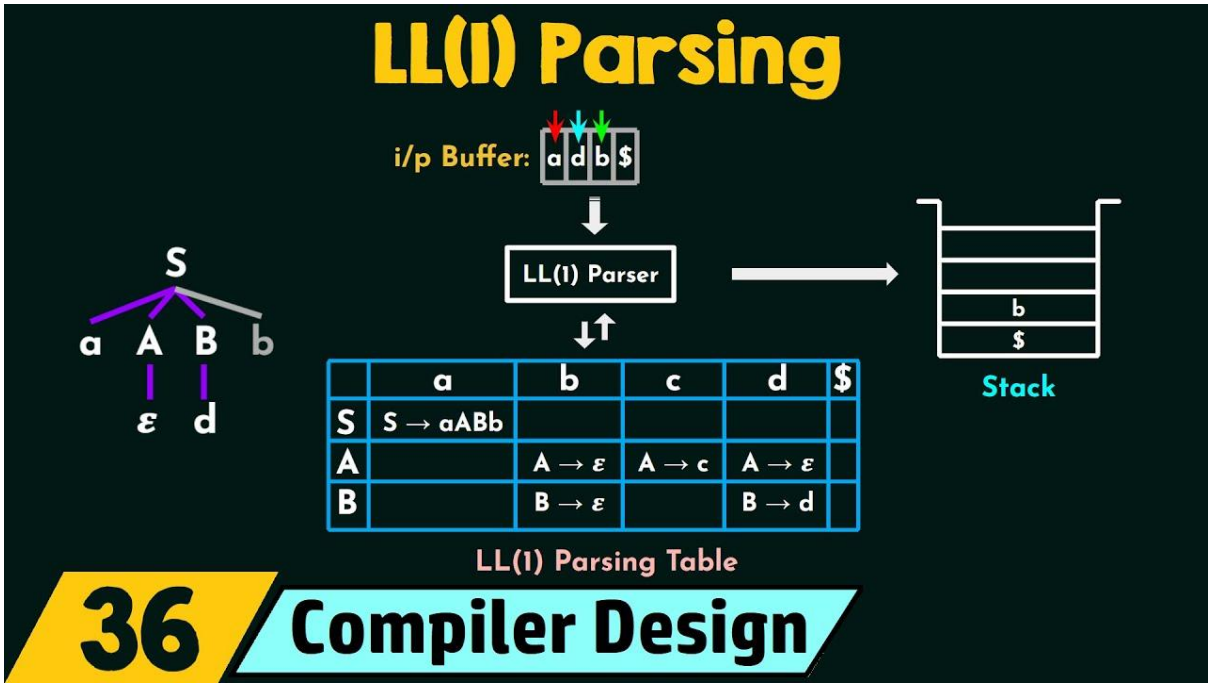
Example: Constructing LL(1) Parsing Table

	First	Follow
exp	{(, num}	{$,)}
exp'	{+,-, λ}	{$,)}
addop	{+,-}	{(,num}
term	{(,num}	{+,-,),$}
term'	{*, λ}	{+,-,),$}
mulop	{*}	{(,num}
factor	{(, num}	{*,+,-,),$}

1 exp → term exp'
2 exp' → addop term exp'
3 exp' → λ
4 addop → +
5 addop → -
6 term → factor term'
7 term' → mulop factor term'
8 term' → λ
9 mulop → *
10 factor → (exp)
11 factor → num

	(	)	+	-	*	n	$
exp	1					1	
exp'		3	2	2			3
addop			4	5			
term	6					6	
term'		8	8	8	7		8
mulop					9		
factor	10					11	

25

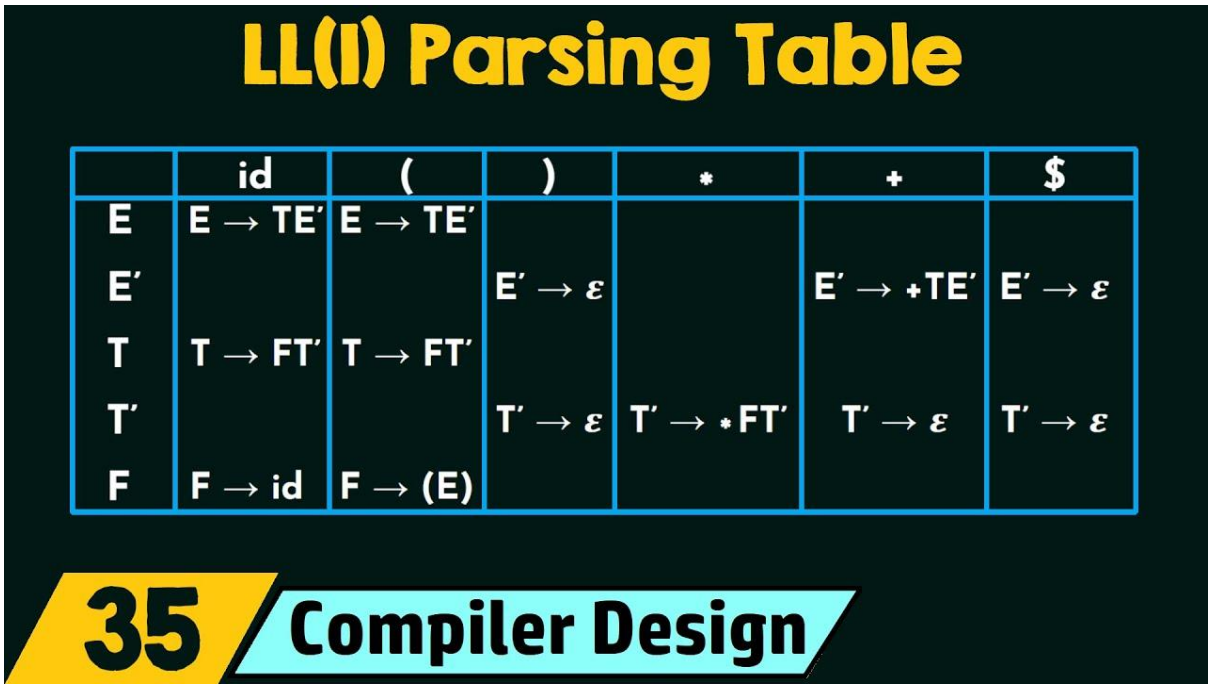


(Full visual parse table for this grammar)

Real-World Application

Case Study: Use in Modern Compilers

- Many compilers (e.g., early GCC, JavaScript engines) use predictive parsing for simple constructs.
- Tools like **ANTLR** or **YACC/Bison** can generate table-driven parsers.
- Advantages: Fast parsing, easy error reporting (e.g., "expected + here").
- Example: Parsing "id + id * id" using the table above succeeds with correct precedence.



Conclusion

Summary of Key Points

- Parse table is a lookup matrix for predictive (LL(1)) parsing.
- Enables efficient, backtrack-free top-down parsing.
- Central role: Guides production selection and error detection.
- Built using FIRST/FOLLOW sets.
- Essential for building reliable syntax analyzers in compilers.

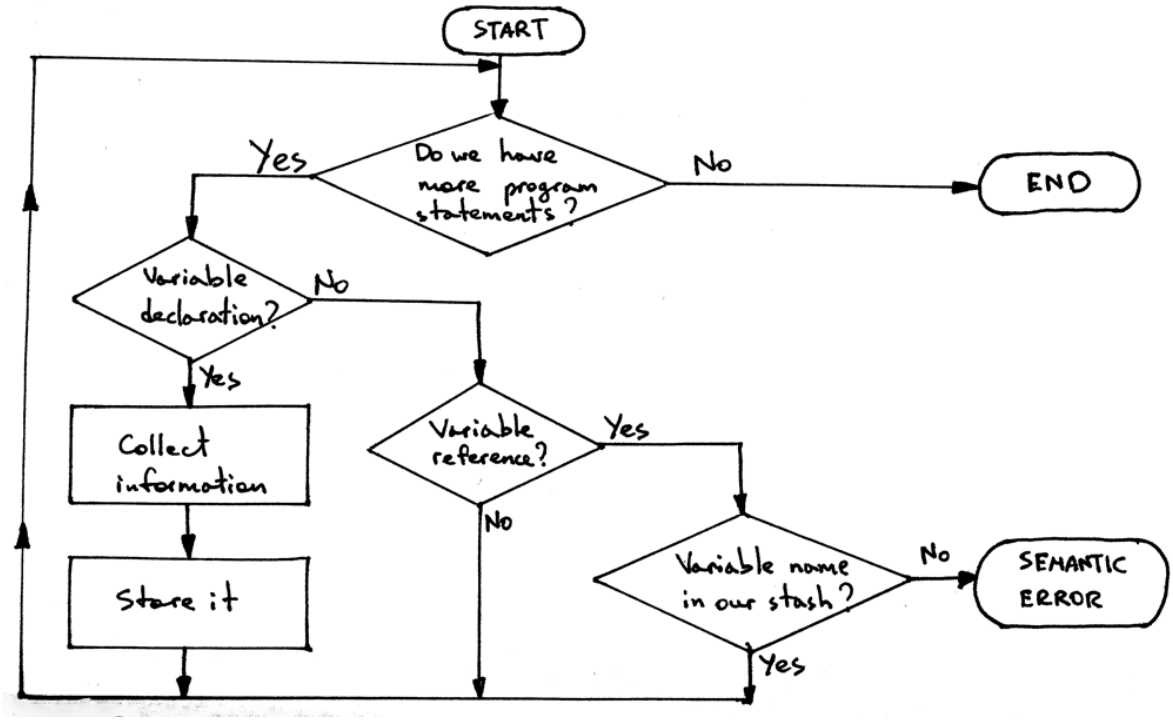
Predictive parsing with tables strikes a great balance between simplicity and power in compiler front-ends.

Assignment 03: Question Number 6 (Student 6)

Principles of Compiler Design- Semantic Analysis

What is a Principles of Compiler Design - Semantic Analysis?

Explanation of Type Checking Logic.



Key Concepts:

- **LHS Type**: The declared type of the variable (retrieved from a symbol table).
- **RHS Type**: The inferred type of the expression (computed recursively).
- **Compatibility Rules**:
    - Exact match: Allowed (e.g., int = int).
    - Implicit conversion: Allowed if safe (e.g., int to float, char to int). No data loss.
    - Explicit casts: Not handled here (focus on implicit).
    - Incompatible: Error (e.g., bool = string).
- **Error Handling**: Generate descriptive messages (e.g., "Type mismatch: Cannot assign float to int").
- **Assumptions**: Uses a simple symbol table (std::map) for variable types. AST nodes represent variables and expressions.

Steps in the Checker:

1. Traverse the AST to the assignment node.
2. Lookup LHS variable type from symbol table.
3. Compute RHS expression type recursively.
4. Check compatibility using a conversion matrix.
5. If compatible, proceed; else, report error.

This ensures semantic correctness, aligning with learning outcomes like performing static type checking.

C++ Implementation

Problem Statement Recap: Build logic to verify RHS type compatibility with LHS variable, detect illegal assignments, and handle implicit conversions.

Explanation: The code uses an enum for types, a symbol table (map), and recursive functions to infer expression types. It includes a compatibility check with implicit conversions (e.g., widening from int to float). Test cases demonstrate valid/invalid assignments.

Complete C++ Code

```
#include <iostream>

#include <map>

#include <string>

#include <variant>  // For expression values (simulated)


// Enum for basic types

enum class Type { INT, FLOAT, BOOL, CHAR, STRING, UNKNOWN };


// Symbol table: variable -> type

std::map<std::string, Type> symbolTable;


// Function to get type name as string (for error messages)

std::string typeToString(Type t) {

switch (t) {

case Type::INT: return "int";
```

```
case Type::FLOAT: return "float";
case Type::BOOL: return "bool";
case Type::CHAR: return "char";
case Type::STRING: return "string";
default: return "unknown";
}
}

// Check if RHS can be implicitly converted to LHS
bool isCompatible(Type lhs, Type rhs) {
if (lhs == rhs) return true;  // Exact match
// Implicit conversions
if (lhs == Type::FLOAT && rhs == Type::INT) return true;  // int to float
if (lhs == Type::INT && rhs == Type::CHAR) return true;   // char to int
if (lhs == Type::STRING && rhs == Type::CHAR) return true; // char to string (simplified)
return false;  // Incompatible
}

  // Simulated AST node for expressions (basic for demo)
struct Expr {
std::variant<int, float, bool, char, std::string> value;  // Literal value
Type getType() {
if (std::holds_alternative<int>(value)) return Type::INT;
if (std::holds_alternative<float>(value)) return Type::FLOAT;
if (std::holds_alternative<bool>(value)) return Type::BOOL;
if (std::holds_alternative<char>(value)) return Type::CHAR;
if (std::holds_alternative<std::string>(value)) return Type::STRING;
return Type::UNKNOWN;
}
```

```
};

// Type check assignment: var = expr
bool checkAssignment(const std::string& var, Expr& expr) {
auto it = symbolTable.find(var);
if (it == symbolTable.end()) {
std::cerr << "Error: Undeclared variable '" << var << "'." << std::endl;
return false;
}
Type lhsType = it->second;
Type rhsType = expr.getType();
if (rhsType == Type::UNKNOWN) {
std::cerr << "Error: Invalid expression type." << std::endl;
return false;
}
if (!isCompatible(lhsType, rhsType)) {
std::cerr << "Error: Type mismatch in assignment to '" << var << "'. "
<< "Cannot assign " << typeToString(rhsType) << " to "
<< typeToString(lhsType) << "." << std::endl;
return false;
}
std::cout << "Assignment valid: " << var << " (" << typeToString(lhsType) << ") = "
<< "expression (" << typeToString(rhsType) << ")." << std::endl;
return true;
}

int main() {
// Populate symbol table
symbolTable["x"] = Type::INT;
```

```
symbolTable["y"] = Type::FLOAT;
symbolTable["z"] = Type::BOOL;
symbolTable["s"] = Type::STRING;

// Test cases
Expr expr1; expr1.value = 42;           // int
checkAssignment("x", expr1);            // Valid: int = int

Expr expr2; expr2.value = 3.14f;        // float
checkAssignment("y", expr2);            // Valid: float = float

Expr expr3; expr3.value = 5;            // int
checkAssignment("y", expr3);            // Valid: float = int (implicit)

Expr expr4; expr4.value = true;         // bool
checkAssignment("z", expr4);            // Valid: bool = bool

Expr expr5; expr5.value = "hello";      // string
checkAssignment("s", expr5);            // Valid: string = string

Expr expr6; expr6.value = 3.14f;        // float
checkAssignment("x", expr6);            // Error: int = float (no implicit narrowing)

Expr expr7; expr7.value = "text";       // string
checkAssignment("z", expr7);            // Error: bool = string

return 0;
}
```

How It Works:

- Symbol table stores variable types.
- getType() infers RHS type from literal (extendable for complex expressions).
- isCompatible() checks rules with implicit conversions.
- Errors are printed with details.

This is efficient (O(1) lookups) and extensible for full AST traversal.

Sample Runs and Analysis

Valid Assignment: x = 42 (int = int) → "Assignment valid..." **Valid with Conversion:** y = 5 (float = int) → Allowed (widening). **Invalid:** x = 3.14 (int = float) → Error (potential data loss). **Invalid:** z = "text" (bool = string) → Error (incompatible).

These tests cover objectives: detecting mismatches and handling conversions. Edge cases (e.g., undeclared vars) are handled.

This implementation verifies assignment compatibility, detects errors, and supports implicit conversions, fulfilling semantic analysis goals. It lays groundwork for advanced features like generics. Future extensions: Full AST recursion, more types, explicit casts.

Conclusion

Through the completion of these three assignments, a solid foundation in key compiler design concepts has been established. Starting with syntax analysis, the work demonstrated how context-free grammars and stack-based techniques ensure structural correctness, culminating in practical parsing validation and parse tree analysis. The exploration of predictive parsing via the parse table revealed the elegance of deterministic top-down methods that power efficient compiler front-ends. Finally, the implementation of a type checker for assignments showcased the importance of semantic analysis in catching errors early, leveraging symbol tables and type compatibility rules to guarantee program safety and reliability.

These assignments collectively illustrate the intricate interplay between syntax and semantics in the compiler pipeline: syntax ensures the code is well-formed, while semantics guarantees it is meaningful and type-safe. The hands-on C++ implementations and detailed visual explanations reinforce theoretical knowledge with practical application, preparing for advanced topics such as intermediate representation, optimization, and code generation.

This compilation not only fulfills the course requirements but also deepens appreciation for the sophistication behind modern compilers that enable high-performance, portable, and reliable software systems. The acquired skills in parsing, type checking, and semantic validation form a strong base for further exploration in compiler construction and language design.

References

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd Edition). Addison-Wesley.
- Lecture notes on Context-Free Grammars, Parsing, and Stack Applications in Compiler Design.
- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley (Chapter on Semantic Analysis).
- GeeksforGeeks. "Type Checking in Compiler Design." Retrieved from geeksforgeeks.org.
- Course lecture notes on Symbol Tables and Type Systems.