**Unit III: Syntax-Directed Translation**

Syntax Directed Translation Schemes, Implementation of Syntax Directed Translators, Intermediate Code, Postfix Notation, Parse Tree & Syntax Tree, Three Address Code, Quadruples & Triples, Translation of Assignment Statements, Boolean Expressions, Statements that alters the Flow of Control, Postfix Translation, Translation with a Top Down Parser, More about Translation: Array Reference in Arithmetic Expressions, Procedure Calls, Declaration, and Case Statements.

**1. Syntax-Directed Translation Schemes (SDTS)**

Syntax-directed translation (SDT) integrates semantic rules with grammar productions to define how source language constructs are to be translated. The scheme extends context-free grammar by associating actions with grammar rules. These actions are based on **attributes**, which are values associated with the grammar's symbols. There are two types of attributes: **synthesized** (computed from child nodes) and **inherited** (computed from parent and sibling nodes). **SDDs** form the basis for compiler tasks such as type checking, intermediate representation generation, and code generation. **SDDs** can be classified as S-attributed or L-attributed, and these can be implemented using SDTs.

* **Core Components of SDT**

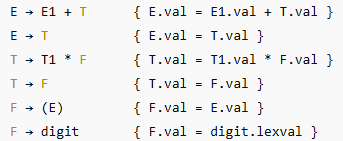
1. **Grammar Rules** – These define the syntax of a language (like E → E + T).
2. **Attributes** – Each grammar symbol has associated data called attributes.
3. **Semantic Rules** – These rules define how attributes are computed during parsing.

* **Types of Attributes**
* **Synthesized Attributes**:
  + Computed **from the children nodes** in the parse tree.
  + Commonly used in bottom-up parsing.
  + Example: computing the value of an expression.
* **Inherited Attributes**:
  + Computed **from parent or siblings** in the parse tree.
  + Often used in top-down parsing for context propagation (e.g., type declarations).
* **Syntax-Directed Definition (SDD)**

An SDD is a grammar where **each grammar rule is associated with attributes** and **semantic rules** for computing those attributes.

**Example 1: Evaluating Arithmetic Expressions**

**Grammar:**

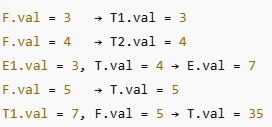
****

Here:

* val is a **synthesized attribute** that carries the numerical value of the expression.
* digit.lexval is the value (e.g., 2, 3) read from the input.

**For input: (3 + 4) \* 5**

The evaluation would be:



So the result is 35.

* **Types of SDDs**

**S-attributed:**

* Uses **only synthesized attributes**.
* Easy to implement in **bottom-up parsers** like LR parsers.
* Each attribute of a node is computed from its children.

**L-attributed:**

* Allows both **synthesized** and **a restricted form of inherited** attributes.
* Suitable for **top-down parsers** like LL parsers.
* Inherited attributes are evaluated using information from the left siblings or the parent.

**Example 2: Type Declaration Using Inherited Attributes**

**Grammar:**

D → T L

T → int { T.type = int }

T → float { T.type = float }

L → L1 , id { id.type = L1.type }

L → id { id.type = L.inh }

Here, type is **inherited** from T to the list of identifiers in L.

**For input:**

int a, b;

* T.type = int
* L.inh = T.type = int
* a.type = int, b.type = int

This mechanism helps populate the **symbol table** with types during parsing.

* **Use Cases of SDTs**
* Expression evaluation
* Type checking
* Intermediate code generation
* Symbol table management
* Error detection (e.g., undeclared variables or type mismatches)
* **Key Benefits**
* Provides a modular, structured way to attach meaning to syntax.
* Integrates semantic analysis and code generation into parsing.
* Supports multiple compiler phases in a clean, attribute-driven way.
* **Visual Illustration of Synthesized Evaluation**

A screenshot of a computer screen

AI-generated content may be incorrect.

Evaluation flow (bottom-up):

* F → 3, F → 4, F → 5
* T → 4, T → 5
* E1 → 7 (3+4), then T → 5
* E → 7 \* 5 → 35

**2. Implementation of Syntax Directed Translators**

The implementation of syntax-directed translators involves combining parsing with semantic analysis and code generation. Typically, a parse tree or syntax tree is used to evaluate the attribute values. Synthesized attributes can be evaluated using **post-order traversal** of the syntax tree, as they depend on child nodes. Inherited attributes, however, may require building **a dependency graph** to determine a valid evaluation order, especially when values depend on parent or sibling nodes. In practice, compilers use **syntax-directed translations (SDTs)**, where **semantic actions** are embedded within parsing rules, to perform semantic analysis and generate intermediate code during parsing. These translators can be implemented using either **bottom-up parsers** (like LR parsers that use parse stacks) or **top-down parsers** (like recursive descent), depending on the grammar and attribute types used.

The implementation of **syntax-directed translators** is a fundamental part of compiler design, focusing on how **semantic rules** are evaluated in tandem with **parsing** to carry out operations like **type checking**, **symbol table construction**, and **intermediate code generation**.

This process integrates three main components:

* **Parsing** (syntax analysis)
* **Semantic analysis** (attribute evaluation)
* **Translation** (e.g., code generation)
* **Attribute Evaluation Mechanism**

To implement a syntax-directed translator, you need to determine how and when each **attribute** of a grammar symbol is evaluated.

There are two main types of attributes:

* **Synthesized attributes**: Computed from child nodes (bottom-up).
* **Inherited attributes**: Computed from parent or sibling nodes (top-down or sideways).
* **Evaluation Order and Traversals**

**Post-order Traversal for Synthesized Attributes**

When parsing is complete, you can evaluate synthesized attributes by traversing the **parse tree** or **abstract syntax tree (AST)** in **post-order**. This ensures that child nodes are processed before their parents.

**Dependency Graphs for Inherited Attributes**

For grammars involving **inherited attributes**, the evaluation order becomes more complex. A **dependency graph** is constructed where:

* **Nodes** represent attributes.
* **Edges** show dependencies.

Topological sorting of this graph helps determine a valid order of evaluation to avoid cyclic dependencies.

**Syntax-Directed Translation (SDT) -** In practical implementations, instead of constructing full parse trees and then evaluating attributes, most compilers use **Syntax-Directed Translation (SDT)**. In SDT:

* Semantic actions are embedded directly within grammar rules.
* These actions are executed **during parsing**.
* This enables **on-the-fly translation**, such as generating code as parsing progresses.

**Example (Postfix Translation):**

E → E1 + T { print('+') }

E → T

T → digit { print(digit.val) }

Input: 3 + 4  
Output: 3 4 + → Postfix representation generated while parsing.

* **Types of Parsers and Integration Strategies**

**Bottom-Up Parsers (e.g., LR Parsers)**

* Use a **parse stack** to manage symbols and attributes.
* Actions are performed **after reductions** (when a production is matched).
* Synthesized attributes work best here.
* Good for **S-attributed grammars**.

**Top-Down Parsers (e.g., Recursive Descent)**

* Implemented using **functions for each non-terminal**.
* Inherited attributes can be **passed as function parameters**.
* Semantic actions are placed **during expansions**.
* Useful for **L-attributed grammars**.
* **Arithmetic Expression Evaluation with SDT**

**Grammar with Actions:**

E → E1 + T { E.val = E1.val + T.val }

E → T { E.val = T.val }

T → digit { T.val = digit.lexval }

**Input: 2 + 3**

During parsing:

* T.val = 2
* T.val = 3
* E.val = 2 + 3 = 5

Thus, the attribute values are evaluated **as rules are applied**, and no explicit tree traversal is required.

* **Advantages of Syntax-Directed Translator Implementations**
* Integrates semantic analysis with parsing.
* Enables **incremental evaluation**, especially useful in **interpreters**.
* Reduces overhead by **avoiding full parse tree storage**.
* Supports **code generation**, **type checking**, and **error handling** inline.
* **Real-World Use**

Modern compilers (like GCC, LLVM frontends) use syntax-directed definitions embedded within parser generators (like YACC, Bison, ANTLR) to build translators that emit intermediate representations (IR) or bytecode.

**3. Intermediate Code**

Intermediate code serves as a bridge between the high-level source language and the low-level target machine code. It is independent of machine architecture and is easier to optimize. Common forms of intermediate code include **Three Address Code (TAC)**, **Postfix Notation**, and **Quadruples or Triples**. The goal of intermediate code is to preserve the logical meaning of the program while simplifying complex constructs into simpler and linear instructions. Because of its structure, intermediate code is particularly well-suited for optimization techniques like constant folding, dead code elimination, and loop unrolling.

**Intermediate Code (IC)** is a crucial phase in the compiler design process. It serves as a middle layer between the high-level source language (like C, Java, or Python) and the target machine code (assembly or binary). The primary goal of IC is to facilitate portability, simplify code generation, and enable effective optimizations.

**Key Characteristics:**

* **Machine-independent**: IC abstracts away specific machine architecture details, making it easier to retarget the compiler for different platforms.
* **Simplified syntax**: High-level constructs are broken down into linear, atomic operations.
* **Structured for analysis**: IC is easier to analyze and transform for optimization purposes.

**Importance:**

* Helps with **code portability** across different architectures.
* Facilitates **early-stage optimization** like:
  + Constant folding
  + Strength reduction
  + Dead code elimination
  + Loop unrolling

**Common Forms of Intermediate Code:**

1. **Three Address Code (TAC)**  
   Each statement involves at most three operands. Typically takes the form:  
   x = y op z  
   Example for a = b + c \* d:

t1 = c \* d

t2 = b + t1

a = t2

1. **Postfix Notation**  
   Operators follow operands (more on this in section 4). It is especially useful for evaluation using stacks.
2. **Quadruples**  
   Each instruction is represented by a 4-tuple:  
   (operator, arg1, arg2, result)  
   Example:  
   (+, b, t1, a)
3. **Triples**  
   Similar to quadruples but omit the result field. The result is implied by the position of the instruction.  
   Example:  
   (+ , b , ( \* , c , d ))

Each of these forms serves different roles depending on the stage of compilation and the specific optimizations being applied.

**4. Postfix Notation**

Postfix notation, also called **Reverse Polish Notation (RPN)**, places operators after their operands. For example, the infix expression a + b becomes a b + in postfix. This notation eliminates the need for parentheses and follows a straightforward evaluation using a stack. The postfix form is especially useful for compilers and interpreters as it simplifies the parsing and evaluation of arithmetic expressions, making it efficient for stack-based machines and intermediate code generation.

**Structure:**

In postfix, the usual order of an arithmetic expression is reversed in terms of operator placement.  
**Infix**: a + b  
**Postfix**: a b +

**Key Properties:**

* **No need for parentheses**: The order of operations is preserved inherently.
* **Left-to-right evaluation** using a stack.
* **Efficient execution** on stack machines (used in interpreters and compilers).
* **Simple parsing** rules compared to infix expressions which require precedence rules.

**Conversion Example:**

Expression: (a + b) \* (c - d)  
Step-by-step conversion:

1. Infix: (a + b) \* (c - d)
2. Postfix: a b + c d - \*

**Evaluation Example (using a stack):**

Postfix: 3 4 + 5 \*

1. Push 3 → Stack: [3]
2. Push 4 → Stack: [3, 4]
3. Encounter + → Pop 4 and 3 → Compute 3 + 4 = 7 → Push 7
4. Push 5 → Stack: [7, 5]
5. Encounter \* → Pop 5 and 7 → Compute 7 \* 5 = 35 → Push 35  
   Final result: 35

**Use in Compilation:**

* Compilers use postfix to represent intermediate forms of expressions.
* Stack-based code generators directly translate postfix to instructions like:

PUSH 3

PUSH 4

ADD

PUSH 5

MUL

This makes postfix notation an essential tool in both expression parsing and code generation.

**Infix to Postfix Conversion**

**Basic Idea**

The **infix expression** (e.g., a + b \* c) is how humans write math, but it's difficult for machines to parse because of operator precedence and associativity.  
**Postfix notation** removes the need for parentheses and follows a consistent, linear structure for evaluation.

**Operator Precedence and Associativity**

| **Operator** | **Precedence** | **Associativity** |
| --- | --- | --- |
| (, ) | Highest | — |
| \*, / | High | Left-to-right |
| +, - | Medium | Left-to-right |

**Infix to Postfix Conversion Algorithm (Using Stack)**

1. Initialize an empty **stack** for operators and an empty **output list**.
2. **Scan the infix expression** from left to right:
   * **Operand** (like a, b, 1, x): Add directly to the output.
   * **Left Parenthesis (**: Push onto the stack.
   * **Right Parenthesis )**: Pop from the stack and add to output until ( is encountered (discard both).
   * **Operator**:  
     a. While there is an operator at the top of the stack with **higher or equal precedence**, pop it to output.  
     b. Push the current operator to the stack.
3. After scanning the input, **pop all remaining operators** from the stack to the output.

**Example 1: a + b \* c**

**Step-by-step:**

| **Symbol** | **Stack** | **Output** |
| --- | --- | --- |
| a |  | a |
| + | + | a |
| b | + | a b |
| \* | + \* | a b |
| c | + \* | a b c |
| end |  | a b c \* + |

**Postfix**: a b c \* +

**Example 2: (a + b) \* (c - d)**

| **Symbol** | **Stack** | **Output** |
| --- | --- | --- |
| ( | ( |  |
| a | ( | a |
| + | ( + | a |
| b | ( + | a b |
| ) |  | a b + |
| \* | \* | a b + |
| ( | \* ( | a b + |
| c | \* ( | a b + c |
| - | \* ( - | a b + c |
| d | \* ( - | a b + c d |
| ) | \* | a b + c d - |
| end |  | a b + c d - \* |

**Postfix**: a b + c d - \*

**Example 3: a + (b \* c - d) / e**

| **Symbol** | **Stack** | **Output** |
| --- | --- | --- |
| a |  | a |
| + | + | a |
| ( | + ( | a |
| b | + ( | a b |
| \* | + ( \* | a b |
| c | + ( \* | a b c |
| - | + ( - | a b c \* |
| d | + ( - | a b c \* d |
| ) | + | a b c \* d - |
| / | + / | a b c \* d - |
| e | + / | a b c \* d - e |
| end |  | a b c \* d - e / + |

**Postfix**: a b c \* d - e / +

**5. Parse Tree and Syntax Tree**

A **parse tree** is a hierarchical representation that reflects the syntactic structure of a string according to a context-free grammar. It includes every non-terminal and terminal symbol used in the derivation. However, because parse trees often contain redundant nodes and details, they are not very efficient for semantic analysis. A **syntax tree**, in contrast, is a condensed version that omits unnecessary grammar rules and focuses on the essential structure of the source code. For example, in the expression a + b \* c, the parse tree would reflect the order of operations and all rule applications, while the syntax tree would more cleanly show + as the root, with a and the subtree for b \* c as children, clearly expressing the precedence of \* over +.

**Parse Tree (Concrete Syntax Tree)**

A **parse tree** is a tree structure that shows how a string is derived from a grammar by repeatedly applying production rules. It includes every single detail from the grammar, including both **non-terminal and terminal symbols**.

**Features:**

* Represents the **full derivation** from the start symbol to the input string.
* Each **internal node** is a **non-terminal**, and each **leaf node** is a **terminal**.
* Demonstrates the **complete syntactic structure** according to the grammar.

**Use:**

* Helpful in validating whether an input string belongs to the language defined by a grammar.
* Not ideal for translation because of its verbosity.

**Syntax Tree (Abstract Syntax Tree - AST)**

A **syntax tree** is a **simplified version** of a parse tree. It abstracts away unnecessary non-terminals and focuses on the **logical structure** of the code. It captures the **essential constructs** and **semantic hierarchy** of an expression or statement.

**Features:**

* Internal nodes represent **operators** or **constructs** (like +, \*, if, etc.).
* Leaf nodes represent **operands** or **identifiers** (like a, b, 3, etc.).
* Easier to **evaluate**, **optimize**, and **generate code from**.

**Use:**

* Used in **semantic analysis**, **intermediate code generation**, and **optimizations**.
* Helps capture **operator precedence** and **associativity** clearly.

**Example: Expression a + b \* c**

**1. Parse Tree (Assume Grammar):**

E → E + T

E → T

T → T \* F

T → F

F → id

**Derivation Steps:**

E → E + T

→ T + T

→ F + T

→ id + T

→ id + T \* F

→ id + F \* F

→ id + id \* id

**Parse Tree:**

A screenshot of a computer screen

AI-generated content may be incorrect.

This tree shows **all non-terminals** and **each rule application**. It is **complete but verbose**.

**2. Syntax Tree for a + b \* c**

A black and red text

AI-generated content may be incorrect.

**Explanation:**

* The \* operator is evaluated first, as it has **higher precedence**, and becomes the **right child** of +.
* This tree removes intermediate non-terminals like E, T, and F.

**Key Differences Between Parse Tree and Syntax Tree**

| **Feature** | **Parse Tree** | **Syntax Tree** |
| --- | --- | --- |
| Structure | Complete derivation | Simplified, abstracted |
| Includes non-terminals | Yes | No |
| Used for | Syntax checking | Semantic analysis, code generation |
| Shows precedence | Implicitly through tree depth | Explicit through tree structure |
| Size | Larger and more complex | Smaller and more readable |
| Efficiency | Less efficient for code translation | Highly efficient for translation |

* **Parse Trees** are useful for validating syntax but are cumbersome for translation.
* **Syntax Trees** are essential for the back-end of compilers, where meaning, structure, and execution logic must be preserved efficiently.

**6. Three-Address Code (TAC)**

**Three-Address Code (TAC)** is a type of **intermediate representation (IR)** used in the design of compilers. It simplifies complex source-level expressions and statements into a linear sequence of instructions, each involving **at most three operands**.

**General Format of TAC:**

x = y op z

Where:

* x is the result
* y and z are operands (variables or constants)
* op is a binary operator (+, -, \*, /, etc.)

This format makes it easier to:

* Perform **machine-independent optimizations**
* Generate **target machine code**
* Perform **data flow analysis** and **register allocation**

**Why Use TAC?**

* **Decomposes complex expressions** into atomic operations.
* Enables better **optimization** and **code scheduling**.
* Facilitates **target-independent code generation**.
* Compatible with various code representations: **quadruples**, **triples**, or **indirect triples**.

**Example: Converting a = b + c \* d to TAC**

**Step-by-step TAC:**

t1 = c \* d

t2 = b + t1

a = t2

Here:

* Intermediate variables (t1, t2) store temporary results.
* Every step contains only one operator and simple operands.

**Types of TAC Instructions**

**1. Assignment**

x = y

Simple value copy.

**2. Binary Operations**

x = y + z

x = y \* z

Includes arithmetic and logical operations.

**3. Unary Operations**

x = -y

x = !y

Negation, logical NOT, etc.

**4. Conditional Jump**

if x < y goto L1

Used for if, while, for, etc.

**5. Unconditional Jump**

goto L2

For flow control and labels.

**6. Function Calls**

param a

call foo, 1

result = RET

**7. Array Access**

t1 = i \* 4

t2 = A[t1]

**8. Pointer or Memory Operations**

x = \*y // Load from address

\*y = x // Store to address

**TAC for Common Control Constructs**

**If-Else Example:**

if (x < y)

z = x;

else

z = y;

**TAC:**

if x >= y goto L1

z = x

goto L2

L1: z = y

L2:

**While Loop Example:**

while (a < b)

a = a + 1;

**TAC:**

L1: if a >= b goto L2

a = a + 1

goto L1

L2:

**Benefits of TAC**

* Easier to implement in **interpreter or virtual machines**.
* Enables **constant propagation**, **dead code elimination**, and **loop optimization**.
* Intermediate step between parsing and machine-specific code generation.

Three-Address Code is a powerful intermediate format used throughout the **middle and backend** phases of a compiler. It allows developers and tools to break down, optimize, and regenerate complex code into highly efficient low-level instructions.

**7. Quadruples and Triples**

In compiler design, **quadruples** and **triples** are two commonly used **data structures to represent three-address code (TAC)** instructions in an intermediate format. These representations are particularly useful during optimization and code generation phases.

**A. Quadruples**

A **quadruple** represents each TAC instruction as a **4-tuple**:

scss

CopyEdit

(op, arg1, arg2, result)

* **op**: Operator (e.g., +, -, \*, etc.)
* **arg1, arg2**: Operands (could be constants, variables, or temporary results)
* **result**: Variable or temporary where the result is stored

**Example:**

Statement: x = y + z  
Quadruple:

(+, y, z, x)

This explicitly states that the addition of y and z is stored in x.

**B. Triples**

A **triple** represents each TAC instruction as a **3-tuple**:

(op, arg1, arg2)

* The **result is not named**, but is **implied by the position (index)** of the triple in the list.
* Subsequent triples can **refer to earlier results** by their index.

**Example:**

Statement: x = y + z  
Triple:

0: (+, y, z)

If another instruction uses this result, it might appear like:

1: (\*, (0), w)

This means: result = result of triple 0 multiplied by w.

**Comparison:**

| **Feature** | **Quadruples** | **Triples** |
| --- | --- | --- |
| Result Field | Explicit (4th element) | Implicit (position index) |
| Easy to Rearrange | Yes (independent of position) | No (results are position-based) |
| Space Efficiency | Slightly more | Slightly less |
| Optimization Ease | Higher | Moderate |

**When to Use:**

* Use **quadruples** when:
  + Code optimization and reordering are important
  + Need clear identification of temporary variables
* Use **triples** when:
  + You want to save space
  + Code is small or no rearrangement is expected

**8. Translation of Assignment Statements**

Assignment statements are fundamental in most programming languages and must be translated carefully during compilation. These statements often involve **expressions** that require honoring **operator precedence** and **associativity**.

**General Structure:**

a = b + c \* d;

To generate correct intermediate code, the compiler must:

* Evaluate c \* d first (**higher precedence**)
* Then evaluate b + result
* Finally, assign the final result to a

**Translation Using TAC:**

t1 = c \* d

t2 = b + t1

a = t2

**Explanation of Steps:**

1. t1 = c \* d  
   Multiplies c and d and stores the result in a temporary variable t1.
2. t2 = b + t1  
   Adds b and t1, storing it in another temporary variable t2.
3. a = t2  
   Assigns the final result to variable a.

**Other Examples:**

**Example 1:**

x = (a + b) \* (c - d);

TAC:

t1 = a + b

t2 = c - d

t3 = t1 \* t2

x = t3

**Example 2:**

m = x - y / z;

TAC:

ini

CopyEdit

t1 = y / z

t2 = x - t1

m = t2

**Significance:**

* Accurate translation ensures **semantic correctness** of the program.
* The breakdown allows the compiler to **optimize individual steps**.
* Simplifies generation of **target machine instructions**.

explanation of the example a = b + c \* d using **Postfix**, **Three-Address Code (TAC)**, **Quadruples**, **Triples**, and its **Syntax Tree**.

**Expression**

a = b + c \* d;

* **Operator Precedence**
* Multiplication (\*) has **higher precedence** than addition (+)
* So, evaluate c \* d first, then add to b, and finally assign to a
* **Postfix Notation (Reverse Polish Notation)**

Convert infix to postfix:

Infix: a = b + c \* d

Right-hand side: b + (c \* d)

Postfix of RHS: b c d \* +

Full expression (if assignment is also included): b c d \* + a =

* **Three-Address Code (TAC)**

Break into atomic operations:

t1 = c \* d

t2 = b + t1

a = t2

Each instruction:

* has at most 3 operands
* uses temporary variables (t1, t2) to store intermediate results
* **Quadruples Representation**

Quadruples format:

(op, arg1, arg2, result)

For our TAC:

( \*, c, d, t1 )

( +, b, t1, t2 )

( =, t2, -, a ) // assignment is treated like a binary op where second arg is empty

* **Triples Representation**

Triples format:

(index: (op, arg1, arg2))

Here, results are implied by their position:

0: ( \*, c, d ) // t1

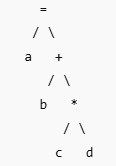
1: ( +, b, (0) ) // t2

2: ( =, (1), - ) // a = t2

References like (0) and (1) point to results from previous steps.

* **Syntax Tree**

Shows the **structure** of the expression:



* \* is evaluated first, forming the right subtree.
* + combines b and c\*d
* = assigns the result to a

This tree directly captures the **precedence** and **associativity** of the operators.

**Summary Table**

| **Representation** | **Format/Example** |
| --- | --- |
| **Postfix** | b c d \* + |
| **TAC** | t1 = c \* d t2 = b + t1 a = t2 |
| **Quadruples** | ( \*, c, d, t1 ) ( +, b, t1, t2 ) ( =, t2, -, a ) |
| **Triples** | 0: ( \*, c, d ) 1: ( +, b, (0) ) 2: ( =, (1), - ) |
| **Syntax Tree** | = → a and (+ → b, (\* → c, d)) |

**9. Boolean Expressions**

Boolean expressions are critical in control flow constructs like if, while, and for. They may use logical operators like AND, OR, and NOT. There are two main ways to translate these expressions: **jumping code** and **boolean code**. Jumping code uses conditional and unconditional jumps to represent logical flow. For example:

if a < b and c > d then ...

**would be translated into:**

if a < b goto L1

goto Lfalse

L1: if c > d goto Ltrue

goto Lfalse

This approach is efficient as it implements **short-circuit evaluation**, meaning evaluation stops as soon as the result is known.

**Role in Control Structures**

* Found in conditions like if (a < b), while (x != y), or for (i < 10)
* May involve logical operators like && (AND), || (OR), and ! (NOT)

**Translation Approaches**

There are **two main ways** to translate boolean expressions:

* **Boolean Code** (evaluate and store result)
* **Jumping Code** (direct use of control flow and labels)

**Boolean Code Approach**

* Translates expressions into instructions that compute boolean values (true or false)
* Stores the result in a temporary variable
* Not optimal for control structures or short-circuit evaluation

**Example:**

t1 = a < b

t2 = c > d

t3 = t1 && t2

This method evaluates both subexpressions regardless of their values.

**Jumping Code Approach (Short-Circuiting)**

* Generates conditional and unconditional **jump instructions**
* Efficient, especially when used in control flow structures
* Allows **short-circuit evaluation**: evaluation stops as soon as the final result is known

**Example: if (a < b && c > d)**

**Jumping Code:**

if a < b goto L1

goto Lfalse

L1: if c > d goto Ltrue

goto Lfalse

* If a < b is false, it jumps directly to Lfalse without checking c > d
* Only if a < b is true does it proceed to evaluate c > d

**Translation Templates**

* **Logical AND (&&):**
  + If the first condition is false, no need to check the second
  + Efficiently skips evaluation using jump
  + Example:

if E1 goto L1

goto Lfalse

L1: if E2 goto Ltrue

goto Lfalse

* **Logical OR (||):**
  + If the first condition is true, the result is true
  + Skip second evaluation if not needed
  + Example:

if E1 goto Ltrue

if E2 goto Ltrue

goto Lfalse

* **Logical NOT (!):**
  + Reverse the branches for the subexpression
  + Example:

if !E1 → becomes:

if E1 goto Lfalse

goto Ltrue

**Example: while (x < y && z != 0)**

Translated to jumping code:

Lstart:

if x < y goto L1

goto Lend

L1:

if z != 0 goto Lbody

goto Lend

Lbody:

...loop body...

goto Lstart

Lend:

**Benefits of Jumping Code**

* Reduces unnecessary computation
* Keeps control flow close to the source language semantics
* Essential for optimizing short-circuit boolean logic

**10. Statements that Alter the Flow of Control**

Control flow statements are used to **change the normal sequential execution** of code. These include:

* if, if-else
* while, do-while
* for loops
* switch-case
* break, continue, goto, and return

During translation, these require **generation of labels** and **jump instructions** to represent branching and looping behavior.

**Translation Technique: Using Labels and Jumps**

Control flow translation relies heavily on:

* **Conditional jumps** like if x < y goto L1
* **Unconditional jumps** like goto L2
* **Labels** (L1, L2, etc.) to mark destinations

This mimics how assembly or machine code handles jumps and branches.

**Example 1: if-else Statement**

if (a > b)

x = 1;

else

x = 2;

**TAC Translation:**

if a <= b goto L1 // if condition fails, go to else part

x = 1 // then part

goto L2 // skip else after then executes

L1: x = 2 // else part

L2: // join point (next statement after if-else)

**Example 2: while Loop**

while (x < y) {

x = x + 1;

}

**TAC Translation:**

L1: if x >= y goto L2

x = x + 1

goto L1

L2:

* L1 is the beginning of the loop.
* If the condition fails, jump to L2 (end of loop).
* Otherwise, execute the body and jump back to L1.

**Example 3: for Loop**

for (i = 0; i < n; i++) {

sum = sum + A[i];

}

**TAC Translation:**

i = 0

L1: if i >= n goto L2

t1 = i \* 4 // assuming 4-byte ints

t2 = A[t1]

sum = sum + t2

i = i + 1

goto L1

L2:

* Initialization (i = 0)
* Test condition (i < n)
* Loop body
* Increment (i = i + 1)
* Repeat

**Example 4: switch-case**

switch(ch) {

case 1: x = 10; break;

case 2: x = 20; break;

default: x = 0;

}

**TAC Translation:**

t1 = ch

if t1 == 1 goto L1

if t1 == 2 goto L2

goto L3

L1: x = 10

goto Lend

L2: x = 20

goto Lend

L3: x = 0

Lend:

* Conditional jumps for each case
* A jump table or comparison ladder
* Lend as a common exit point

**Summary**

| **Construct** | **Technique Used** |
| --- | --- |
| if | Conditional jump |
| if-else | Conditional jump + goto + labels |
| while | Loop label, condition check, backjump |
| for | Similar to while, with init & increment |
| switch | Multiple conditional jumps or jump table |

**11. Postfix Translation**

**Postfix notation** is a mathematical expression format where **operators follow operands**. It is extensively used in compilers and interpreters for **stack-based evaluation** of expressions.

**Why Postfix?**

* **No parentheses** are needed — the order of operations is preserved inherently.
* **No operator precedence table** is required once conversion is done.
* Ideal for **stack machines** and **intermediate code generation** in compilers.
* Simple and efficient for **expression evaluation** at runtime.

**Infix vs Postfix:**

| **Infix Expression** | **Postfix Equivalent** |
| --- | --- |
| a + b | a b + |
| a + b \* c | a b c \* + |
| (a + b) \* (c - d) | a b + c d - \* |

**How Postfix Translation Works**

The compiler translates infix expressions into postfix by:

* Recursively parsing subexpressions.
* **Placing operators after operands**.
* Eliminating the need for parentheses by maintaining **operator precedence**.

This is commonly implemented using **syntax-directed definitions (SDDs)** or **recursive descent parsers**.

**Postfix Generation Rule (Recursive)**

Consider the grammar:

E → E + T | T

T → T \* F | F

F → (E) | id

Add semantic rules:

E → E1 + T { E.postfix = E1.postfix T.postfix '+' }

E → T { E.postfix = T.postfix }

T → T1 \* F { T.postfix = T1.postfix F.postfix '\*' }

T → F { T.postfix = F.postfix }

F → (E) { F.postfix = E.postfix }

F → id { F.postfix = id.lexeme }

**Example: Translating a + b \* c**

**Step-by-step derivation:**

1. Expression: a + (b \* c)
2. Follow grammar:
   * E → E + T
   * E → T, T → F, F → a
   * T → T \* F, T → F, F → b, F → c

**Postfix generation:**

F → a → a

F → b → b

F → c → c

T → b \* c → b c \*

E → a + (b \* c) → a b c \* +

**Algorithm (Using Stack – Shunting Yard)**

This is another method (used in expression parsing):

* **Operands** are added directly to output.
* **Operators** are pushed to a stack.
* **Higher precedence** operators stay on top.
* **When encountering )**, pop and output until (.
* At the end, pop remaining operators to output.

**Example: Infix to Postfix (a + (b \* c - d) / e)**

1. Postfix: a b c \* d - e / +

Steps:

* Evaluate b \* c → then - d → then / e → finally + a

**Postfix for Code Generation**

Postfix is particularly suitable for generating stack-based intermediate code:

PUSH b

PUSH c

MUL

PUSH a

ADD

This can be derived directly from the postfix expression.

Postfix translation is a foundational compiler technique that:

* Simplifies parsing and evaluation.
* Avoids the complexity of precedence and associativity.
* Maps naturally to stack-based execution environments.

**12. Translation with a Top-Down Parser**

Top-down parsers begin from the start symbol and attempt to derive the input string by expanding productions **left to right**. The most common implementation is the **recursive descent parser**, where:

* Each **non-terminal** corresponds to a **function**
* Parsing decisions are made using **lookahead**
* Parsing can be directly integrated with **semantic actions** (translation tasks)

**Integration with Syntax-Directed Translation**

In a **syntax-directed translator**, each production rule is associated with:

* **Synthesized attributes** (computed from children)
* **Inherited attributes** (passed from parent/siblings)

Top-down parsers work **naturally** with **L-attributed grammars**, which allow:

* Passing inherited attributes as **function parameters**
* Computing synthesized attributes as **return values**

**How it Works in Recursive Descent Parsing**

* Each parsing function corresponds to a non-terminal.
* **Inherited attributes** are passed as parameters.
* **Synthesized attributes** are returned by the function.
* Semantic actions (like generating TAC, calculating expressions) are embedded at appropriate points.

**Example: Arithmetic Expression Evaluation**

**Grammar:**

E → T E'

E' → + T E' | ε

T → id

**Semantic Rules:**

* Pass inh (inherited value) down and return synthesized result.
* Compute values as parsing proceeds.

**Pseudo Code in Recursive Descent Style:**

def E():

val1 = T()

return E\_dash(val1)

def E\_dash(inh):

if lookahead == '+':

match('+')

val2 = T()

result = inh + val2

return E\_dash(result)

else:

return inh

def T():

if lookahead is id:

val = id.lexval

match(id)

return val

**Input: 3 + 4 + 5**

Evaluation trace:

* T → 3 → val1 = 3
* E' sees +, gets T → 4 → result = 3 + 4 = 7
* E' sees +, gets T → 5 → result = 7 + 5 = 12

**Final result: 12**

**Postfix Generation Instead of Evaluation**

You can also modify the semantic action to build postfix:

def E():

postfix1 = T()

return E\_dash(postfix1)

def E\_dash(inh\_postfix):

if lookahead == '+':

match('+')

postfix2 = T()

result = inh\_postfix + postfix2 + '+'

return E\_dash(result)

else:

return inh\_postfix

def T():

if lookahead is id:

tok = id.lexeme

match(id)

return tok

For a + b + c → Output: a b + c + (postfix)

**Advantages of Using Top-Down Parsing with Translation**

* **Immediate action**: Actions can be taken as soon as input is parsed.
* **No need for explicit parse trees**.
* **Clear code structure**: Each non-terminal as a function makes integration with semantic actions intuitive.
* **Supports L-attributed definitions** naturally.
* Useful for **interpreters** where evaluation occurs at parse time.

**Limitations**

* Can't handle **left-recursive grammars** directly.
* Parsing decisions must be deterministic and fit **LL(1)** form.
* Complex expressions might require more advanced attribute handling or preprocessing.

Top-down parsers, especially recursive descent parsers, are well-suited for integrating syntax-directed translations through:

* Passing inherited attributes as parameters
* Returning synthesized results
* Embedding code generation or evaluation logic inline

They are especially effective for **interpreters**, **expression evaluators**, and **simple compilers**.

**13. More about Translation**

This section explores how various advanced constructs are handled during intermediate code generation using TAC (Three Address Code), focusing on memory, control flow, and function semantics.

**a) Array References in Arithmetic Expressions**

Array access requires computing the **address of the element** being referenced. For a one-dimensional array:

* **Address formula**:  
  Address = base + index \* width

where:

* + base is the starting address of the array
  + index is the position being accessed
  + width is the size (in bytes) of each element (e.g., 4 for integers)

**Example:**

x = A[i] + B[j];

**TAC Translation:**

t1 = i \* 4 // assuming integer width = 4

t2 = A[t1] // access A[i]

t3 = j \* 4

t4 = B[t3] // access B[j]

x = t2 + t4

* Temporary variables are used to store addresses and intermediate results.
* This translation is critical for **pointer arithmetic and memory alignment**.

**b) Procedure Calls**

Translating a **function or procedure call** involves:

* **Evaluating and passing arguments**
* **Calling the procedure**
* **Capturing the return value**

**Example:**

result = sum(a, b);

**TAC Translation:**

param a

param b

call sum, 2 // function 'sum' takes 2 parameters

result = RET // return value captured

* param instructions push arguments to the call stack.
* call <name>, <arg\_count> initiates the function call.
* RET holds the returned value.
* Works well for both fixed and variable-parameter functions.

**c) Declarations**

**Variable declarations** like int x; or float y; are primarily for:

* **Allocating space**
* **Adding entries to the symbol table** with:
  + Variable name
  + Type
  + Size
  + Scope

**Example:**

int x;

float y;

**Translation:**

* No TAC is usually generated unless there is **initialization**.
* The **symbol table** is updated as:

x → int, size = 4 bytes

y → float, size = 4 bytes

**Example with initialization:**

int x = 5;

TAC:

ini

CopyEdit

x = 5

**d) Case Statements (switch)**

Switch-case is translated using either:

* **Sequential if checks**, or
* A **jump table** for faster indexing

**Example:**

switch(expr) {

case 1: x = 1; break;

case 2: x = 2; break;

default: x = 0;

}

**TAC Translation:**

t1 = expr

if t1 == 1 goto L1

if t1 == 2 goto L2

goto L3

L1: x = 1

goto Lend

L2: x = 2

goto Lend

L3: x = 0

Lend:

* Each case label is converted to a jump target (L1, L2, L3)
* goto Lend ensures only one block is executed
* This form is readable and compiler-friendly

**Summary Table**

| **Feature** | **Translation Strategy** |
| --- | --- |
| Array Access | Compute address using i \* width, dereference |
| Function Call | Use param, call, and assign RET |
| Declarations | Update symbol table; generate code if initialized |
| Case/Switch | Use if-chains or jump tables with labels |