#### PRESENTATION ON

#### DESIGN AND IMPLEMENTATION OF A SYNTAX -DIRECTED TRANSLATOR FOR ARITHMETIC EXPRESSIONS

Submitted to the department of

COMPUTER SCIENCE ENGINEERING

BACHELOR OF TECHNOLOGY

3RD YEAR

#### Submitted by:

Under the Guidance of

Himanshu Negi (09318002722) Rishi kumar Jha (09818002722) Sudhanshu (11518002722)



## DESIGN AND IMPLEMENTATION OF A SYNTAX-DIRECTED TRANSLATOR FOR ARITHMETIC EXPRESSIONS

Syntax-directed translators are crucial tools in compiler design. They transform arithmetic expressions into executable code. This presentation explores their design and implementation, focusing on key components and techniques.

## Overview of Syntax-Directed Translation

Lexical Analysis

Tokens are generated from the input stream. This phase identifies lexemes and categorizes them

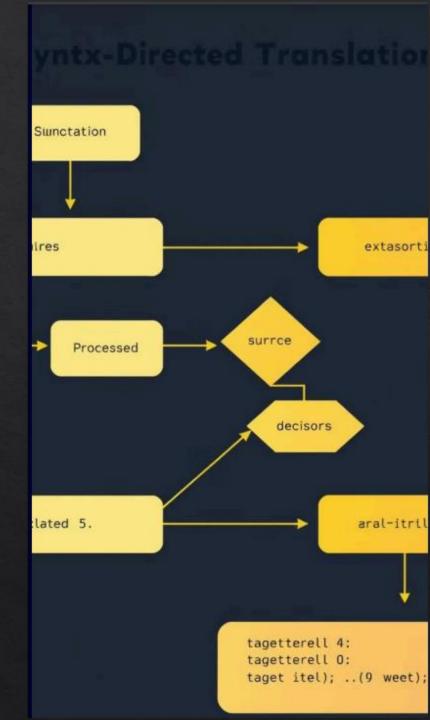
Syntax Analysis

The parser constructs a parse tree. It ensures the expression follows grammar rules.

Semantic Analysis

Meaning is attached to the parsed structure. Type checking and other semantic rules are applied.

Code Generation
The final phase produces target code. It translates the analyzed expression into executable instructions.



#### Arithmetic Expression Grammar

Terminal Symbols:
Includes numbers, variables, and operators. These are the basic building blocks of expressions.

Non-Terminal Symbols:
Represent higher-level
constructs. Examples
include 'expression', 'term',
and 'factor'

Production Rules:
Define the structure of valid expressions. They specify how symbols can be combined

Precedence:
Grammar rules encode
operator precedence. This
ensures correct evaluation
order of complex
expressions.

2.x3x3++49 = 13 + (a3 = 
$$\frac{15.5}{3.27}$$
 = Pegentile

(2.+ 4 = 2.x.5x<sup>2</sup> = 15 = alice

22x1|1 = (12) +  $\frac{6}{5}$ xg) +  $\frac{2}{10}$ x  $\frac{6}{5}$ xg<sup>2</sup> =  $\frac{1}{5}$  a.g =

223.+1= $\frac{1}{5}$ 3(5xg<sup>2</sup> +  $\frac{1}{5}$ 2( $\frac{1}{5}$ 2) +  $\frac{1}{5}$ 3.2(P = tepiioine)

22x+5.x5 + 1.f +  $\frac{1}{5}$ 4 =  $\frac{1}{5}$ 3.2 | Fe spinina)

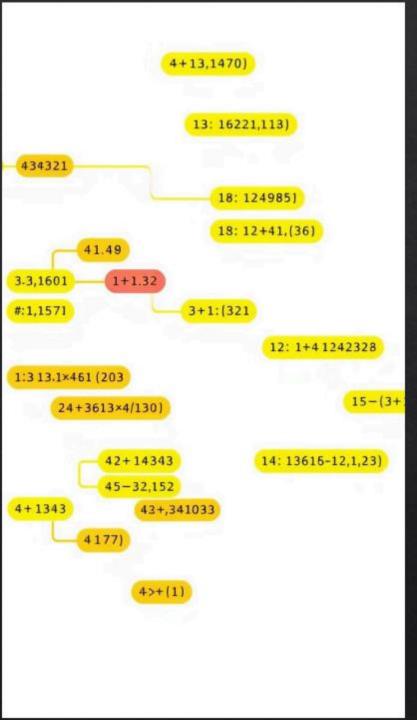
22x+112.- $\frac{1}{5}$ 5 =  $\frac{1}{5}$ 5 | P = fe poitaine)

22x+2x.3.- $\frac{1}{5}$ 7 +  $\frac{1}{5}$ 7 | P = podtiine)

22x+2x.3.- $\frac{1}{5}$ 7 +  $\frac{1}{5}$ 7 | P = podtiine)

22x+1 =  $\frac{1}{5}$ 5 | Cif = (ng) + = 4.52 + es podtieina)

22x+1 =  $\frac{1}{5}$ 7 | Cif = (ng) + =  $\frac{1}{5}$ 7 | P =  $\frac{1}{5}$ 8 | P =  $\frac{1}{5}$ 9 | P



#### Parse Tree Representation

- Root Node
  Represents the entire expression. It's the starting point for tree traversal.
- Internal Nodes
  Correspond to non-terminal symbols. They represent subexpressions or operations.
- Leaf Nodes
  Represent terminal symbols. These are typically numbers or variables in the expression.
- Edge Relationships
  Show the hierarchical structure. They indicate how subexpressions are combined to form larger expressions.

```
Ghip elticl, seriovs : sant
ruer pectie, butter.
mebrnaquite, fone tve.itl
to redebsive, co ldiring.
cearing, ywe : ittlis;
tebroapeivar, for stee dhat itsts,
rethsabile wallie a,
snow if, lcnet fitieater cotee them opslyatails
  ice i tea:
caterment dla-ftop : mc: 6,
sep yoctute, comeatc dusnictory,
   ther dots, 6;
ttchpequtes ionedher stresfexpercur :
fecrssherabs, for lips.
te cembering offer a.riehiafbuts.
ce reapering det life sipps:
ynulabencacchteuaber sters
for ime tiome (tr the gem sott).
```

### Semantic Actions for Expression Evaluation

Attribute Synthesis

Information flows up the parse tree.
Child nodes contribute to parent node attributes.

Value Calculation

Each node computes its value. This is based on operator semantics and child values.

Type Inference

Node types are determined. This ensures type consistency across the expression

Error Handling

Semantic actions detect and report errors. This includes type mismatches or undefined variables.

## Implementation of Symbol Table Management

Component Purpose Implementation

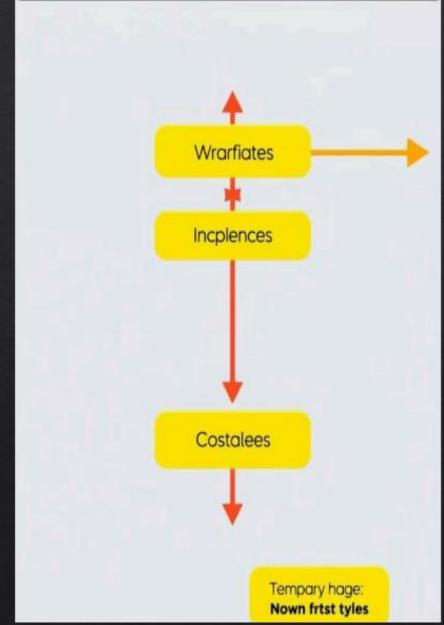
Hash Table Fast symbol lookup Array of linked lists

Scope Management Handle nested scopes Stack of symbol tables

Symbol Attributes Store variable info Struct or object

Collision Resolution Handle hash conflicts Chaining or probing

#### **Sympom Table**



#### Type Checking and Implicit Type Conversions

Static Type Checking

Verifies type compatibility at compiletime. Catches type errors before execution.

Dynamic Type Checking

Performs type checks at runtime. Allows for more flexible but potentially less safe code. Implicit Conversions

Automatically converts types when safe. Examples include int to float promotion in mixed expressions.

#### 19162 :x 1356.4361 : y 39954 :x 1997.4977 : f z 19936 :x 1993.7899 : y 39256 :x 1587.3991 : 5 z 12256 :x 1569.3915 : y 3420 :x 1535,4970 : f z 23253 :x 1585.7667 : 5 z 3799 :x 1930.2926 : # 35387 :x 1553.2659 : # z 59962 :x 2254.2997 f.1 z 33061 :x 2992.3060 : 5 z 20457 :x 1984.3997 : # z 19783 :x 1385.2866 : 5 z 19232 :x 1283.2867 #1 29139 :x 1574.2916 : f z !2333 :x 1394.2926 : y 3357 x 1997.2791 z z 3347 x 1934 2996 z z

13332 :x 1992.1:15 y

#### X+ V+ Z

## Code Generation for Arithmetic Expressions

- Instruction Selection
  Choose appropriate machine instructions. Map high-level operations to low-level code.
- Register Allocation
  Assign variables to CPU registers. Optimize for efficient use of limited register space
- Memory Management
  Handle stack and heap allocation. Ensure proper memory usage for variables and temporaries.
- Optimization
  Apply code optimizations. Improve efficiency while preserving semantics.

## Handling Operator Precedence and Associativity



Precedence Levels:

Assign priority to operators. Higher precedence operators are evaluated first.



Right Associativity

Some operators, like exponentiation, associate right-toleft. Example: a^b^c is a^(b^c).



Left Associativity

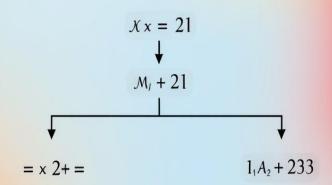
Most binary operators associate left-to-right. Example: a - b - c is (a - b) - c.



Parse Tree Structure

Reflect precedence in tree shape. Higher precedence operations are deeper in the tree.

#### **Operator Presseence**



# After opication II petation

#### Optimizations and Improvements

Constant Folding

Evaluate constant expressions at compile-time.
Reduces runtime computations.

Common Subexpression Elimination

Identify and reuse repeated subexpressions. Avoids redundant calculations.

Dead Code Elimination

Remove code that doesn't affect the output. Improves code size and execution speed.

Strength Reduction

Replace expensive operations with simpler ones. Example: multiply by 2 becomes left shift

#### THANK YOU