Author: Ahmed Waled Farouk Awad

ID: 120210287

Introduction:

This project is a complete (to-do) compiler for the `Tiny` programming language. `Tiny` is considered as subset of the `C` language contains:

- Data types: `int`, `float`, `string`, `bool`
- Variables declaration is the same as `C`

  ```
  int x;
  ```

- Conditions

  ```
  if condition
  then
  ```

- loops

  ```
  repeat ...
  until condition
  ```

- Functions

  ```
  int sum(int x, int y) {
      \\ some code
      return ;
  }
  ```

- Arithmetic, conditional, and boolean expressions

  ```
  x + y, a > b, x > 0 || y
  ```

- Read variables

  ```
  read x
  ```

- Write statement

  ```
  write some_expr
  ```

- Function call statement

  ```
  sum(x, y), print()
  ```

- Single Comment:

  ```
  \\ this is comment
  ```

- Multi line Comment

  ```
  \* this is
  mulit line
  comment *\
  ```

We will discuss what each phase mainly does.

Scanner Phase:

This is the phase where we define our tokens, i.e. reserved words, variables, literals (Integers, Real, strings).

Reserved Words in `Tiny`:

```
main, if, then, else, else, end , repeat, until, read, write, return, endl, int ,
float, string
```

These can be easily defined by setting a **unique token** for each word. For example

```
MAIN: 'main' ;
```

Other tokens exists as:

- Operators:
  - Arithmetic: `+, -, *, / %`
  - conditional: `>, <, >=, <=, =, !=`
  - Assigning: `:=`
  - Boolean: `&&, ||, !`

   - Other symbols: ☐ ; ' " \
- `ID` (variables names, function names): start with `letter` or `_` and followed by any number of `letter`, '_' or `digits`
- `Integer`: any number start with **non-zero digit**, followed by any number of `digits` or the number `0` itself.
- `Realnumber` (float): starts with `Integer` followed by '.' along with any number of digits
- `String`: starts and ends with `"` and contains any character (`String` can be nested by using `\"  \"`).
- `Letter`: any english character either lower or upper case `(a-zA-Z)`
- `Single-line comment`: start with `\\` followed by any character except `endline('\n') or carriage return ('\r')`
- `Multi-line commment`: start with `\*` and ends with `*\` and can contain any character (can't be nested)
- Whitespace: or `\n\t\r`

> `Whitespaces`, `single-line commnet`, `multi-line comment` are **skipped** during the scanner phase

The file `grammar\lexer\tinyLexer.g4` contains the regex and the tokes of all the mentioned above.

Parser Phase:

This is the phase where the **rules** of the language is written and gets created and visualized using the `parse tree`.

Rules for `Tiny`:

- A **program** consists of a number of functions (possibly zero) and a mandatory `main` function at the end.
   - The `main` function:
      - Always returns `int` and has no parameters.
      - Must have a **body** enclosed in `{}`:

        ```
        int main() {
            // body
        }
        ```

   - User-defined functions:
      - Have a **return type** (`int`, `float`, `bool`, `string`).
      - Have a function **name** (`ID`).
      - Have **optional parameters** enclosed in `()`.
      - Their body must be enclosed in `{}`.
      - Example:

        ```
        int sum(int x, int y) {
            return x + y;
        }
        ```

   - Every function (`main` or user-defined) must have a `body`.

Body:

A function body consists of **statements** (zero or more).

- **Read statement:**

  ```
  read ID;
  ```

- **Write statement:**

  ```
  write some_expr;
  ```

- **Declaration statement:**

  - Can declare variables with or without initialization.
  - Example:

    ```
    int x, y := 5;
    ```

- **Assignment statement:**

  ```
  ID := expr;
  ```

- **Function call statement:**

  - A function call consists of a function name followed by `(` and an optional **argument list**, ending with `);`.
  - Example:

    ```
    sum(x, y);
    ```

  - The argument list may include:
    - `ID`
    - `Literals`
    - `Expressions`
    - Another function call

      ```
      print(sum(2, 3));
      ```

- **If statement:**

```
if condition then
    // body
(any number of) elseif condition then
    // body
(must end with) else
    // body
end
```

- **Repeat statement:**
  - A `repeat` statement executes the **body** until a **condition** becomes true.

```
repeat
    // body
until condition
```

Expressions:

An **expression** can be:

- **Arithmetic expression**: `term` `arithmetic operator` `expression`

```
x + y, a * b, c / d, e % f
```

- **Conditional expression**: `term` `conditional operator` `expression`

```
x > y, a <= b, c != d
```

- **Boolean expression**: `term` `boolean operator` `expression`

```
x && y, a || b, !c
```

- **Function call**:

```
sum(x, y)
```

- **Term (single entity)**:

```
(expr), ID, literals
```

- **Literals**:

```
5, 3.14, "hello", true
```

AST Construction:

After generating the **parse tree**, it becomes necessary to convert it into a more meaningful and compact structure known as the **Abstract Syntax Tree (AST)**. The AST represents the **hierarchical structure** of the program while eliminating unnecessary syntactic information (e.g., parentheses, semicolons).

This process is done using the **Visitor Pattern**, where each grammar rule has a corresponding `visit` method in the `ASTVisitor.java` class. These methods return specific subclasses of `ASTNode` representing the actual tree structure.

Design of `ASTNode` and its subclasses:

Each component of the Tiny language has a corresponding node class:

- `ProgramNode`: stores a list of `FunctionNode`s and a `MainFunctionNode`.
- `FunctionNode`: stores function name, return type, parameters (`ParamListNode`), body (`BodyNode`), and return statement.
- `MainFunctionNode`: similar to `FunctionNode`, but always returns `int` and takes no parameters.
- `BodyNode`: holds a list of `statements`, each of which is also an `ASTNode`.
- `AssignmentNode`, `ReadNode`, `WriteNode`, `DeclarationNode`: represent the respective statements.
- `BinaryExprNode`, `UnaryExprNode`, `LiteralNode`, `IdentifierNode`: represent expressions and operands.
- `IfNode`, `ElseIfNode`, `RepeatNode`: represent control flow blocks.
- `FunctionCallNode` and `ArgListNode`: represent function invocation and arguments.

Example:

```
int sum(int a, int b) {
    return a + b;
}
```

Corresponding AST:

```
Program
  └── FunctionNode
        ├── Name: sum
        ├── Params: [int a, int b]
        ├── Return Type: int
        └── ReturnStatement
            └── BinaryExprNode('+')
                ├── Identifier: a
                └── Identifier: b
```

All AST Nodes implement the `ASTNode` interface. This allows us to easily traverse and manipulate the tree during semantic analysis and code generation.

`ASTPrinter.java`

After building the AST from a Tiny program using the `ASTVisitor`, the AST structure exists in memory but is not human-readable by default. The `ASTPrinter` solves this by:

- **Recursively visiting each node** in the AST
- Printing **meaningful labels** (e.g., `Function Name`, `Return`, `assign`, `op`) with proper **indentation**
- Providing a **hierarchical view** of the syntax tree that reflects the nesting and flow of the program

Main Methods

1. `print(ASTNode node)`

- Entry point for printing the AST.
- Calls the overloaded method `print(node, 0)` to start from indent level 0.

2. `print(ASTNode node, int indent)`

- This is a **recursive method** that inspects the type of each node and prints relevant information based on the node type.
- It uses `instanceof` checks to determine the specific subclass of the node.
- For each node type:
  - It prints a descriptive label
  - Then recursively calls `print()` on its children (like expressions, blocks, parameters, etc.)

3. `printIndent(int indent)`

- Utility method that prints indentation spaces.
- Used to create a **tree-like indentation** for better readability.
- Each level of depth adds 2 spaces.

Semantic Analysis Phase:

The **Semantic Analysis** phase ensures that the program is not only syntactically correct but also **semantically meaningful**.

This phase verifies:

- Correct variable and function declarations
- Proper type usage (e.g., arithmetic operations only between numeric types)
- Correct number and type of function arguments
- Variable scoping (variables declared within the right block)

The Symbol Table:

The compiler uses a `SymbolTable` to keep track of:

- Variable names and their types.
- Function names, their return types, and parameter lists.

Design:

```
private Deque<Map<String, Symbol>> variableScopes;  // for variables only
private Map<String, List<FunctionSymbol>> functions; // global function registry
```

> Since Tiny don't support multiple block definition as in C, we only care about the current scope. However, this isn't the case using conditional block like `if` or `repeat`, since we are allowing nesting ifs. Hence a `deque` is required to keep track of the scopes and fetch the nearest (lowest) scope ex
>
> ```
> if(x > 5){
>   int z;
>   if(z < 3){
>     int z; // different z
>   }
> }
> ```

Symbol Types:

We define two subclasses of `Symbol`:

- `VariableSymbol`: for tracking variables (`name`, `type`, `kind`)
- `FunctionSymbol`: for tracking function definitions, which includes:
  - `name`, `return type`
  - List of parameter types for overload resolution multiple functions names with different signature (return types, parameters)

  For example **this is valid in Tiny**

  ```
  int sum(int a, int b);
  int sum(int a, int b, int c); // added extra parameter
  float sum(int a, int b); // different return type
  ```

Expression Type Checking:

The `Analyzer.java` class recursively walks the AST and:

- **Resolves** variable and function declarations
- Checks type correctness for:
    - Binary operations
    - Unary operations
    - Function arguments and return types

Here's an enhanced version of the **Expression Type Checking** section with clear examples of both valid and invalid type conversions based on your compiler's semantic rules:

Valid Type Conversions:

Your Tiny language supports **implicit conversion** from `int` to `float` during arithmetic operations, allowing expressions like:

```
int x := 5;
float y := 2.5;
float z := x + y;   // Valid: int is promoted to float
```

Also, valid comparisons (relational expressions) between compatible types are allowed:

```
int a := 5;
float b := 7.1;
bool c := a < b;    // Valid: int and float comparison, allowed
```

Boolean expressions are allowed between boolean operands:

```
bool a := true;
bool b := false;
bool c := a || b;   // Valid: logical OR between two bools
```

Invalid Type Conversions:

Your semantic analyzer **disallows** operations between incompatible types. Some examples:

Mixing `string` with arithmetic:

```
string s := "hi";
int x := 3;
int y := s + x;     //  Error: Cannot perform '+' on 'string' and 'int'
```

Assigning `float` to `int` (no implicit cast down):

```
int x := 3.14;      //  Error: Cannot assign 'float' to 'int'
```

Using `bool` in arithmetic:

```
bool flag := true;
int result := flag + 5;   //  Error: Cannot add 'bool' to 'int'
```

Mismatched operand types in boolean logic:

```
int a := 3;
bool b := true;
bool c := a && b;    //  Error: Logical '&&' requires both operands to be 'bool'
```

Output in case of Semantic Error:

Here are some errors for the invalid semantic code provided below

```
int sum(int a, int b)
{
    return a + b;
}
int sum(int a){
    return 0;
}

int sum(){
    return "hello";
}


int main()
{
    int x;
    int x;

    float b := "sdfg";
    int a;
    a := 5 + true;

    string s;

    a := s;
    a := s + s;
    a := s + 5;

    sum(a, s);
}
```

The semantic analyzer catches this with the error:

```
Redeclaration of variable: x

Arithmetic operations are not supported on 'bool' types

Invalid operation: binary operator '+' cannot be applied to types 'string' and
'string'

Invalid operation: binary operator '+' cannot be applied to types 'string' and
'int'

no matching function named sum with types int, string

Available functions named sum
int, int
int
No args

Unmatched type between function sum and arguments provided
```

Three Address Code Generation (TAC):

The final phase is converting the AST into **TAC (Three Address Code)**, which is a low-level intermediate representation suitable for code generation or optimization.

TAC Format:

Each TAC instruction is a **quadruple**:

```
OP operand1 operand2 result
```

We generate TAC using the `CodeGen.java` class which:

- Traverses the AST
- Emits instructions for each node
- Manages temporary variables and labels

Temporary Variables and Labels:

- **Temporaries (`$T0`, `$T1`, ...)**: generated using `freshTemp()` for intermediate results.
- **Labels (`L0`, `L1`, ...)**: used for branching (`if`, `repeat`) with `freshLabel()`.

Example 1: Arithmetic Expression

```
x := a + b * c;
```

AST:

```
AssignmentNode
  └── BinaryExprNode('+')
        ├── Identifier: a
        └── BinaryExprNode('*')
              ├── Identifier: b
              └── Identifier: c
```

TAC:

```
MULT b c $T0
ADDI a $T0 $T1
STOREI $T1 x
```

Example 2: If-Else Statement

```
int z1, counter;
if  z1 > 5 || z1 < counter && z1 = 1 then
  write z1;
elseif z1 < 5 then
  z1 := 5;
else
  z1 := counter;
end
return 0;
```

TAC:

```
LABEL main
DECL z1
DECL counter
GT z1 5 $T0
LT z1 counter $T1
EQ z1 1 $T2
AND $T1 $T2 $T3
OR $T0 $T3 $T4
JUMPZ $T4 L0
WRITE z1
JUMP L1
LABEL L0
LT z1 5 $T5
JUMPZ $T5 L2
STORE 5 z1
JUMP L1
LABEL L2
STORE counter z1
LABEL L1
RET 0
```

Each `JUMPZ` instruction checks the condition. If false, it jumps to the `else` block. After the `then` block, an unconditional `JUMP` skips over the `else`.

Example 3: Repeat Loop

```
repeat
    x := x - 1;
until x = 0
```

TAC:

```
LABEL L0
SUBI x 1 $T0
STOREI $T0 x
EQ x 0 $T1
JUMPZ $T1 L0
```

The loop starts at `L0`, and jumps back unless the condition (`x = 0`) is true.

To run the project:

- Open the project as `inteliJ project`.
- Add the `antlr` jar file dependency from the `project Structure`.
- In the `src\Main.java`, state the test file to try on: `fileToParse`.

- Run the project, you can see the `parse tree` generated on the screen, and two additional files are produced in the `Testcases` directory:
  - The tokenizer file (with the same name as the test file): which has each token and the corresponding type.
  - The parse tree as nested parenthesis.