Assignment 7

Tue Apr 26 (125 points)

For this assignment, you will write code to translate a Calc parse tree into an intermediate representation, as defined below. Start with the `assn7` project directory in the `cs664pub` repository. It contains a working parser, type checker, and a substantial number of test programs annotated with their expected outputs.

The test programs are in the `src/test/resources/examples/run-expect` directory. For example, here is `011-add-many-ints.calc`:

```
print 3+5+8+13+21+37+58;  #expect 145
```

The jUnit test class in `src/test/java/TestRunExpect.java` loads each Calc test program, compiles it, ensures it type-checks, and then applies your `ConvertToIR` visitor. Once the IR is generated, it uses `InterpretIR` to run the compiled Calc program, and ensure its outputs match the documented expectations.

Here is a synopsis of the structure of `assn7`:

- `src/main/antlr` — the location of `CalcLang.g4`. Don't forget to run **Tools » Generate ANTLR Recognizer.** This time, we must configure the ANTLR tool to generate its code into the `calc.grammar` package:

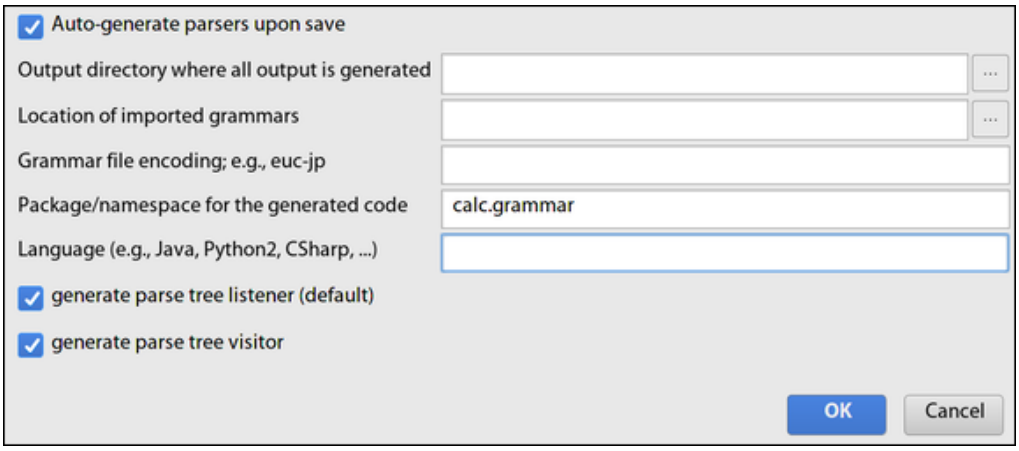

Figure 1: Set the package to `calc.grammar` in **Tools » Configure ANTLR...**

- `src/main/java` — the primary modules of the compiler, including:
  - `Type` — the enumeration representing all base types in Calc, as well as the `ERROR` designation.
  - `TypeChecker` and `TypeCheckingVisitor` — runs the lexer, parser, and type checker, and counts any errors that occur.

  - `Address` — represents operands in the IR, which may be constants, variables, or temporaries.
  - `Instruction` — represents a single IR instruction. The `Kind` enumeration defined within there denotes the instruction opcode. Instructions have a destination address and 0, 1, or 2 operand addresses.
  - `BasicBlock` — represents a sequence of straight-line IR instructions. For now it's just a thin interface around `ArrayList`, but as the language becomes more advanced in assignment 8, it will take on additional responsibilities.
  - `ConvertToIR` — **your work goes here.** This is a `CalcLangBaseVisitor` that traverses the parse tree and adds instructions to a `BasicBlock` object.
  - `InterpretIR` — Runs a basic block by interpreting each instruction. This is useful for validating translations in the absence of a back end to compile the IR to native code.
- `src/test/resources/examples` — sample programs in the Calc language:
  - `type-ok` — programs that should parse and type-check without errors
  - `type-err` — programs that should parse but contain type errors
  - `run-expect` — programs that should parse and type-check but also contain `#expect` annotations on each `print` statement.
- `src/test/java` — jUnit classes to run unit tests, including:
  - `TestTypeOk` — ensures the test programs in `type-ok` parse and type-check.
  - `TestTypeErr` — ensures the test programs in `type-err` parse but produce type errors.
  - `TestNodeTypes` — ensures the type-checker properly tracks the types of parse tree nodes and notes implicit coercions.
  - `TestRunExpect` — parses, type-checks, and translates test programs in `run-expect`, using `ConvertToIR`. Then it runs `InterpretIR` and compares the results to the documented expectations. **These will fail until you start making progress on ConvertToIR.**
  - `FindTestPrograms` — uses Java resource directories to locate the test programs.
  - `TestUtils` — other helper methods used by various test classes.

Compared to the implementation of the type checker in the assignment 5 solution, the type checker for this assignment remembers the type assigned to *every* expression node in the parse tree. So given a node `ctx` (context object) derived from `CalcLangParser.ExprContext`, we can call:

```
Type t1 = typeChecker.getNodeType(ctx);
```

to determine the type assigned to it by the type checker. This is helpful because the IR needs to know the type of every temporary address.

In addition, whenever the type checker uses leastUpperBound to allow an implicit conversion from INT to FLOAT, it remembers those too:

```
Type t2 = typeChecker.getCoercion(ctx);
```

If the returned type `t2` is non-null, it means that the expression represented by node `ctx` had to be promoted from type `t1` to type `t2`. Such coercions can happen in any operator, in a function argument, or in assigning to a variable.

Source language

The Calc language defined here supports only straight-line code; there are no branches, loops, or user-defined functions. (We will add some of these features in the next assignment.)

Values in the Calc language can be either strings, floats (double-precision), integers (64-bit), or booleans. Integers may be automatically promoted to floats, but those are the only implicit conversions.

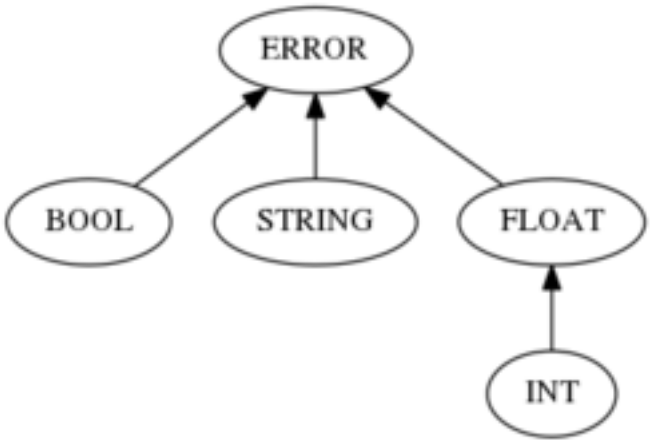

Figure 2: Type hierarchy for Calc language used in this assignment

The syntax is partitioned into statements (executed for effects such as output or assignment to a variable) and expressions (executed to produce a value). Below is the basic grammar; see `src/main/antlr/CalcLang.g4` for full details.

```
prog : (stmt ';')+
     ;
stmt : 'var' ID '=' expr                              #VarStmt
     | ID '=' expr                                    #AssignStmt
     | 'print' expr                                   #PrintStmt
     ;
expr : op=('-'|'!') expr                              #NegExpr
     | <assoc=right> left=expr op='^' right=expr      #OpExpr
     | left=expr op=('*'|'/'|'%') right=expr          #OpExpr
     | left=expr op=('+'|'-') right=expr              #OpExpr
     | left=expr op=('<'|'<='|'>'|'>=') right=expr    #OpExpr
     | left=expr op=('='|'<>') right=expr             #OpExpr
     | left=expr op='&' right=expr                    #OpExpr
```

```
    | left=expr op='|' right=expr                #OpExpr
    | ID '(' expr (',' expr)* ')'               #FunExpr
    | '(' expr ')'                              #ParenExpr
    | INT                                       #IntExpr
    | FLOAT                                     #FloatExpr
    | STR                                       #StringExpr
    | ID                                        #VarExpr
    | bool                                      #BoolExpr
    ;
```

There are two kinds of statements related to variables, `VarStmt` and `AssignStmt`. The `var` keyword introduces a new variable. Its type is determined from the expression which initializes it. Without the `var` keyword, we are assigning to an existing variable that must have already been declared.

```
1 var x = 6;     # x:int
2 x = 6.28;      # error, won't convert float to int
3
4 var y = 3.14;  # y:float
5 y = 3;         # okay because int<float
6
7 var y = 9;     # New var y:int shadows previous one
8 y = 9.87;      # error, won't convert float to int
```

Line 5 is an example of an automatic promotion from int to float. So if `ctx5e` refers to the `IntExprContext` object representing that constant `3` in the parse tree, we'd have:

- `Type.INT == typeChecker.getNodeType(ctx5e)`
- `Type.FLOAT == typeChecker.getCoercion(ctx5e)`

But in the constant `9` on line 7, there is no promotion. So if `ctx7e` refers to that `IntExprContext` object, we'd have:

- `Type.INT == typeChecker.getNodeType(ctx7e)`
- `null == typeChecker.getCoercion(ctx7e)`

Typing rules for the operators are mostly as before. We added the ^ exponent operator, which works on floats or ints (if the exponent is non-negative). The + operator is also expected to concatenate strings:

```
var name = "Chris";
print "Hello, " + name;  #expect Hello, Chris
```

Here are several built-in functions that are useful for creating sample programs:

- `floor` : (FLOAT) → INT
- `sqrt` : (FLOAT) → FLOAT
- `log` : (FLOAT, FLOAT) → FLOAT
- `parseInt` : (STRING) → INT
- `parseFloat` : (STRING) → FLOAT
- `showInt` : (INT) → STRING
- `showFloat` : (FLOAT) → STRING
- `readLine` : ( ) → STRING
- `random` : ( ) → FLOAT

And a sample program to calculate the volume of a sphere, given its radius:

```
var pi = 3.14159;
var ratio = 4.0/3;

print "Enter sphere radius:";
var radius = parseFloat(readLine());

var volume = ratio * pi * radius^3;
print "The sphere volume is " + showFloat(volume);
```

Intermediate representation

Our intermediate language supports the same set of types: STRING, BOOL, FLOAT, and INT, although we now treat booleans as being equal to the integers 0 (false) and 1 (true). We use that to implement logical AND in terms of integer multiplications: `0*1==0` for `false & true`. The boolean negation (NOT) can also be implemented in terms of integer subtraction: `1-x` for `!x`.

Otherwise, the IR doesn't support any implicit conversions from int to float. They must be made explicit using the `I2F` instruction.

Address

Addresses serve as the operands and destinations of instructions in the IR. They represent either variables (memory locations, `VAR`), constants (`CONST`), or temporaries (`TEMP`). They also keep track of the type of data they contain.

```
public class Address {
    enum Kind {
        VAR, CONST, TEMP
    }
```

```
    Kind kind;
    Type type;
    String value;  // Data value for CONST; variable name for VAR; null otherwise
    int serialNum; // Only used for TEMP

    Address(Type type)  // Create a new temporary of given type
    {…}
    Address(Kind kind, Type type, String value) // Constructor for CONST, VAR
    {…}
}
```

There are two constructors. One creates a fresh, new temporary of the given type. The other is used to construct addresses representing constants or variables.

Instruction

An instruction performs some action, specified by its `kind`. It has a destination address and zero, one, or two operand addresses, depending on the kind of instruction being represented.

```
public class Instruction {
    enum Kind {…}         // Listed below
    Kind kind;
    Address destination;  // Can be null
    ArrayList<Address> operands = new ArrayList<Address>();
    // Constructors for zero, one, or two operands
    Instruction(Kind kind, Address destination) {…}
    Instruction(Kind kind, Address destination, Address source) {…}
    Instruction(Kind kind, Address destination, Address left, Address right) {…}
}
```

The destination address **must** be a temporary, except in a few cases: for a `PRINT` instruction, it should be null. For a `STORE` instruction, the destination address should be a variable.

Operand addresses can be either variables, temporaries, or constants. The kind of instruction determines the number of operands needed and what types are acceptable.

Here are brief descriptions of all the kinds of instructions:

- `ADD` — takes two operands of the same numeric type, adds them.
- `DIV` — takes two operands of the same numeric type, divides them. For integers, this means integer division (the result is also an integer).
- `EQ` — takes two operands of the same type, compares them for equality, returns a boolean.

- F2S — takes one float operand, converts it to a string.
- FLOOR — takes one float operand, converts it to an int by rounding down.
- I2F — takes one integer operand, converts it to a float.
- I2S — takes one int operand, converts it to a string.
- LE — takes two operands of the same numeric type, returns boolean if first is <= second.
- LOG — takes two float operands, calculates log with base.
- LT — takes two operands of same numeric type, returns boolean if first is strictly < second.
- MOD — takes two integer operands, returns modulus.
- MUL — takes two operands of the same numeric type, multiplies them. Also can be used for logical and of boolean values.
- NE — inverse of EQ
- OR — takes two boolean values and performs logical or.
- POW — takes two operands of the same numeric type, calculates the first to the power of the second. If operands are integers, second operand must be non-negative.
- PRINT — takes operand of any type, prints its result to the output stream.
- RANDOM — takes no operands, returns a random float >= 0 and < 1.
- READLINE — takes no operands, reads one line from input stream and returns it as a string with newline character removed.
- S2F — takes one string operand and attempts to convert it to a float. It's a run-time error if it cannot be converted.
- S2I — takes one string operand and attempts to convert it to an int. It's a run-time error if it cannot be converted.
- SCONCAT — takes two string operands and returns a new string in which the two operands are concatenated.
- SQRT — takes one float operand and produces its square root as a float.
- STORE — takes one operand of any type. Copies it into the destination operand, which is permitted to be a variable address.
- SUB — takes two operands of the same numeric type, subtracts them.

BasicBlock

A basic block is just a sequence of straight-line IR instructions, stored in an `ArrayList` collection. The `BasicBlock` class has an `add()` method to append a new instruction at the end of the block. Here is an example where we build up an IR program directly using Java code, and then interpret it.

```
Address a1 = new Address(Type.FLOAT);
Address a2 = new Address(Type.FLOAT);
Address a3 = new Address(Type.STRING);
Address a4 = new Address(Type.STRING);

Address pi   = new Address(Address.Kind.CONST, Type.FLOAT,  "3.14159");
```

```
Address two  = new Address(Address.Kind.CONST, Type.INT,    "2");
Address mesg = new Address(Address.Kind.CONST, Type.STRING, "PI^2 is ");

BasicBlock block = new BasicBlock();
block.add(new Instruction(Kind.I2F, a1, two));
block.add(new Instruction(Kind.POW, a2, pi, a1));
block.add(new Instruction(Kind.F2S, a3, a2));
block.add(new Instruction(Kind.SCONCAT, a4, mesg, a3));
block.add(new Instruction(Kind.PRINT, null, a4));

System.out.println(block);

InterpretIR interpreter = new InterpretIR();
ArrayList<String> output = new ArrayList<>();
interpreter.run(block, output);
System.out.println(output);
```

And the output of that program is:

```
FLOAT t1 = I2F(INT 2)
FLOAT t2 = POW(FLOAT 3.14159, FLOAT t1)
STRING t3 = F2S(FLOAT t2)
STRING t4 = SCONCAT(STRING PI^2 is , STRING t3)
PRINT(STRING t4)

[PI^2 is 9.869588e+00]
```

For the sake of consistency, our programs will print all floating-point values in scientific notation with six digits after the decimal point. This corresponds to `"%.6e"` in Java's `String.format()` method, or `"%.6le"` with C's `printf`.