TREE WALKING AND CODE GENERATION

Introduction

This lab session provides a framework for type checking and generating code from an AST. These techniques can be used to generate a linear IR or potentially to generate TAM code directly.

This lab is a bit more guided than previous ones, in response to feedback. You are not required to implement your compiler following the suggestions in this lab, but I am providing some more concrete suggestions for how to implements these compiler features. Feel free to deviate from the suggestions here however you like.

Note: This lab sheet is not necessarily intended to all be completed in the two hours. This is a guide to the work that needs to be completed to implement the features we have discussed over the last week or so.

Setup

There is a zip file on Moodle containing some suggested data structures for use in type checking and linear IR generation. You are free to incorporate as much or as little of this code into your compiler as you like. If and when you copy the code across, pay attention to the package names and make sure you copy them into the right places.

Note: I make no claims about my code being the *best* way of doing these things, I just want to suggest *a* way of doing them.

Tree visitors

The visitor pattern is an excellent approach to translating an AST into a linear format. We cannot easily implement a 'translate' method on our AST nodes directly because translation requires context from outside the node itself. Employing the visitor pattern allows us to traverse an AST and carry out arbitrary computations.

Sample code

Recall the simple AST hierarchy we designed for the Lisp language:

```
sealed interface Ast permits Atom, ExprList {}
sealed interface Atom extends Ast permits Number, Symbol {}
record Number(int value) implements Atom {}
record Symbol(String value) implements Atom {}
record ExprList(List<Ast> elems) implements Ast {}
```

Implementing a visitor for this structure simply involves defining a 'visit' method for each discrete type of AST node, perhaps with a general 'visit' method that takes an

arbitrary node and defers to the appropriate specific method. An implementation of this top-level function might look like this:

```
T visit(Ast ast) {
    return switch (ast) {
        case Number num -> visitNumber(num);
        case Symbol sym -> visitSymbol(sym);
        case ExprList list -> visitList(list);
    };
}
```

We assume that our visitor class returns a generic type in the same way as ANTLR's parse tree visitor.

Task: Design an `AstVisitor<T>` interface for your code. This interface should resemble the `NottranVisitor<T>` interface that ANTLR generated, except that it should have a visitor method for every AST node, as opposed to a method for every nonterminal.

Type checking

A naïve method of type checking is to walk an AST. The type of an expression is determined by the types of its subexpressions, so we can naturally check an expression's type by checking the type of its subexpressions and then applying an appropriate inference rule. Statements don't have types, but they can be checked for correctness by ensuring that any subexpressions are of the correct types (e.g. ensuring that the condition in an if-statement is a logical statement).

Task: Define a hierarchy of classes to represent the possible types in Nottran. This may look like the hierarchy of AST classes. I might design something like this:

```
sealed interface Type permits BaseType, ArrayType {}
enum BaseType implements Type { Number, Logical, etc. }
record ArrayType(Type base, List<Integer> dims) implements Type {}
```

Task: Modify the data structure you use to track symbol data to also store the symbol's type.

Task: Design a `TypeChecker` class that implements your `AstVisitor<T>` interface. Implement each of the visit methods that correspond to expressions to compute the type of the expression and return it. When checking statements, the actual value you return doesn't matter, you simply need to check that the statement's subexpressions are correct and throw an exception if not.

Code generation

Code generation can also be carried out using a tree walk. The code required to represent a single statement is determined by the kind of statement and by the substatements or subexpressions it includes. As such, a simple approach to code

generation is to talk the AST again and use visitor methods to generate code for a single node.

I recommend you lower your AST to a linear IR first, before trying to translate directly to TAM code. The example code online has a simple set of data structures for representing three-address code that you can use.

Task: Design a `TacGenerator` class that implements `AstVisitor<T>`. Each visit method should return a list of TAC instructions that represent the provided AST node.

We represent labels by assigning each TAC instruction a unique ID and referencing these IDs for jump targets. When translating to TAM code these labels will then be translated into actual code addresses.

As a hypothetical example for a simple if-statement, this might be what the method could look like:

```
List<TacInstr> visitIfStmt(IfStmt stmt) {
    List<TacInstr> condCode = visitExpression(stmt.cond());
    List<TacInstr> action = visitStatement(stmt.action());
    TacInstr noop = new TacInstr(TacOp.Noop, 0, 0, 0);

    List<TacInstr> code = new ArrayList<>();
    code.addAll(condCode);
    code.add(
        new TacInstr(TacOp.JumpIf, noop.id(), code.getLast().dst());
    code.addAll(action);
    code.add(noop);
    return code;
}
```

This method generates code for the Boolean test and the body of the statement. An imaginary ‘no-op’ instruction is used to label the instruction after the body so that we can jump over it if needed. We then create an instruction to jump to the no-op instruction if the logical test fails.

All these instructions are then added to a single list, and the list is returned.