CS4201 Assignment 2: Intermediate Representations

Due date: Friday 17th November 2023, 9pm

50% of the practical grade for the module

You are expected to have read and understood all the information in this specification and any accompanying documents at least a week before the deadline. You must contact the lecturer regarding any queries well in advance of the deadline.

Aim

In this practical you will implement part of a compiler/translator for a small expression language, by converting to increasingly simple intermediate representations, then finally generating code in a target language (Java). The purpose is to:

- give you experience in constructing and manipulating abstract syntax trees.
- reinforce your understanding of defunctionalisation and administrative normal form (ANF).
- demonstrate the purposes of different intermediate representations in a compiler.
- demonstrate what is required to translate a source program to an executable program in an existing target language.

You are given starter code (in Haskell) but you are not required to use it. If you prefer, you may use Java as an implementation language. However, I strongly recommend using Haskell because you will be doing a lot of analysis of syntax trees, for which algebraic data types and pattern matching are highly suited.

If you feel any part of the specification is ambiguous, you may interpret it in any way you feel is appropriate, and explain your decisions in your report.

Preparation

Make sure you are familiar with the lecture material from weeks 7 and 8, covering definitional interpreters, defunctionalisation, and administrative normal form (ANF).

Requirements

Your task is to take a small intermediate expression language, of the form which might be produced by the first few stages of a compiler, and transform it, via ANF, into an executable Java program. A program in this language (let's call it "Defun") consists of top level functions, and a main expression, which should evaluate to an integer. Therefore, the resulting Java program's main method should compute and print that integer.

There are three steps:

1. *Defunctionalise* the program, so that all function calls are to defined top level functions, with the correct number of arguments.
2. Convert the defunctionalised program to ANF, so that the operands to all function calls, operators, and scrutinees of `case` and `if` expressions are variable names.
3. Generate Java code from the ANF. There should be one method for each function in the ANF (which will be the original functions, plus the `EVAL` and `APPLY` functions generated by defunctionalisation), and a main method which evaluates the main expression.

The expression language consists of the following forms (with some examples):

- Variables, which you can represent as strings
- Integer values
- Let binding (e.g. `let x = 94 in x + x`)
  - Assigns `94` to `x` then evaluates the scope of the binding `x + x`
- Function calls (e.g. `double(2)` and `f(1, 2, 3)`)
- Binary operators `+`, `-`, `*`, `/`, `==`, `<`, `>`, all of which operate on integers and return integers (where 0 means `True` and nonzero means `False`)
- If expressions (e.g. `if x == y then 1 else 2`)
  - Evaluates the expression `x==y`, if it is nonzero evaluates to `1` otherwise evaluates to `2`
- Data constructors (e.g. `Nil()`, `Cons(x, xs)`, `Inl(2)`, `Inr(3)`)
  - Constructs a value applied to some arguments (like a Haskell data constructor). Constructors are always fully applied.
- Case blocks (e.g. `case exp of { Nil() -> empty(); Cons(x,xs) -> nonempty(x, xs) }`)
  - Evaluates the expression `exp`. If it matches `Nil`, evaluates `empty()` otherwise extracts `x` and `xs` from the constructor and passes them to the function `nonempty`.

Functions consist of a name, an argument list, and the body, which is an expression, i.e. `f(arg1, arg2, ...., argn) = body`

This language can be formally represented using the following Haskell data types, as given in the starter code in `Defun.hs`:

```
data Op = Plus | Minus | Times | Divide | Eq | Lt | Gt

data Expr
     = Var Name
     | Val Integer
     | Let Name Expr Expr
     | Call Expr [Expr]
```

```
      | Con Name [Expr]
      | If Expr Expr Expr
      | BinOp Op Expr Expr
      | Case Expr [CaseAlt]

data CaseAlt = IfCon Name [Name] Expr

data Function = MkFun Name -- function name
                      [Name] -- argument names
                      Expr -- function body

data Program = MkProg [Function] -- all the function definitions
                      Expr -- expression to evaluate
```

Correspondingly, in Java, `Expr` could be represented as an abstract base type or interface, with instances for each of the constructors `Var`, `Val`, `Let`, etc.

Your job, therefore, is to first defunctionalise every `Expr` in a `Program` so that all instances of `Call` apply a top level function to the exact number of arguments that function expects (this will involve generating `EVAL` and `APPLY` functions for a data type representing partially applied functions, as discussed in lectures); then, transform the program to ANF so that every function argument and every scrutinee of an `if` and `case` expression (that is, the value which is being tested) is a variable. Once every function body and expression is in ANF, it has a direct translation to Java code.

The starter code consists of the above definition of `Expr` in a file `Defun.hs` with several example programs and a file `ANF.hs` which includes a suggested representation of expressions in ANF, an outline of the top level functions you will need to write, and a `main` program which, if successful, will print out Java code that evaluates the expression and prints the result.

You are not required to use the starter code, and you may implement your program in either Haskell or Java. Note also that there is no need to write a parser, it is fine to enter examples in the syntax tree directly. (You may nevertheless find it helpful while debugging to write a pretty printer for the syntax tree, to confirm your inputs are what you expect!)

Examples

The following examples, plus several others, are included in `Defun.hs`.

A function to double an integer would be defined as follows:

```
double(val) = val * 2
```

This would be represented in the syntax tree as

```
MkFun "double" ["val"]
        (BinOp Times (Var "val") (Val 2))
```

A complete program using this might be

```
double(val) = val * 2
main = double(3)
```

The complete program would be represented in the syntax tree as

```
MkProg allDefs (Call (Var "double") [Val 3])
```

(Assuming that `allDefs` is a list of function definitions which includes the above representation of `double`)

Running your program should generate (at least) a Java method `double` which takes an `int` and returns an `int`, and a `main` method which evaluates `double(3)` and prints the result.

A function to sum all the elements of a list would be

```
sum(xs) = case xs of
               Nil -> 0
               Cons(y, ys) -> y + sum(ys)
```

This would be represented in the syntax tree as

```
MkFun "sum" ["xs"]
    (Case (Var "xs")
          [IfCon "Nil" [] (Val 0),
           IfCon "Cons" ["y", "ys"]
                 (BinOp Plus (Var "y") (Call (Var "sum") [Var "ys"]))])
```

This assumes that an empty list is represented as a constructor `Nil` applied to zero arguments, and a non-empty list with a head and a tail is represented as a constructor `Cons` applied to two arguments, the head and the tail. For example, a function to produce a list of `1` and `2` could be:

```
testlist() = Cons(1, (Cons(2, Nil())))
```

which would be represented in the syntax tree as

```
MkFun "testlist" []
    (Con "Cons" [Val 1, Con "Cons" [Val 2, Con "Nil" []]])
```

Deliverables

Submit a zip file containing your code and report, in separate subdirectories, to the P2 slot on MMS. Note that your submission *must* include a report, which must describe the steps you took to implement the requirements. The report should include:

- a brief introduction, summarising what you achieved.
- a description of your design and implementation choices and descriptions of where you have resolved any ambiguities you noticed in the specification.

- sample outputs of your programs, demonstrating how you tested your implementation.
- an evaluation/conclusion, reflecting on what you have achieved, what you have learned, and any changes you would consider given what you have learned.

Also, please note that:

- You must include instructions for how to run and test your program, in a `README.txt` file in the root directory of your submission. This should assume that I am running your program on a lab machine. A single line giving the command for me to run is fine.

Hints and simplifying assumptions

- You can assume that all variable names are unique, and all programs have been type checked and are type correct, as these steps would have been performed earlier in a full-scale compiler.
- You can assume that the `main` expression will always evaluate to an integer (but you may want to extend it to be able to display constructor forms too).
- Start simple: some of the example programs can be translated directly to ANF without a defunctionalisation step (notably `double` and `testProg1`). Make sure these work before moving on.
- Make sure `if` and binary operators work before moving on to `case` and constructors.
- The resulting Java does not have to look pretty. It just has to run!
- In the generated Java, you will need a uniform representation of values. For simplicity, I suggest an `Object` which can be either an instance of `Integer` or a class that represents constructors applied to arguments.
- A constructor form can be represented in Java as a `String` and a list of arguments.
- The Java will mostly be a direct mapping from the ANF representation. One difficulty, however, is with `case` expressions. These can be translated to `switch` blocks which inspect the constructor "tag" then extract its arguments into local variables.
- The function which generates Java will be easier to write using a helper function which takes two arguments: the ANF to be translated; and the variable to store the result in.

So, for a function `fun` defined in ANF:

```
fun(x) = let y = case x of
                   Nil() -> 0
                   Cons(z, zs) -> z + z
             in y + y
```

The resulting Java might be (something like)

```
Object fun(Object x) {
    Object z, zs, y, res; // intermediate variables
    switch(x) { // Compiling the case block, putting the result in `y`
    case "Nil":
        y = Integer(0);
        break;
    case "Cons":
        z = getArg(x, 0);
        zs = getArg(x, 1);
        y = z + z;
        break;
    }
    res = y + y;
    return res;
}
```

Here we translate the top level expression to a Java program which assigns the result to `res` then returns it. When we compile the definition in the `let`, we know that the result needs to be assigned to the variable `y`, then when we compile the body, we know that `y` has been bound.

Marking

This practical will be marked according to the guidelines at https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html.

It is possible to achieve a grade of 7 by implementing only one of the three steps, or by only implementing part of the language end-to-end, with a report that explains what was achieved including test cases. To achieve a grade of 17 or higher, you will need to implement all three steps, with a report that clearly explains what you have achieved including testing and evaluation.

Also note that:

- Standard lateness penalties apply as outlined in the student handbook at https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html
- Guidelines for good academic practice are outlined in the student handbook at https://info.cs.st-andrews.ac.uk/student-handbook/academic/gap.html