Coursework 5[1]

This coursework is worth 12% and is due on 24 January at 18:00. You are asked to implement a compiler targeting the LLVM-IR. Be careful that this CW needs some material about the LLVM-IR that has not been shown in the lectures and your own experiments might be required. You can find information about the LLVM-IR at

- `https://bit.ly/3rheZYr`
- `https://llvm.org/docs/LangRef.html`

You can do the implementation of your compiler in any programming language you like, but you need to submit the source code with which you generated the LLVM-IR files, otherwise a mark of 0% will be awarded. You should use the lexer and parser from the previous courseworks, but you need to make some modifications to them for the 'typed' fun-language. I will award up to 4% if a lexer and parser are implemented. At the end, please package everything(!) in a zip-file that creates a directory with the name `YournameYourFamilyname` on my end.

Disclaimer

It should be understood that the work you submit represents your own effort. You have not copied from anyone else. An exception is the Scala code I showed during the lectures or uploaded to KEATS, which you can both use. You can also use your own code from the CW 1 – CW 4.

Task

The goal is to lex and parse the Mandelbrot program shown in Figure 2 and generate corresponding code for the LLVM-IR. Unfortunately the calculations for the Mandelbrot set require floating point arithmetic and therefore we cannot be as simple-minded about types as we have been so far (remember the LLVM-IR is a fully-typed language and needs to know the exact types of each expression). The idea is to deal appropriately with three types, namely `Int`, `Double` and `Void` (they are represented in the LLVM-IR as `i32`, `double` and `void`). You need to extend the lexer and parser accordingly in order to deal with type annotations. The Fun-language includes global constants, such as

```
val Ymin: Double = -1.3;
val Maxiters: Int = 1000;
```

where you want to assume that they are 'normal' identifiers, just starting with a capital letter—all other identifiers should have lower-case letters. Function

[1] September 4, 2021

definitions can take arguments of type `Int` or `Double`, and need to specify a return type, which can be `Void`, for example

```
def foo(n: Int, x: Double) : Double = ...
def bar() : Void = ...
```

The idea is to record all typing information that is given in the program, but then delay any further typing inference to after the CPS-translation. That means the parser should generate ASTs given by the Scala dataypes:

```
abstract class Exp
abstract class BExp
abstract class Decl

case class Def(name: String, args: List[(String, String)],
               ty: String, body: Exp) extends Decl
case class Main(e: Exp) extends Decl
case class Const(name: String, v: Int) extends Decl
case class FConst(name: String, x: Float) extends Decl

case class Call(name: String, args: List[Exp]) extends Exp
case class If(a: BExp, e1: Exp, e2: Exp) extends Exp
case class Var(s: String) extends Exp
case class Num(i: Int) extends Exp    // integer numbers
case class FNum(i: Float) extends Exp // floating numbers
case class Aop(o: String, a1: Exp, a2: Exp) extends Exp
case class Sequence(e1: Exp, e2: Exp) extends Exp
case class Bop(o: String, a1: Exp, a2: Exp) extends BExp
```

This datatype distinguishes whether the global constant is an integer constant or floating constant. Also a function definition needs to record the return type of the function, namely the argument `ty` in `Def`, and the arguments consist of an pairs of identifier names and types (`Int` or `Double`). The hard part of the CW is to design the K-intermediate language and infer all necessary types in order to generate LLVM-IR code. You can check your LLVM-IR code by running it with the interpreter `lli`.

LLVM-IR

There are some subtleties in the LLVM-IR you need to be aware of:

- **Global constants**: While global constants such as

  ```
  val Max : Int = 10;
  ```

  can be easily defined in the LLVM-IR as follows

  ```
  @Max = global i32 10
  ```

they cannot easily be referenced. If you want to use this constant then you need to generate code such as

```
%tmp_22 = load i32, i32* @Max
```

first, which treats `@Max` as an Integer-pointer (type `i32*`) that needs to be loaded into a local variable, here `%tmp_22`.

- **Void-Functions**: While integer and double functions can easily be called and their results can be allocated to a temporary variable:

```
   %tmp_23 = call i32 @sqr (i32 %n)
```

  void-functions cannot be allocated to a variable. They need to be called just as

```
  call void @print_int (i32 %tmp_23)
```

- **Floating-Point Operations**: While integer operations are specified in the LLVM-IR as

```
  def compile_op(op: String) = op match {
    case "+" => "add i32 "
    case "*" => "mul i32 "
    case "-" => "sub i32 "
    case "==" => "icmp eq i32 "
    case "<=" => "icmp sle i32 " // signed less or equal
    case "<"  => "icmp slt i32 " // signed less than
  }
```

  the corresponding operations on doubles are

```
  def compile_dop(op: String) = op match {
    case "+" => "fadd double "
    case "*" => "fmul double "
    case "-" => "fsub double "
    case "==" => "fcmp oeq double "
    case "<=" => "fcmp ole double "
    case "<"  => "fcmp olt double "
  }
```

- **Typing**: In order to leave the CPS-translations as is, it makes sense to defer the full type-inference to the K-intermediate-language. For this it is good to define the `KVar` constructor as

```
case class KVar(s: String, ty: Ty = "UNDEF") extends KVal
```

where first a default type, for example `UNDEF`, is given. Then you need to define two typing functions

```
def typ_val(v: KVal, ts: TyEnv) = ???
def typ_exp(a: KExp, ts: TyEnv) = ???
```

Both functions require a typing-environment that updates the information about what type each variable, operation and so on receives. Once the types are inferred, the LLVM-IR code can be generated. Since we are dealing only with simple first-order functions, nothing on the scale as the 'Hindley-Milner' typing-algorithm is needed. I suggest to just look at what data is avaliable and generate all missing information by simple means.

- **Build-In Functions**: The 'prelude' comes with several build-in functions: `new_line()`, `skip`, `print_int(n)`, `print_space()` and `print_star()`. You can find the 'prelude' for example in the file `sqr.ll`.

```
// Mandelbrot program

val Ymin: Double = -1.3;
val Ymax: Double =  1.3;
val Ystep: Double = 0.05;  //0.025;

val Xmin: Double = -2.1;
val Xmax: Double =  1.1;
val Xstep: Double = 0.02;  //0.01;

val Maxiters: Int = 1000;

def m_iter(m: Int, x: Double, y: Double,
                   zr: Double, zi: Double) : Void = {
  if Maxiters <= m
  then print_star()
  else {
    if 4.0 <= zi*zi+zr*zr then print_space()
    else m_iter(m + 1, x, y, x+zr*zr-zi*zi, 2.0*zr*zi+y)
  }
};

def x_iter(x: Double, y: Double) : Void = {
  if x <= Xmax
  then { m_iter(0, x, y, 0.0, 0.0) ; x_iter(x + Xstep, y) }
  else skip()
};

def y_iter(y: Double) : Void = {
  if y <= Ymax
  then { x_iter(Xmin, y) ; new_line() ; y_iter(y + Ystep) }
  else skip()
};


y_iter(Ymin)
```

Figure 1: The Mandelbrot program in the ‘typed’ Fun-language.

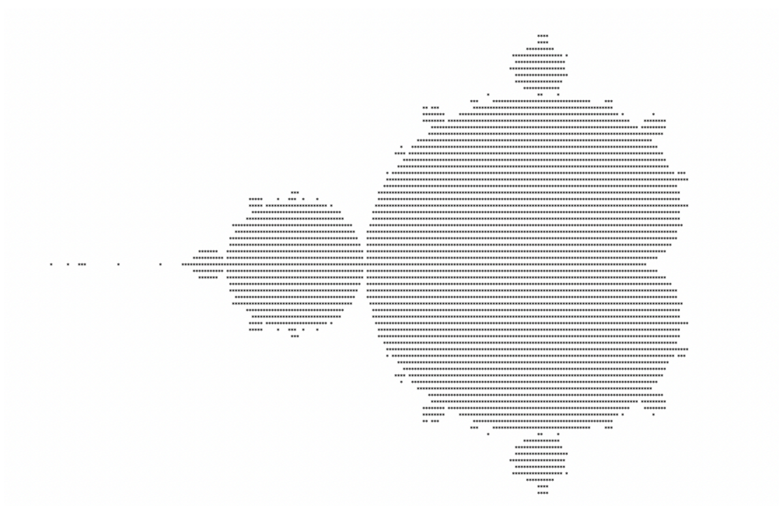

Figure 2: Ascii output of the Mandelbrot program.