# Compiler Extensions for Amy

Computer Language Processing

LARA

Autumn 2022

## 1 Introduction

In this document you will find some compiler extension ideas for the last assignment of the semester. The ideas are grouped in sections based on the broader subject they cover. Every extension has an integer indicating its (estimated) difficulty.

## 2 Your own idea!

We will be very happy to discuss an idea you come up with yourselves.

## 3 Language features

Projects in this section extend Amy by adding a new language feature. To implement one of these projects, you will probably need to modify every stage of the compiler, from lexer to code generation. If the project is too hard, you might be allowed to skip the code generation part and only implement the runtime of your project in the interpreter.

### 3.1 Imperative features (2+)

With the exception of input/output, Amy is a purely functional language: none of its expressions allow side effects. Your task for this project is to add imperative language features to Amy. These should include:

- Mutable local variables.

```
var i: Int;
var j: Int = 0;
i = j;
```

```
    j = i + 1;
    Std.printInt(i);
    Std.printInt(j) // prints 0, 1
```

Make sure your name analysis disallows plain **val**s to be mutated.

- While loops.

```
fn fact(n: Int): Int = {
  var res: Int = 1;
  var j: Int = n;
  while(1 < j) {
    res = res * j;
    j = j - 1
  };
  res
}
```

- *Bonus:* Arrays. You should support at least array initialization, indexing and extracting array length.

## 3.2  Implicit parameters (1)

Much like Scala, this feature allows functions to take implicit parameters.

```
fn foo(i: Int)(implicit b: Boolean): Int = {
  if (i <= 0 && !b) { i }
  else { foo(i - 1) + i } // good, implicit parameter in scope
}
foo(1)(true); // good, argument explicitly provided
foo(1); // bad, no implicit in scope
implicit val b: Boolean = true;
foo(1); // good, implicit in scope
        // equivalent to foo(1)(b)
implicit val b2: Boolean = false;
foo(1) // Bad, two boolean implicits in scope.
```

When a function that takes an implicit parameter is called and the implicit parameter is not explicitly defined, the compiler will look at the scope of the call for an implicit variable/parameter definition of the same type. If exactly one such definition is found, the compiler will complete the call with the defined variable/parameter. If more than one or no such definitions are found, the compiler will fail the program with "implicit parameter conflict" or "no implicit found" errors respectively.

## 3.3  Implicit conversions (1)

Much like Scala, this feature allows specified functions to act as implicit conversions.

```
implicit fn i2b(i: Int): Boolean = { !(i == 0) }
2 || false // Good, returns true
fn foo(b: Boolean): List = { ... }
foo(42) // Also good
1 + true // Bad, no implicit in scope.
fn b2s(b: Boolean): String = { ... }
1 ++ "Hello" // Bad, we cannot apply two conversions
```

An implicit conversion is a function with the qualifier `implicit`. It must have a single parameter. At any point in the program, when an expression `e` of type `T1` is found but one of type `T2` is expected, the compiler searches the current module for an implicit conversion of type `(T1) => T2`. If exactly one such conversion `f` is found, the compiler will substitute the `e` by `f(e)` (and the program typechecks). If multiple such conversions are found, the compiler fails with an ambiguous implicit error. If none is found, an ordinary type error is emitted.

Only a single conversion is allowed to apply to an expression. For example, in the above example, we cannot implicitly apply `i2b` and then `b2s` to get a `String`.

## 3.4   Tuples (1)

Add support for tuples in Amy. You should support tuple types, literals, and patterns:

```
fn maybeNeg(v: (Int, Boolean)): (Int, Boolean) = { // Type
  v match {
    case (i, false) => // pattern
      (i, false) // literal
    case (i, true) =>
      (-i, false)
  }
}
```

There are two ways you could approach this problem:

- Treat tuples as built-in language features. In this case, you need to support tuples of arbitrary size.

- Desugar tuples into case classes. A phase after parsing and before name analysis will transform all tuples to specified library classes, e.g. `Tuple2, Tuple3` etc. In this case, you cannot support tuples of arbitrary size, but you still need to support all sizes up to, say, 10. With this approach, you don't have to modify any compiler phases from the name analysis onwards, except maybe to print error messages that make sense to the user.

## 3.5  Improved string support (1+)

Improve string support for Amy. As a starting point, you can add functionality like substring, length, and replace, which will require you to write auxiliary WebAssembly or JavaScript code. To avoid adding additional trees, you can represent these functions as built-in methods in `Std`.

If you want a more elaborate project, you can add `Char` as a built-in type, which opens the door for additional functionality with Strings. You can also implement string interpolation. In general, look at Java/Scala strings for inspiration.

## 3.6  Higher-order functions (2+, challenging)

Add support for higher-order functions to Amy. You need to support function types and anonymous functions.

```
fn compose(f: Int => Int, g: Int => Int): Int => Int = {
  (x: Int) => f(g(x))
}
compose((x: Int) => x + 1, (y: Int) => y * 2)(5) // returns 11

fn map(f: Int => Int, l: List): List = {
  l match {
    case Nil() => Nil()
    case Cons(h, t) => Cons(f(h), map(f, t))
  }
}
map( (x: Int) => x + 1, Cons(1, Cons(2, Cons(3, Nil()))) )
  // Returns List(2, 3, 4)

fn foo(): Int => Int = {
  val i: Int = 1;
  val res: Int => Int = (x: Int) => x + i
    // Problem! How do we access i from within res?
  res
}
foo()(42) // Returns 43
```

You have to think how to represent higher order functions during runtime.

In a bytecode setting, a first approach is to represent a higher-order function as a pointer to a named function, which is then called indirectly. You have to read about tables and indirect calls in WebAssembly.

This works fine for `compose` or `map` above, but not for `foo`. The problem is that higher order functions can refer to variables in their scope, like `res` above refers to `i`. The set of those variables are called the *environment* of the function. If its environment is empty, the function will be called *closed*. Above, we have no way to refer to `i` from within `res` at runtime: `i` is in the frame of `foo` which

is not accessible in `res`. In fact, by the time we need `i`, `foo` may have returned and its frame disappeared!

The way to solve this problem is a technique called *closure conversion*. The idea is the following: At runtime, a function are represented as a *closure*, i.e. a function pointer along with the environment it captures from its scope. When we create a closure at runtime, we create a pair of values in memory, one of which points to the code (which will be a function) and the other to the environment, which will be a list of the captured variables. When we call the function, we really call the function pointer in the closure. We need to make sure to extract and somehow pass to the function pointer its environment from the other pointer. You can find a detailed explanation of closure conversion here (or pdf).

In the interpreter, things are simpler in both cases: you can define a new value type `FunctionValue` which contains all necessary information. In fact, you should probably start here as an exercise.

For your project, we recommend that you first asssume all functions in the source code are closed, but if you are motivated, you can try to implement closure conversion.

## 3.7 Custom operators (2)

Allow the user to define operators.

```
operator fn :::(l1: List, l2: List): List = {
  l1 match {
    case Nil() => l2
    case Cons(h, t) => Cons(h, t ::: l2)
  }
}


Cons(1, Cons(2, Nil())) ::: Cons(3, Nil()) // returns List(1, 2, 3)
```

You can choose specific priorities for the operators based e.g. on their first character, or you can allow the user to define it; e.g. `operator 55 fn :::(...)` could signify that `:::` has a precedence between `+` and `*` (with `||` having 10, up to `*` having 60).

You can also choose to have built-in binary operators of Amy subsumed by this project. Of course, their implementation will be left to be hard-coded by the compiler backend:

```
operator 50 fn +(i1: Int, i2: Int): Int = { error("+") }
```

In any case, your parser will be in no position to know what operators are available in your program before actually parsing it. Therefore, when you have more than one operators in a row, your parser will just have to parse the tree as a flat sequence of operand, operator, operand, ..., and then fix the mess afterwards. Of course other solutions are welcome.

## 3.8 Improved Parameters (2)

Add support for named and default parameters for functions and classes. If a value for a parameter with a default value is not given, the compiler completes the default value. One can choose to explicitly name parameters when calling a function/constructor, which also allows reordering:

```
fn foo(i: Int, j: Int = 42): Int = { i + j }
foo(1) // OK, j has default value
foo(i = 5, j = 7) // OK
foo(j = 5, i = 7) // OK, can reorder named parameters
foo(i = 7) // OK
foo(j = 7) // Wrong, i has no default value
foo() // Wrong, i has no default value


foo(i: Int = 5, j: Int): Int = { i + j }
  // Wrong, default parameters have to be at the end

// Similarly for case classes
case class Foo(i: Int, j: Int = 42) extends Bar
```

Notice that names for case class parameters are currently not preserved in the AST, which you will have to change.

## 3.9 List comprehensions (2)

Extend Amy with list comprehensions, which allow programmers to succinctly express transformations of `Lists`.

```
val xs: L.List = L.Cons(1, L.Cons(2, L.Cons(3, L.Nil())));
val ys: L.List = [ 2*x for x in xs if x % 2 != 0 ];
Std.printString(L.toString(ys)) // [2, 6]
```

Your list-comprehension syntax should support enumerating elements from one or multiple lists, filtering them with and mapping them to arbitrary expressions.

It is up to you to decide whether to treat these comprehensions as primitives in your compiler. If you do so, you will have a dedicated AST node for comprehensions in the entire compiler pipeline and generate specific code or interpret them accordingly in the end. Alternatively, you can *desugar* list comprehensions earlier in the pipeline, e.g. right after (or during) parsing. You could, for instance, generate auxiliary functions that compute the result of the list comprehension and are called in place of the comprehensions.

## 3.10 Inlining (1+)

Implement inlining on the AST level, that is, allow users to force the compiler to inline certain functions and perform optimizations on the resulting AST.

```
inline fn abs(n: Int): Int = { if (n < 0) -n else n }

abs(123); // inlined and constant-folded to '123'
abs(-456); // inlined and constant-folded to '456'
// inlined, not cf-ed; careful with side-effects!
abs(Std.readInt())
```

Inlining is effective when we can expect optimizations to make code significantly more efficient given additional information on function arguments. At a minimum, you would add an `inline` qualifier for function definitions and perform *constant folding* on inlined function bodies.

Inlining is particularly useful when applied to auxiliary functions that only exist for clarity. While inlining can lead to *code explosion* when applied too liberally, note that inlining a non-recursive function that is only called in a single location will strictly reduce code size and potentially lead to more efficient code. This makes it very attractive to *automatically* apply inlining to such functions:

```
fn foo(n: Int): Int = {
  fn plus1(n: Int): Int = { n + 1 }
  inline fn times2(n: Int): Int = { 2 * n }
  plus1(times2(times2(n))) // inlined and cf-ed to '4 * n + 1'
}

fn bar(): Int = {
  fn fib(n: Int): Int = {
    if (n <= 2) { 1 }
    else { fib(n-2) + fib(n-1) }
  }
  fib(10) // should *not* be automatically inlined
}
```

To incentivize the user to break functions down into the composition of many auxiliary functions we can introduce *local function definitions*. That is, the user may define a function within a function. For this project it is sufficient to enforce local functions that only have access to their own parameters and locals, but not the surrounding function's parameters or locals.

### 3.11   More `Int`s (1+)

Allow other integer types than `Int(32)`. You can start by adding `Int(64)`. Those should support the same operations as `Int(32)` (+, *, ...), and you should allow some conversions between them.

An easy way to do so is by adding conversion functions to the standard library:

```
val i: Int(32) = 5;
val j: Int(64) = Std.int32To64(i) // Possible function name
```

but it would be more interesting to allow some kind of implicit conversion:

```
  val i: Int(32) = 5;
  val j: Int(64) = i // Implicit conversion
```

in any case, you will have to think if you want to support narrowing conversions
and how:

```
  val i: Int(64) = 4294967296; // = 2^32
  val j: Int(32) = i // What should happen ?
```

# 4   Type systems

## 4.1   Polymorphic types (2)

Allow polymorphic types for functions and classes.

```
abstract class List[A]
case class Nil[A]() extends List[A]
case class Cons[A](h: A, t: List[A]) extends List[A]

fn length[A](l: List[A]): Int = {
  l match {
    case Nil() => 0
    case Cons(_, t) => 1 + length(t)
  }
}

case class Cons2[A, B](h1: A, h2: B, t: List[A]) extends List[A]
  // Wrong, type parameters don't match
```

You can assume the sequence of type parameters of an extending class is
identical with the parent in the extends clause (see example).

## 4.2   Case class subtyping (2)

Add subtyping support to Amy. Case classes are now types of their own:

```
val y: Some = Some(0) // Correct, Some is a type
val x: Option = None() // Correct, because None <: Option
val z: Some = None() // Wrong

y match {
  case Some(i) => () // Correct
  case None() => () // Wrong
}
```

Since case classes are types, you can declare a variable, parameter, ADT
field or function return type to be of a case class type, like any other type.

Case class types are subtypes of their parent (abstract class) type. This means you can assign case class values to variables declared with the parent type. Since we have subtyping, you can now optionally support the `Nothing` type in source code, which is a subtype of every type and the type of `error` expressions.

For this project you will probably rewrite the type checking phase in its entirety. Rather than dealing with explicit constraints, the resulting phase could perform more classical type-checking based on the minimal type satisfying all the local subtyping constraints (the so-called *least-upper bound*).

## 4.3 Arrays and range types

In both of the following two projects you would add fixed-size arrays of integers as a primitive language feature along with a type system that allows users to specify the range of integers. The information about an integer's range can then be used to make array accesses safe by ensuring that indices are in-bounds. The difference between the two projects lies in *when* integer bounds are checked, i.e., at compile-time (*statically*) or at runtime (*dynamically*).

In either case you will add two kinds of types: First, a family of primitive types `array`[$n$] that represent integer arrays of size $n$. Furthermore, *range types* that represent subsets of `Int` taking the following form: [$i$ .. $j$] where $i$ and $j$ are integer constants. The intended semantics is for [$i$ .. $j$] to represent a signed 32-bit integer $n$ such that $i \leq n \leq j$.

### 4.3.1 Dynamically-checked range types (2)

Your type system should allow users to specify *concrete* ranges, e.g., [0 .. 7] to denote integers $0 \leq n \leq 7$. Values of `Int` and any range types will be compatible during type-checking, but your system will have to be able to detect when an integer might not fall within a given range at runtime. During code generation your task will then be to emit *runtime checks* to ensure that, e.g., an `Int` in fact falls within the range [0 .. 7].

```
// initialize an array of size 8:
val arr: array[8] = [10, 20, 30, 40, 50, 60, 70, 80];
arr[0]; // okay, should not emit any runtime check
arr[arr.length-1]; // okay, same as above
// also okay, but should emit a runtime bounds check:
arr[Std.readInt()];
```

In effect, your system will ensure that array accesses are always in-bounds, i.e., do not over- or under-run an array's first, respectively last, element. Note that the resulting system should only emit the minimal number of runtime checks to ensure such safety. For instance, consider the following program:

```
fn printBoth(arr1: array[8], arr2: array[8], i: [0 .. 7]): Unit = {
  Std.printInt(arr1[i], arr2[i])
}
```

```
val someInt: Int = 4;
printBoth([1,2,3,4,5,6,7,8], [8,7,6,5,4,3,2,1], someInt)
```

Here it is not necessary to perform any checks in the body of `printBoth`, since whatever values are passed as arguments for parameter `i` should have previously been checked to lie between 0 and 7. In this concrete case, a runtime check should occur when `someInt` is passed to `printBoth`.

### 4.3.2   Statically-checked range types (2+, challenging)

Your type system should be strict and detect potential out-of-bounds array accesses early. In particular, when your type checker cannot prove that an integer lies in the required range it should produce a type error (and stop compilation as usual).

```
val arr: array[8] = [10, 20, 30, 40, 50, 60, 70, 80];
arr[0]; // okay
arr[arr.length-1]; // okay
arr[arr.length]; // not okay, type error "Idx 8 is out-of-bounds"
val i: Int = Std.readInt();
arr[i]; // not okay, type error "Int may be out-of-bounds"
if (i >= 0 && i < 8) {
  arr[i] // okay, branch is only taken when i is in bounds
}
```

To allow as many programs as possible to be accepted your type-checker will have to employ precise typing rules for arithmetic expressions and if-expressions. What you will implement are simple forms of *path sensitivity* and *abstract intepretation*.

**Constant-bounds version (2)**   Implement statically-checked range types for arrays of fixed and statically-known sizes. Range types will only involve constant bounds and in effect your type-checker will only have to accept programs that operate on arrays whose sizes are known to you as concrete integer constants.

The typing rules that you come up with should be sufficiently strong to prove safety of simple array manipulations such as the following:

```
fn printArray(arr: array[4], i: [0 .. 4]): Unit = {
  if (i < arr.length) {
    Std.printInt(arr[i]);
    printArray(arr, i+1)
  }
}

printArray([1,2,3,4], 0)
```

**Dependently-typed version (3)**   Rather than relying on the user to provide the exact sizes of arrays, also allow arrays to be of a fixed, but not statically-known size. To enable your type system to accept more programs, you should also extend the notion of range types to allow bounds *relative to* a given array's size.

The resulting types will extend the above ones in at least two ways: In addition to `array`[*n*] there is a special form `array`[*] which represents an array of arbitrary (but fixed) size. For a range type [*i* .. *j*] *i* and *j* may not only be integer constants, but may also be expressions of the form `arr.length` + *k* where `arr` is an Array-typed variable in scope and *k* is an integer constant.

Your system should then be able to abstract over concrete array sizes by referring to some Array-typed binding's length like in the following example:

```
fn printArray(arr: array[*], i: [0 .. arr.length]): Unit = {
  if (i < arr.length) {
    Std.printInt(arr[i]);
    printArray(arr, i+1)
  }
}

printArray([1,2,3,4], 0)
printArray([1,2,3,4,5,6,7,8], 0)
```

Note that the resulting language will be *dependently-typed*, meaning that types can depend on terms. In the above example, for instance, the type of parameter i of function `printArray` depends on parameter `arr`.

# 5   Alternative frontends/backends

This section contains projects that do not modify the language features of Amy, but change the implementation of a part of the Amy compiler frontend or backend.

## 5.1   Code formatter (1)

Build a code formatter for the Amy language. A straightforward way to accomplish this would be to add a special mode (e.g. `--format`) that the user can start the compiler in. It would then only run the existing pipeline up to, say, parsing and subsequently go through a special pretty-printing phase that outputs the program according to code style rules configurable by the user. A more sophisticated version could instead work on the Token-level, allowing your formatter to be aware of whitespace, e.g., respecting new-lines that a user inserted. In any case, you will have to maintain comments (which are not part of the AST).

You can look at `scalafmt` for some inspiration.

## 5.2  Language Server (2)

Implement a language server for Amy. VSCode and similar IDEs use the [language server protocol](#) to communicate with compilers and thereby provides deep integration with a variety of programming languages. Among other things, this enables features like showing type-checking errors within the editor, jumping to definitions and looking up all usages of a given definition. Your goal is to provide such functionality for Amy by implementing an additional mode in your compiler, in which it acts as a client for the language server protocol. Your implementation should be demonstrable with VSCode and include the aforementioned features. You can using an existing library, such as [LSP4J](#), to simplify your task.

## 5.3  Formalization of Amy (1)

Develop an operational semantics for Amy and use your definitions along with Amy's typing rules to prove type safety. Note that this might require you to do some additional reading on type systems.

## 5.4  JVM backend (2)

Implement an alternative backend for Amy which outputs JVM bytecode. You can use [this library](#). You first have to think how to represent Amy values in a class-based environment, and then generate the respective bytecode from Amy ASTs.

## 5.5  C backend (3)

Implement an alternative backend for Amy which outputs C code. You have to think how to represent Amy values in C, and then generate respective C code from Amy ASTs.

## 5.6  Multiple syntax styles (1)

Allow more of Scala 3 syntaxes in Amy. Amy syntax currently mixes *braces*-style (e.g. `if`, `match`) and *end markers*-style (e.g. `if`, `match`).

The goal would be to allow both syntaxes everywhere. You can more generally take inspiration from Scala 3 new syntaxes ([here](#) and [here](#)) and implement the ones you prefer.

```
object Syntax {
  fn foo(): Int = {...}
  if (true)
    0
  else
    foo() match
      case _ => 1
    end match
  end if
}
```

# 6 Execution

This section suggests projects that change how Amy code is executed.

## 6.1 Memory deallocation (3)

Allow explicit memory deallocation by the user.

```
val x: List = Cons(1, Nil());
length(x); // OK
free(x);
length(x) // Wrong, might return garbage
```

When an object in linear memory is freed, the space it used to occupy is considered free and can be allocated again. Any further reference to the freed object is undefined behavior.

You need to change how memory allocation works in code generation to maintain a list of free blocks, which will now not be a continuous part at the end of the memory. The list should not be external, but rather implemented in the memory itself: each free block needs to contain a pointer to the next one. Each block will also need to record its size. This means that free blocks have to be of size at least 2 words. When you allocate an object, you need to look through the list of blocks for one that fits and if none does, the program should fail. Make sure you always modify the free list in the simplest way possible, i.e. the blocks in the list don't have to be in the same order as in memory.

## 6.2 Lazy evaluation (1-2)

Change the evaluation strategy of Amy to lazy evaluation. Only input and output are evaluated strictly.

```
val x: Int = (Std.printInt(42); 0); // Nothing happens
val y: Int = x + 1 // Still nothing...
Std.printInt(y); // 42 and 1 are printed
```

```
val l: List = Cons(1, Cons(2, Cons(error("lazy"), Nil()))));
  // No error is thrown

l match {
  case Nil() => () // At this point, we evaluate l just enough
                   // to know it is a Cons
  case Cons(h, t) => Std.printInt(h) // Prints 1
  case Cons(h1, Cons(h2, Cons(h3, _))) =>
    // Still no error...
    Std.printInt(h3)
    // This forces evaluation of the third list element
    // and an error is thrown!
}

// We can do neat things like define infinite lists, i.e. streams
fn countFrom(start: Int): List = Cons(start, countFrom(start + 1))
Std.printString(L.listToString(
  take(countFrom(0), 5)
) // Will terminate and return 'List(1, 2, 3, 4, 5)'
```

Each value is not evaluated until it is required. Things that are not evaluated have to live in the runtime state as *thunks*, or suspensions to be evaluated later. A thunk is essentially a closure (see Section 3.6) with memoization: it is either an already calculated value, or an expression to be evaluated and an evaluation environment. In turn, an evaluation environment is a mapping from identifiers to other thunks.

You have to make sure that pattern matching only evaluates expressions as much as needed. Maybe this will help you understand the concept.

For simplicity, you can implement lazy evaluation directly in the interpreter, i.e., as an extension of the first lab (1 person). If you implement lazy evaluation for the WebAssembly-backend, you can work on this project as a team of two. (Note that this variant might be significantly harder.)

## 6.3   Final code optimizations (1+)

Optimize the WebAssembly binary produced by your Amy compiler.

The simplest thing you can do is eliminate some obvious redundancies such as

```
i32.const 0
if (result i32)
  e1
else
  e2
end
// equivalent to e2
```

```
if (result i32)
  i32.const 1
else
  i32.const 0
end
// completely redundant
```

Preferably, you can implement a control flow analysis and some abstract

interpretations to implement more advanced optimizations, also involving local parameters.

You can have a look at these slides for some ideas on optimization.

## 6.4   Tail call optimization (1)

Implement tail call optimization for Amy. Tail-recursive functions should not create any additional stack frames, i.e. use the `call` instruction.

A way to implement tail recursive functions is to do a source-to-source transformation which transforms tail recursive functions to loops. You will need to define new ASTs.

When it comes to tail calls that are not tail recursion, things are tougher. If you feel like also handling those cases, look here for ideas.

## 6.5   Foreign-function interface (FFI) to JavaScript (2)

Design a cross-language interaction layer between Amy and JavaScript. At a minimum you should support calling JavaScript functions with primitive parameter- and result-types from Amy. You can also consider supporting calls from JavaScript into Amy. You will have to decide how WebAssembly representations of Amy objects should map to JavaScript objects. To ensure that programs can be meaningfully type-checked, you should add syntax for `external` functions, e.g.

```
object FS {
  external fn open(path: String): Int
  external fn read(fd: Int): String
  // ...
}

object Example {
  val f: Int = FS.open("/home/foo/hello.txt") // open file
  Std.printString(FS.read(f)) // print contents of hello.txt
}
```

Conversely, Amy functions exposed to JavaScript could be annotated with an `export` keyword.

Ideally you will also demonstrate your FFI's capabilities by wrapping some NodeJS or browser APIs and exposing them to Amy. For instance, you might expose the file system API of NodeJS, thus allowing Amy programs to read from and write to files. Another idea is to adapt the HTML wrapper file that we provide with the compiler and use the FFI to write an interactive browser application in Amy.

A more sophisticated version of this project would also support foreign functions involving case classes such as `List`.

## 6.6 REPL: Read-Eval-Print Loop (3)

Implement a REPL for Amy. It should support defining classes, functions and local variables, and evaluating expressions. You don't have to support redefinitions. You can take a look at the Scala REPL for inspiration.

## 6.7 Virtual machine (3)

Develop your own VM to run WebAssembly code! To simplify things, you will implement the VM in Scala. Your VM should take as input a wasm `Module` from amyc's `CodeGen` Pipeline (so you don't need to implement a parser from wasm text or binary) and execute the code contained within. Despite using Scala, you still need to follow the VM execution model as much as possible: translate labels to addresses, use an array for the memory, a stack for execution etc. You can choose the VM parameters, such as memory size, any way you choose, and hard-code built-in functions that are not already implemented in WebAssembly.

# 7 Some references

You can also take a look at the following references: they might help you find an idea, or be helpful to implement your project. All of them are available at EPFL's library.

[1] Alfred V. Aho et al. *Compilers : principles, techniques, and tools.* 2nd ed., new international ed. Pearson custom library. Harlow: Pearson, 2014. ISBN: 1292024348.

[2] Andrew W Appel and Jens Palsberg. *Modern Compiler Implementation in Java.* Cambridge: Cambridge University Press, 2002. ISBN: 9780521820608.

[3] Niklaus Wirth. *Compiler construction.* International computer science series. Harlow: Addison-Wesley, 1996. ISBN: 0201403536.