# Compiler Extensions for Amy

Computer Language Processing

LARA

Autumn 2017

## 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 indicates the maximum size of a group that is allowed to take it up next to its title. Some assignments suggest additional features which allow the group to include additional members.

## 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.

```
def 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. If you add this feature, you can add an additional member to the group.

## 3.2 Implicit parameters (1)

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

```
def 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 def i2b(i: Int): Boolean = { !(i == 0) }
2 || false // Good, returns true
def foo(b: Boolean): List = { ... }
foo(42) // Also good
1 + true // Bad, no implicit in scope.
def 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:

```
def 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 mostly some WebAssembly/JavaScript coding. To avoid adding additional trees, you can represent these functions as built-in methods in `Std`.

If you want a more elaborate project for a larger group, you can add `Char` as a build-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+)

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

```
def 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

def 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)

def 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.

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 assume all functions in the source code are closed, but if you are motivated to implement closure conversion, we will allow an additional group member.

## 3.7 Custom operators (2)

Allow the user to define operators.

```
operator def :::(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 def :::(...)` 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 def +(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.

# 4 Type systems

## 4.1 Hindley-Milner type inference (2)

Allow users to skip type annotations by inferring types for them!

```
def fact(i) = {
  if (i <= 1) { i }
  else { fact(i - 1) * i }
}
```

You should use the Hindley-Milner type inference algorithm. Notice that, without polymorphism, your work is much easier.

## 4.2 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]

def 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).

## 4.3 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. In general, you have to modify type checking to account for the subtyping relation. Take particular care of the least upper bound functions and expressions that use them.

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.

# 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 Parser combinators (1)

Reimplement the Amy parser and optionally lexer using parser combinators. You can use this library.

## 5.2 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.

## 5.3 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.

# 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 (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
def 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.

Because of the difficulty of this project, you should only implement it in the evaluator.

### 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. This would involve a larger group.

### 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 Browser mode (2)

Adapt Amy to run in the browser. You will need to define new primitive functions that make sense in a browser (buttons, popups etc) and implement them in JavaScript. You will also need to fix and adapt the html wrapper file that we provide with the compiler.

### 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.