

# From Amy to WebAssembly

Computer Language Processing

LARA

Autumn 2017

# Introduction

In this presentation, we will see how to translate an Amy program, represented as an Abstract Syntax Tree, to WebAssembly binary code.

# Running example

```
def fact(i: Int): Int = {  
  if (i < 2) { 1 }  
  else {  
    val rec: Int = fact(i-1);  
    i * rec  
  }  
}
```

# Running example - output

```
(func $fact (param i32) (result i32) (local i32)
  get_local 0
  i32.const 2
  i32.lt_s
  if (result i32)
    i32.const 1
  else
    get_local 0
    i32.const 1
    i32.sub
    call $fact
    set_local 1
    get_local 0
    get_local 1
    i32.mul
  end
)
```

# WebAssembly basics

- *Stack.* WebAssembly is a stack-based VM: All operations add/remove values from the top of the stack.  
The stack starts empty at the beginning of each function.  
Question: How will the stack evolve in our example for `fact(3)`?
- *Types.* WebAssembly supports different integer and float types. In our compiler we will only use 32-bit integers, `i32` to represent all values.

- *Locals*: The header of a function defines
  - (1) the number and type of function parameters,
  - (2) the return type of the function,
  - (3) the number and type of local variables.

Parameters and local variables are both accessible with the `getLocal` and `setLocal` commands. Both use a common numbering, with parameters coming first, followed by local variables.

In our example:

`getLocal 0` refers to the function parameter (`i` in `Amy`),  
`setLocal 1` to the local variable (`rec`).

# Structured blocks

- WebAssembly offers the loop and block structured control constructs.
- Branch instructions are used to jump to the beginning, respectively end, of the construct.

*This is only possible from within the block.* This is to simulate control constructs of higher-level languages, where one cannot jump in the middle of a loop/if-then-else etc.

- Example:

```
loop $label
  ...
  br $label // Good, jump to $label
end
br $label // Bad, outside the block
```

# If-construct

Webassembly offers an if-else construct, similar to high level languages:

```
i32.const 1
if
  ...
else
  ...
end
```

`if` will pop a value from the stack and will interpret it as a boolean condition. It will then execute the `if` branch iff the popped value is nonzero, otherwise it will execute the `else` branch. In the above example, the `if` branch will be executed.



# Control constructs and accessing stack values

Control constructs are *not* allowed to pop values from the stack that were there before the start of the construct.

E.g.

```
i32.const 0
```

```
i32.const 1
```

```
if
```

```
    i32.eqz // Wrong!
```

```
    ...
```

Although the stack has the value 0 when we enter the `if`, we are not allowed to access it within the if-block.

# Control constructs and returning stack values

The `if` and `block` are typed. Each construct *must* leave on the stack a value of the specified type.

For `if`, the two versions we will use are

`if` (no value) and `'if (result i32)'` (an `i32` value)

Examples:

```
if
  i32.const 0 // Wrong
else
  i32.const 0
  get_local 0
  i32.add
  set_local 0 // Correct
end
```

```
if (result i32)
  i32.const 0 // Correct
else
  i32.const 0
  get_local 0 // Wrong
end
```

# Wasm memory

- In the header of a WebAssembly module, we can declare a *linear memory* object. Memory is basically an array of bytes, from where we can load and store values:
  - `i32.load` will pop a value `addr` from the stack, fetch an `i32` value from address `addr` of the memory, and push the value onto the stack
  - `i32.store` will pop 2 values `addr`, `v` from the stack, and store `v` on address `addr` in the memory.
- Memory is indexed by `i32`.
- We will use the linear memory as the program heap, to store heap-allocated values, i.e. strings and ADTs.
- We will use a global variable (*memory boundary*) to represent the first available address on memory.

Every time we allocate a value in the memory, we will increase the memory boundary accordingly.
- We don't have garbage collection; when the memory is full, our program fails.

# Representing Amy values

We mentioned we will only use i32 values to represent all values. But how do we represent all different Amy values as integers?

- Amy integers are conveniently defined as 32-bit integers, so we can represent them directly.
- We represent booleans as follows: `false` with 0, `true` with 1.
- We represent the unit literal with 0.

# Representing strings in wasm

- Strings are sequences of bytes in the linear memory. Each byte corresponds to the ASCII code of the corresponding character in the string. We use 0-terminated strings, and pad them to use space in memory in multiples of 4 bytes.
- For example, "Hello" will be the sequence (72, 101, 108, 108, 111, 0, 0, 0) (8 bytes total). The empty string is (0, 0, 0, 0).
- A string will be represented by the address of its first character in memory. For example, if a function allocates a string starting at address 48 and needs to return it to the caller, it will return the i32 value 48.
- Strings only support two operations and we provide helper functions for them, so they will be trivial for you to implement.

# Representing ADTs in wasm

- ADTs are also heap-allocated values.
- To represent an ADT, we need to somehow represent in memory its constructor and its fields.
- The fields are just values themselves, so they will be represented as i32 values, like all other values.
- To represent the constructor, we will assign an individual *index* or code to every constructor of a type. For example, if we have a `List` type with `Nil` and `Cons` constructors, we could assign 0 to `Nil` and 1 to `Cons`. The index is stored in memory before the fields of the ADT.

Conveniently, we already created those indexes in the symbol table (calling the `getConstructor` method returns a `ConstrSig`, which contains a field `index`).

- An ADT is represented by a reference to its base address in memory, i.e. as the address of its first word in memory (the address of the index), which by the way is also an i32!

# Allocating ADT values

- 1 Save the old memory boundary  $b$
- 2 Increment memory boundary by the size of the allocated ADT
- 3 Store the constructor index to address  $b$
- 4 For each field of the constructor, generate code for it and store it in memory in the correct offset from  $b$
- 5 Push  $b$  to the stack (base address of the ADT)

Note: Feel free to use fresh locals as temporary storage!

# Allocating ADT values: Example

```
abstract class List
case class Nil() extends List
case class Cons(h: Int, t: List) extends List
def foo(): List = { Cons(5, Cons(42, Nil())) }
```

Suppose at some point during the execution of the program, the memory boundary is 100. We encounter a call to `foo`.

After the call, the memory could look like this:

Address:	100	104	108	112	116	120	124
Content:	1	5	112	1	42	124	0
Meaning:	Cons(	5,	→	Cons(	42,	→	Nil()

The memory boundary would be 128 and `foo` would return 100 to its caller



# Compiling pattern matching (1)

A pattern matching expression

```
e match {  
  case p1 => e1  
  ...  
  case pn => en  
}
```

can be considered to be equivalent to the following pseudocode:

```
val v = e;  
if      (matchAndBind(v, p1)) e1  
else if (matchAndBind(v, p2)) e2  
else ...  
else if (matchAndBind(v, pn)) en  
else error("Match error!")
```

## Compiling pattern matching (2)

*matchAndBind*(*v*, *p*) is a function not expressible in Amy which you will have to implement in WebAssembly. It will examine if the pattern *p* matches with a value *v*, and make sure it binds the correct values to the identifiers in *p*.

*matchAndBind* is defined as follows:

```
matchAndBind(v, _) = true
matchAndBind(v, id) = { id = v; true } // assign v to id
matchAndBind(v, lit) = { v == lit } // lit is a literal
matchAndBind(C_1(v_1, ..., v_n), C_2(p_1, ..., p_m)) = {
  C_1 == C_2 &&
  matchAndBind(v_1, p_1) &&
  ...
  matchAndBind(v_m, p_m) }
```

# Compiling pattern matching (3)

You have to translate the pseudocode above to binary code.  
Think about

- how you translate the above to the wasm postfix format
- when you have to push values on the stack
- when you have to drop useless values from the stack
- when you need to use extra local variables as temporary storage
- in case of case class patterns, how the object is laid out in memory and how you can access its constructor and fields.
- how to nest if-blocks correctly.
- why is it safe to make the recursive calls to *matchAndBind*?  
How are we certain the memory even contains the things we expect?

## Extended example: List concatenation

```
object L extends App {  
  abstract class List  
  case class Nil() extends List  
  case class Cons(h: Int, t: List) extends List  
  
  def concat(l1: List, l2: List): List = {  
    l1 match {  
      case Nil() => l2  
      case Cons(h, t) => Cons(h, concat(t, l2))  
    }  
  }  
}
```

Find the commented output code on the course website.

For further examples, don't hesitate to use the reference compiler!