Next, let’s add support for:

- **Multiple datatypes** (number and boolean)
- **Calling external functions**

In the process of doing so, we will learn about:

- **Tagged Representations**
- **Calling Conventions**

**Plan**

Our plan will be to (start with boa) and add the following features:

- **Representing** boolean values (and numbers)
- **Arithmetic Operations**
- **Arithmetic Comparisons**
- **Dynamic Checking** (to ensure operators are well behaved)
1. Representation

Motivation: Why booleans?
In the year 2018, its a bit silly to use
• 0 for false and
• non-zero for true.
But really, boolean is a stepping stone to other data
• Pointers,
• Tuples,
• Structures,
• Closures.

4

The Key Issue

How to distinguish numbers from booleans?
• Need to store some extra information to mark values as number or bool.

5

Option 1: Use Two Words

First word is 1 means bool, is 0 means number, 2 means pointer etc.

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation (HEX)</th>
<th>DEC</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0000000000000001</td>
<td>1</td>
<td>0x00000001</td>
</tr>
<tr>
<td>5</td>
<td>0x0000000000000005</td>
<td>5</td>
<td>0x00000005</td>
</tr>
<tr>
<td>12</td>
<td>0x000000000000000c</td>
<td>12</td>
<td>0x0000000c</td>
</tr>
<tr>
<td>42</td>
<td>0x000000000000002a</td>
<td>42</td>
<td>0x0000002a</td>
</tr>
<tr>
<td>FALSE</td>
<td>0x0000000000000000</td>
<td>0</td>
<td>0x00000000</td>
</tr>
<tr>
<td>TRUE</td>
<td>0x0000000000000000</td>
<td>0</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

Pros
• Can have lots of different types, but

Cons
• Takes up double memory,
• Operators +, - do two memory reads [eax], [eax - 4].

In short, rather wasteful. Don’t need as many types.
Option 2: Use a Tag Bit

Can distinguish two types with a single bit. Least Significant Bit (LSB) is

- 0 for number
- 1 for boolean

Why not 0 for boolean and 1 for number?

Tag Bit: Numbers

So number is the binary representation shifted left by 1 bit
- Lowest bit is always 0
- Remaining bits are number’s binary representation

For example,

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation (Binary)</th>
<th>Value</th>
<th>Representation (HEX)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>[0b...00000110]</td>
<td>3</td>
<td>[0x00000006]</td>
</tr>
<tr>
<td>5</td>
<td>[0b...00000101]</td>
<td>5</td>
<td>[0x0000000a]</td>
</tr>
<tr>
<td>12</td>
<td>[0b...00001100]</td>
<td>12</td>
<td>[0x00000018]</td>
</tr>
<tr>
<td>42</td>
<td>[0b...01010100]</td>
<td>42</td>
<td>[0x00000054]</td>
</tr>
</tbody>
</table>

Tag Bit: Booleans

Most Significant Bit (MSB) is

- 1 for true
- 0 for false

For example,

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation (Binary)</th>
<th>Value</th>
<th>Representation (HEX)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>[0b010000000001]</td>
<td>TRUE</td>
<td>[0x00000001]</td>
</tr>
<tr>
<td>FALSE</td>
<td>[0b010000000001]</td>
<td>FALSE</td>
<td>[0x00000001]</td>
</tr>
</tbody>
</table>
Types

Let's extend our source types with boolean constants.

```haskell
data Expr a = ... |
  Boolean Bool a
```

Correspondingly, we extend our assembly `Arg` (values) with

```haskell
data Arg = ... |
  HexConst Int
```

So, our examples become:

<table>
<thead>
<tr>
<th>Value</th>
<th>Representation (HEX)</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>HexConst 0x00000001</td>
</tr>
<tr>
<td>True</td>
<td>HexConst 0x80000001</td>
</tr>
<tr>
<td>3</td>
<td>HexConst 0x00000006</td>
</tr>
<tr>
<td>5</td>
<td>HexConst 0x0000000a</td>
</tr>
<tr>
<td>12</td>
<td>HexConst 0x00000018</td>
</tr>
<tr>
<td>42</td>
<td>HexConst 0x0000002a</td>
</tr>
</tbody>
</table>

Transforms

Next, let's update our implementation.

The `parse`, `anf` and `tag` stages are straightforward. Let's focus on the `compile` function.

A TypeClass for Representing Constants

It's convenient to introduce a type class describing Haskell types that can be represented as x86 arguments:

```haskell
class Repr a where
  repr :: a -> Arg
```

We can now define instances for `Int` and `Bool` as:

```haskell
instance Repr Int where
  repr = Const (Data.Bits.shift n 1)  -- left-shift `n` by 1

instance Repr Bool where
  repr False = HexConst 0x00000001
  repr True  = HexConst 0x00000001
```
Immediate Values to Arguments

Boolean b is an immediate value (like Number n).

Let's extend immArg that transforms an immediate expression to an x86 argument.

\[
\text{immArg :: Env -> ImmTag -> Arg}
\]

\[
\text{immArg (Var x _)} = \ldots
\]

\[
\text{immArg (Number n _)} = \text{repr n}
\]

\[
\text{immArg (Boolean b _)} = \text{repr b}
\]

Compiling Constants

Finally, we can easily update the compile function as:

\[
\text{compileEnv :: Env -> AnfTagE -> Asm}
\]

\[
\text{compileEnv _ e} @ (\text{Number} _ _) = \text{IMov (Reg EAX) (immArg env e)}
\]

\[
\text{compileEnv _ e} @ (\text{Boolean} _ _) = \text{IMov (Reg EAX) (immArg env e)}
\]

(The other cases remain unchanged.)
Let's run some tests to double check.

Output Representation

Say what?! Ah. Need to update our run-time printer in main.c

```c
void print(int val){
  if (val == CONST_TRUE)
    printf("true");
  else if (val == CONST_FALSE)
    printf("false");
  else // should be a number!
    printf("%d", val >> 1); // shift right to remove tag bit.
}
```

Can you think of some other tests we should write?
2. Arithmetic Operations

Constants like 2, 29, false are only useful if we can perform computations with them.

First let’s see what happens with our arithmetic operators.

Shifted Representation and Addition

We are representing a number \( n \) by shifting it left by 1. \( n \) has the machine representation \( 2^n \n \)

Thus, our source values have the following representations:

<table>
<thead>
<tr>
<th>Source Value</th>
<th>Representation (DEC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>6 + 10 = 16</td>
<td></td>
</tr>
<tr>
<td>n1 + n2</td>
<td>2<em>n1 + 2</em>n2 = 2*(n1 + n2)</td>
</tr>
</tbody>
</table>

That is, addition (and similarly, subtraction) works as is with the shifted representation.

Shifted Representation and Multiplication

We are representing a number \( n \) by shifting it left by 1. \( n \) has the machine representation \( 2^n \n \)

Thus, our source values have the following representations:

<table>
<thead>
<tr>
<th>Source Value</th>
<th>Representation (DEC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>6 + 10 = 60</td>
<td></td>
</tr>
<tr>
<td>n1 + n2</td>
<td>2<em>n1 + 2</em>n2 = 4*(n1 + n2)</td>
</tr>
</tbody>
</table>

Thus, multiplication ends up accumulating the factor of 2. The result is two times the desired one.
Strategy

Thus, our strategy for compiling arithmetic operations is simply:

• Addition and Subtraction “just work” as before, as
  shifting “cancels out”,
• Multiplication result must be “adjusted” by dividing-by-two
  • i.e. right shifting by 1

Types

The source language does not change at all, for the Asm lets add a “right shift” instruction (shr):

```haskell
data Instruction = ...
  | IShr Arg Arg
```

Transforms

We need only modify compileEnv to account for the “fixing up”

```haskell
compileEnv :: Env -> AnfTagE -> [Instruction]
compileEnv env (Prim2 o v1 v2) = compilePrim2 env o v1 v2
where the helper compilePrim2 works for Prim2 (binary) operators and immediate arguments:
```
Transforms

\[
\text{compilePrim2 :: Env} \rightarrow \text{Prim2} \rightarrow \text{ImmE} \rightarrow \text{ImmE} \rightarrow [\text{Instruction}]
\]

\[
\text{compilePrim2 env Plus v1 v2} = \begin{cases} 
\text{IMov \ (Reg EAX) \ (immArg env v1)} \\
\text{IAdd \ Reg EAX \ (immArg env v2)} 
\end{cases}
\]

\[
\text{compilePrim2 env Minus v1 v2} = \begin{cases} 
\text{IMov \ Reg EAX \ (immArg env v1)} \\
\text{ISub \ Reg EAX \ (immArg env v2)} 
\end{cases}
\]

\[
\text{compilePrim2 env Times v1 v2} = \begin{cases} 
\text{IMov \ Reg EAX \ (immArg env v1)} \\
\text{IMul \ Reg EAX \ (immArg env v2)} \\
\text{IShr \ Reg EAX \ (Const 1)} 
\end{cases}
\]

Tests

Let’s take it out for a drive.

What does “2 * (-1)” evaluate to?

2147483644

Whoa!)

Well, its easy to figure out if you look at the generated assembly:

\[
\begin{align*}
\text{mov eax, 4} \\
\text{imul eax, -2} \\
\text{shr eax, 1} \\
\text{ret}
\end{align*}
\]

The trouble is that the negative result of the multiplication is saved in twos-complement format, and when we shift that right by one bit, we get the weird value (does not “divide by two”)

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hexadecimal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8</td>
<td>FFFFFFF8</td>
<td>0b11111111111111111111111111111000</td>
</tr>
<tr>
<td>2147483644</td>
<td>7FFFFFFC</td>
<td>0b01111111111111111111111111111100</td>
</tr>
</tbody>
</table>

Solution: Signed/Arithmetic Shift

The instruction \text{shr} shift arithmetic right does what we want, namely:

- preserves the sign-bit when shifting
- i.e. doesn’t introduce a 0 by default
Transforms Revisited

Let's add `sar` to our target:

```plaintext
data Instruction
  ∗ : ...↓ ISar Arg Arg
```

and use it to fix the post-multiplication adjustment

- i.e. use `ISar` instead of `IShr`

```plaintext
compilePrim2 env Times v1 v2 = [ IMov (Reg EAX) (immArg env v1) , Imul (Reg EAX) (immArg env v2) , ISar (Reg EAX) (Const 1) ]
```

After which all is well:

- "2 * (-1)"
- produces -2

3. Arithmetic Comparisons

Next, let's try to implement comparisons:

Many ways to do this:

- branches `jne, jl, jg`
- bit-twiddling.

Comparisons via Bit-Twiddling

Key idea: negative number's most significant bit is 1

To implement `arg1 < arg2`, compute `arg1 - arg2`

- When result is negative, MSB is 1, ensure `eax` set to `0x80000001`
- When result is non-negative, MSB is 0, ensure `eax` set to `0x00000000`
- Can extract `msb` by bitwise `and` with `0x80000000`
- Can set tag bit by bitwise `or` with `0x00000001`

So compilation strategy is:

```plaintext
mov eax, arg1
sub eax, arg2
and eax, 0x80000000 ; mask out "sign" bit (msb)
or eax, 0x80000001 ; set tag bit to bool
```
Comparisons: Implementation

Let's go and extend:

- The `Instruction` type

```haskell
data Instruction = ...
  | IAnd Arg Arg
  | IOr Arg Arg
```

- The `instrAsm` converter

```haskell
instrAsm :: Instruction -> Text
instrAsm (IAnd a1 a2) = ...
instrAsm (IOr a1 a2) = ...
```

- The actual `compilePrim2` function

Exercise: Comparisons via Bit-Twiddling

- Can compute `arg1 > arg2` by computing `arg2 < arg1`.
- Can compute `arg1 == arg2` by computing `arg1 < arg2 || arg2 < arg1`.
- Can compute `arg1 = arg2` by computing `!(arg1 != arg2)`

For the above, can you figure out how to implement:

- Boolean `??`
- Boolean `||`
- Boolean `&&`

You may find these instructions useful

4. Dynamic Checking

We've added support for `Number` and `Boolean` but we have no way to ensure that we don't write gibberish programs like:

```
2 + true
```

or

```
7 < false
```

In fact, let's try to see what happens with our code on the above:

```
ghci> exec "2 + true"
Oops.
```
Checking Tags at Run-Time

Later, we will look into adding a static type system. For now, let’s see how to abort execution when the wrong types of operands are found when the code is executing.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Op-1</th>
<th>Op-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>-</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>*</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>&lt;</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>&gt;</td>
<td>int</td>
<td>int</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>bool</td>
<td>bool</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>bool</td>
<td></td>
</tr>
<tr>
<td>if</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Strategy

Let’s check that the data in eax is an int:

- Suffices to check that the LSB is 0
- If not, jump to special error_non_int label

For example, to check if arg is a Number:

```assembly
mov eax, arg
mov ebx, eax ; copy into ebx register
and ebx, 0x00000001 ; extract lsb
cmp ebx, 0 ; check if lsb equals 0
jne error_non_number...
```

at error_non_number we can call into a C function:

```c
error_non_number:
push eax ; pass erroneous value
push 0 ; pass error code
call error ; call run-time "error" function
```

Strategy

Finally, the error function is part of the run-time and looks like:

```c
void error(int code, int v){
    if (code == 0) {
        fprintf(stderr, "Error: expected a number but got %010x\n", v);
    } else if (code == 1) {
        // print out message for errorcode 1 ...
    } else if (code == 2) {
        // print out message for errorcode 2 ...
    } ...
    exit(1);
}
```
Strategy By Example

```assembly
section .text
extern error
extern print
global our_code_starts_here
our_code_starts_here:
    mov eax, 1 ; not a valid number
    mov ebx, eax ; copy into ebx register
    and ebx, 0x00000001 ; extract lsb
    cmp ebx, 0 ; check if lsb equals 0
    jne error_non_number
error_non_number:
    push eax
    push 0
    call error
make tests/output/int-check.result
... segmentation fault ...

What happened?
```

Managing the Call Stack

To properly call into C functions (like `error`), we must play by the rules of the C calling convention:

1. The local variables of an (executing) function are saved in its stack frame.
2. The start of the stack frame is saved in register `ebp`.
3. The start of the next frame is saved in register `esp`.

Calling Convention

We must preserve the above invariants as follows:

In the Callee:

At the start of the function

```assembly
push ebp ; save (previous, caller's) ebp on stack
mov ebp, esp ; make current esp the new ebp
sub esp, 4*N ; "allocate space" for N local variables
```

At the end of the function

```assembly
mov esp, ebp ; restore value of esp to that just before call
pop ebp ; new value at [ebp] is caller's (saved) ebp
ret ; return to caller
```
Calling Convention

We must preserve the above invariants as follows:

In the Caller:
To call a function \texttt{target} that takes \( N \) parameters:

\begin{verbatim}
push arg_N  \quad / push last arg first ...  
push arg_{N-1}  \quad / then the second ...  
push arg_{N-2}  \quad / finally the first  
call target  \quad / make the call (which puts return addr on stack)  
add esp, \ 4*N  \quad / now we are back: "clear" args by adding 4*numArgs
\end{verbatim}

NOTE: If you are compiling on MacOS, you must respect the 16-Byte Stack Alignment Invariant.

Fixed Strategy By Example

Let's implement the above in a simple file \texttt{tests/output/int-check.s}.

\begin{verbatim}
section .text
section .data
section .rodata
section .data
section .text

our_code_starts_here:
push ebp
mov ebp, esp
sub esp, 0  \; 0 local variables here
mov eax, 1  \; not a valid number
mov ebx, eax  \; copy into ebx register
and ebx, 0x00000001  \; extract lsb
cmp ebx, 0  \; check if lsb equals 0
jne error_non_number
mov esp, ebp
pop ebp
ret

error_non_number:
push eax
push 0
call error
\end{verbatim}

Aha, now the code works!

\begin{verbatim}
make tests/output/int-check-result
\end{verbatim}

Q: What NEW thing does our compiler need to compute?

Hint: Why do we \texttt{sub esp, 0} above?

Types

Let's implement the above strategy.

To do so, we need a new data type for run-time types:

\begin{verbatim}
data Ty = TNumber | TBoolean
\end{verbatim}

a new \texttt{Label} for the error

\begin{verbatim}
data Label
= ... | TypeError Ty  \quad -- Type Error Labels  
| BuiltIn Text  \quad -- Functions implemented in C
\end{verbatim}

and that's it.
Transforms

The compiler must generate code to:

- Perform dynamic type checks,
- Exit by calling \texttt{error} if a failure occurs,
- Manage the stack per the convention above.

### 1. Type Assertions

The key step in the implementation is to write a function

```haskell
assertType :: Env -> IExp -> Ty -> [Instruction]
assertType env v ty = [ IMov (Reg EAX) (immArg env v) , IMov (Reg EBX) (Reg EAX) , IAnd (Reg EBX) (HexConst 0x00000001) , ICmp (Reg EBX) (typeTag ty) , IJne (TypeError ty) ]
```

where \( \text{typeTag} \) is:

```haskell
typeTag :: Ty -> Arg
typeTag TNumber = HexConst 0x00000000
typeTag TBoolean = HexConst 0x00000001
```

You can now splice \texttt{assertType} prior to doing the actual computations, e.g.

```haskell
compilePrim2 :: Env -> Prim2 -> ImmE -> ImmE -> [Instruction]
compilePrim2 env Plus v1 v2 = assertType env v1 TNumber ++ assertType env v2 TNumber ++ IMov (Reg EAX) (immArg env v1) ++ IAdd (Reg EAX) (immArg env v2)
```
2. Errors

We must also add code at the TypeError TNumber and TypeError TBoolean labels.

```haskell
errorHandler :: Ty -> Asm
errorHandler t =
  the expected-number error
  [ ILabel (TypeError t)
    -- push the second "value" param first,
    , IPush (Reg EAX)
      -- then the first "code" param,
    , IPush (ecode t)
      -- call the run-time's "error" function.
    , ICalls (Builtin "error")
  ]

ecode :: Ty -> Arg
ecode TNumber = Const 0
ecode TBoolean = Const 1
```

3. Stack Management

Local Variables

First, note that the local variables live at offsets from ebp, so let's update

```haskell
immArg :: Env -> ImmTag -> Arg
immArg (Number n _) = Const n
immArg env (Var x _) = RegOffset EBP i

where
  i = fromMaybe err (lookup x env)
  err = error (printf "Error: Variable '%%s' is unbound" x)
```

Maintaining esp and ebp

We need to make sure that all our code respects the C calling convention.

To do so, just wrap the generated code, with instructions to save and restore esp and ebp.

```haskell
compiledBody :: AnfTagE -> Asm
compiledBody e = compiledBody e emptyEnv
  ++ compiledEnv e emptyEnv
  ++ exitCode e

entryCode :: AnfTagE -> Asm
entryCode e =
  [ IPush (Reg EBP),
    IMove (Reg ESP) (Reg EBP),
    ISub (Reg ESP) (Const 4 * n)
  ]

where
  n = countVars e

exitCode :: AnfTagE -> Asm
exitCode e =
  [ IMove (Reg ESP) (Reg EBP),
    IPop (Reg EBX),
    IRet
  ]
```

```python
```
3. Stack Management

Q: But how shall we compute \( \text{countVars} \)?

Here’s a shady kludge:

\[
\text{countVars} :: \text{AnfTagE} \rightarrow \text{Int}
\]

\[
\text{countVars} = 100
\]

Obviously a sleazy hack (why?), but let’s use it to test everything else; then we can fix it.

4. Computing the Size of the Stack

Once everything (else) seems to work, let’s work out:

\[
\text{countVars} :: \text{AnfTagE} \rightarrow \text{Int}
\]

Finding the exact answer is undecidable in general, i.e. is impossible to compute.

However, it is easy to find an over-approximate heuristic, i.e.

- a value guaranteed to be larger than the than the max size,
- and which is reasonable in practice.

Strategy

Let \( \text{countVars} \ e \) be:

- The maximum number of let-binds in scope at any point inside \( e \), i.e.
- The maximum size of the \( \text{Env} \) when compiling \( e \)

Let’s work it out on a case-by-case basis:

- Immediate values like \text{Number} or \text{Var}
  - are compiled without pushing anything onto the \( \text{Env} \)
  - i.e. \( \text{countVars} = 0 \)
- Binary Operations like \text{Prime2 o v1 v2}
  - take immediate values,
  - are compiled without pushing anything onto the \( \text{Env} \)
  - i.e. \( \text{countVars} = 0 \)
- Branches like \text{If v e1 e2}
  - can go either way
  - can’t tell at compile-time
  - i.e. worst-case is larger of \( \text{countVars} \ e1 \) and \( \text{countVars} \ e2 \)
- Let-bindings like \text{Let x e1 e2}
  - require
    - evaluating \( e1 \) and
    - pushing the result onto the stack and then evaluating \( e2 \)
  - i.e. larger of \( \text{countVars} \ e1 \) and \( 1 + \text{countVars} \ e2 \)
Implementation

We can implement the above a simple recursive function:

```haskell
countVars :: AnfTagE -> Int
countVars (If v e1 e2) = max (countVars e1) (countVars e2)
countVars (Let x e1 e2) = max (countVars e1) (1 + countVars e2)
countVars _ = 0
```

Naive Heuristic is Naive

The above method is quite simplistic. For example, consider the expression:

```haskell
let x = 1,
    y = 2,
    z = 3
in 0
```

`countVars` would tell us that we need to allocate 3 stack spaces but clearly none of the variables are actually used.

Will revisit this problem later, when looking at optimizations.

Recap

We just saw how to add support for

- Multiple datatypes (number and boolean)
- Calling external functions

and in doing so, learned about

- Tagged Representations
- Calling Conventions

To get some practice, in your assignment, you will add:

- Dynamic Checks for Arithmetic Overflows (see the jo and jno operations)
- A Primitive `print` operation implemented by a function in the `c` runtime.

And next, we’ll see how to easily add user-defined functions.
Questions?