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, lets try to see what happens with our code on the above:

```
ghci> exec "2 + true"
```

Oops.

Static vs. Dynamic Type Checking

Later we will add a *static* type system

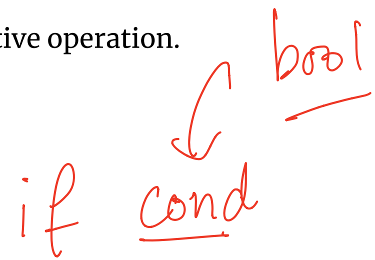
- that rejects meaningless programs at *compile* time.

Now lets add a *dynamic* system

- that *aborts execution* with wrong *operands* at *run* time.

Checking Tags at Run-Time

Here are the **allowed** types of operands for each primitive operation.



Operation	Op-1	Op-2
`+`	`int`	`int`
`-`	`int`	`int`
`*`	`int`	`int`
`<`	`int`	`int`
`>`	`int`	`int`
`&&`	`bool`	`bool`
`||`	`bool`	`bool`

Operation	Op-1	Op-2
`!`	`bool`	
`if`	`bool`	
`=`	`int` or `bool`	`int` or `bool`

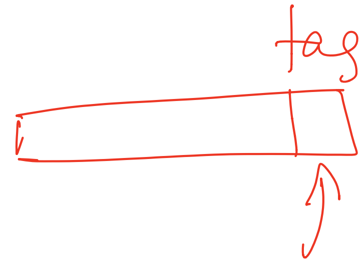
Strategy: Asserting a Type

To check if `arg` is a `number`

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

For example

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



tag = 0 int

tag = 1 bool

at `error_non_number` we can call into a `C` function:

```
error_non_number:
  mov rdi, 0              ; pass error code
  mov rsi, rax            ; pass erroneous value
  call error              ; call run-time "error" function
```

as param #1

param #2

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

rdi
rsi
rdx
rcx
r8
r9

```c
void error(long code, long 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

Lets implement the above in a simple file `tests/output/int-check.s`

```
section .text
extern error
extern print
global our_code_starts_here
our_code_starts_here:
  mov rax, 1                ; not a valid number
  mov rbx, rax              ; copy into rbx register
  and rbx, 0x00000001       ; extract lsb
  cmp rbx, 0                ; check if lsb equals 0
  jne error_non_number
error_non_number:
  mov rdi, 0
  mov rsi, rax
  call error
```

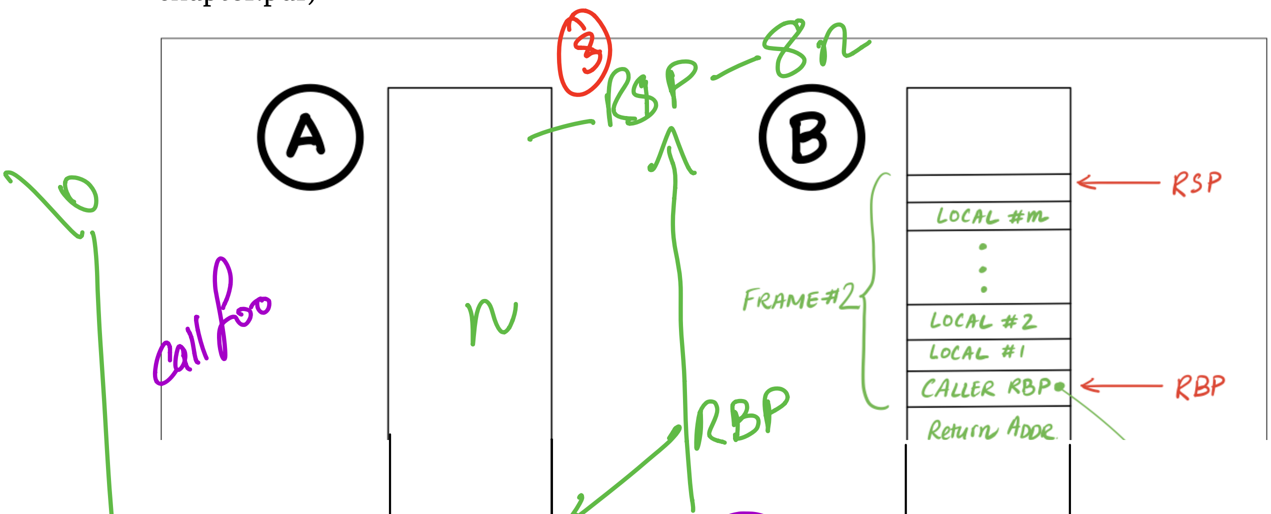
Alas

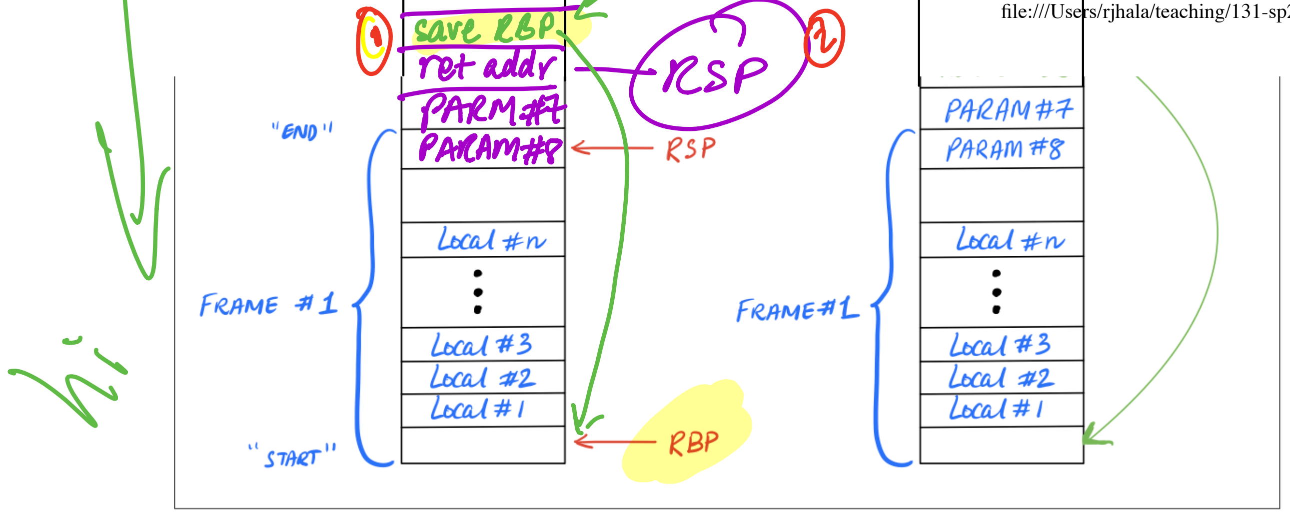
```
make tests/output/int-check.result
... segmentation fault ...
```

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 (https://aaronbloomfield.github.io/pdr/book/x86-64bit-ccc-chapter.pdf)





Stack Layout

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 `rbp` ,
3. The *start* of the *next* frame is saved in register `rsp` .

Calling Convention

We must **preserve the above invariant** as follows:

In the Callee

At the **start** of the function

```
push rbp            ; SAVE (previous) caller's base-pointer on stack
mov rbp, rsp        ; set our base-pointer using the current stack-poin
ter
sub rsp, 8*N        ; ALLOCATE space for N local variables
```

At the **end** of the function

```
add rsp, 8*N0       ; FREE space for N local variables
pop rbp             ; RESTORE caller's base-pointer from stack
ret                 ; return to caller
```

Fixed Strategy By Example

Lets implement the above in a simple file `tests/output/int-check.s`

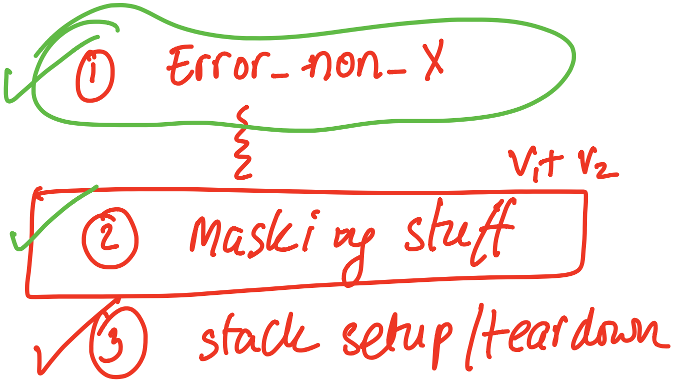
```
section .text
extern error
extern print
global our_code_starts_here
our_code_starts_here:
  push rbp                    ; save caller's base-pointer
  mov rbp, rsp                ; set our base-pointer
  sub rsp, 1600               ; alloc '100' vars

  mov rax, 1                  ; not a valid number
  mov rbx, rax                ; copy into rbx register
  and rbx, 0x00000001         ; extract lsb
  cmp rbx, 0                  ; check if lsb equals 0
  jne error_non_number

  add rsp, 1600               ; de-alloc '100' vars
  pop rbp                     ; restore caller's base-pointer
  ret
error_non_number:
  mov rdi, 0
  mov rsi, rax
  call error
```

Aha, now the above works!

```
make tests/output/int-check.result
... expected number but got ...
```

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

Hint: Why do we `sub esp, 1600` above?



Types

Lets implement the above strategy.

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

```
data Ty = TNumber | TBoolean
```

a new `Label` for the error

```haskell
data Label
  = ...
  | TypeError Ty          -- Type Error Labels
  | Builtin   Text        -- Functions implemented in C
```

and thats it.

$v_1 + v_2$

checkType v_1 Int
checkType v_2 Int
mov RAX, <v_1>
add RAX, <v_2>

Transforms

The compiler must generate code to:

1. Perform dynamic type checks,
2. Exit by calling `error` if a failure occurs,

3. Manage the stack per the convention above.

1. Type Assertions

The key step in the implementation is to write a function

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

where `typeTag` is:

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

You can now splice `assertType` prior to doing the actual computations, e.g.

```
compilePrim2 :: Env -> Prim2 -> ImmE -> ImmE -> [Instruction]
compilePrim2 env Plus v1 v2   = assertType env v1 TNumber
                             ++ assertType env v2 TNumber
                             ++ [ IMov (Reg RAX) (immArg env v1)
                                , IAdd (Reg RAX) (immArg env v2)
                                ]
```

2. Errors

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

```
errorHandler :: Ty -> Asm
errorHandler t =
  [ ILabel   (TypeError t)        -- the expected-number error
  ,   IMov   (Reg RDI) (ecode t)  -- set the first  "code" param,
  ,   IMov   (Reg RSI) (Reg RAX)  -- set the second "value" param fir
st,
  ,   ICall  (Builtin "error")    -- call the run-time's "error" func
tion.
  ]

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

3. Stack Management

Maintaining `rsp` and `rbp`

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 `rbp` and `rsp`

```
compileBody :: AnfTagE -> Asm
compileBody e = entryCode e
             ++ compileEnv emptyEnv e
             ++ exitCode e

entryCode :: AnfTagE -> Asm
entryCode e = [ IPush (Reg RBP)                       -- SAVE caller'
s RBP
              , IMov  (Reg RBP) (Reg RSP)             -- SET our RBP
              , ISub  (Reg RSP) (Const (argBytes n))  -- ALLOC n loca
l-vars
              ]
  where
    n       = countVars e

exitCode :: AnfTagE -> Asm
exitCode e = [ IAdd (Reg RSP) (Const (argBytes n))      -- FREE n loc
al-vars
             , IPop (Reg RBP)                            -- RESTORE ca
ller's RBP
             , IRet                                      -- RETURN to
caller
             ]
  where
```

```
    n       = countVars e
```

the `rsp` needs to be a multiple of `16` so:

```haskell
argBytes :: Int -> Int
argBytes n = 8 * n'
  where
    n' = if even n then n else n + 1
```

Q: But how shall we compute `countVars`?

Here's a shady kludge:

```haskell
countVars :: AnfTagE -> Int
countVars = 100
```

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

4. Computing the Size of the Stack

Ok, now that everything (else) seems to work, lets work out:

```
countVars :: AnfTagE -> Int
```

Finding the *exact* answer is **undecidable** in general (CSE 105), i.e. is *impossible* to compute.

However, it is easy to find an *overapproximate* heuristic, i.e.

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

As usual, lets see if we can work out a heuristic by example.

QUIZ

How many stack slots/vars are needed for the following program?

```
1 + 2
```

A. `0`

B. `1`

C. `2`

QUIZ

How many stack slots/vars are needed for the following program?

```
let x = 1
  , y = 2
  , z = 3
in
  x + y + z
```

A. 0

B. 1

C. 2

D. 3

E. 4

QUIZ

How many stack slots/vars are needed for the following program?

```
if true:
  let x = 1
    , y = 2
    , z = 3
  in
    x + y + z
else:
  0
```

A. 0

B. 1

C. 2

D. 3

E. 4

A. `0`

B. `1`

C. `2`

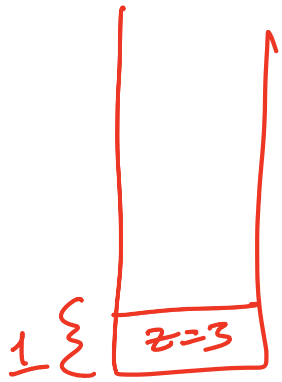
D. `3`

E. `4`

QUIZ

How many stack slots/vars are needed for the following program?

```
let x =
  let y =
    let z = 3
    in z + 1
  in y + 1
in x + 1
```



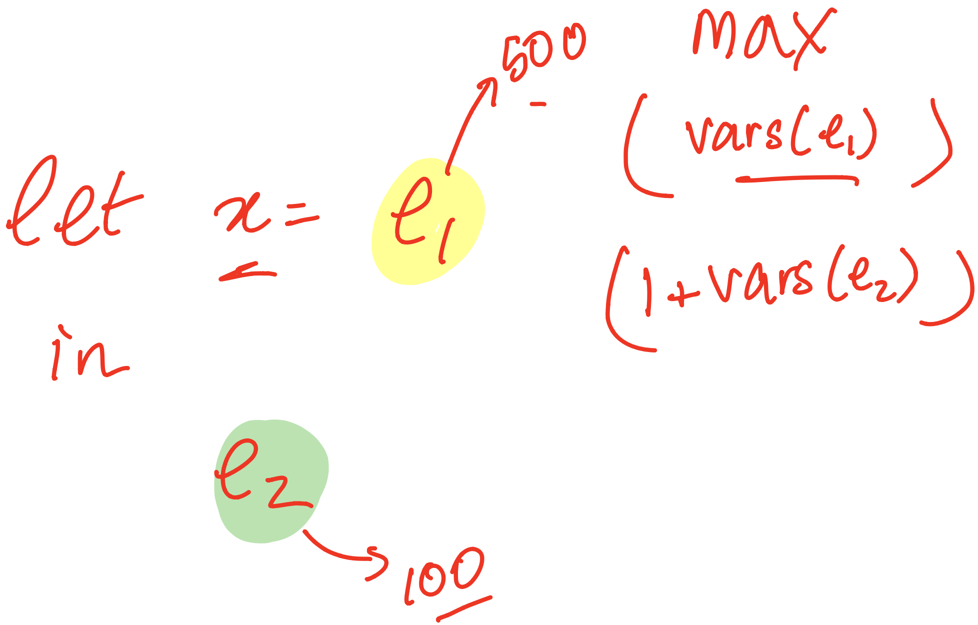
A. 0

B. 1

C. 2

D. 3

E. 4



Strategy

Let `countVars e` be:

- The *maximum* number of let-binds in scope at any point *inside* `e`, i.e.
- The *maximum* size of the `Env` when compiling `e`

Lets work it out on a case-by-case basis:

- **Immediate values** like `Number` or `Var`
  - are compiled *without pushing* anything onto the `Env`
  - i.e. `countVars` = 0
- **Binary Operations** like `Prim2 o v1 v2` take immediate values,
  - are compiled *without pushing* anything onto the `Env`
  - i.e. `countVars` = 0
- **Branches** like `If v e1 e2` can go either way
  - can't tell at compile-time
  - i.e. worst-case is larger of `countVars e1` and `countVars e2`
- **Let-bindings** like `Let x e1 e2` require
  - evaluating `e1` and

  - *pushing* the result onto the stack and then evaluating `e2`
  - i.e. larger of `countVars e1` and `1 + countVars e2`

Implementation

We can implement the above a simple recursive function:

```
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:

```
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:

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

And next, we'll see how to add **user-defined functions**.

(https://ucsd-cse131.github.io/sp21/feed.xml)

(https://twitter.com/ranjitjhala)

(https://plus.google.com/u/0/106612421534244742464)

(https://github.com/ucsd-cse131/sp21)