Language processing in Haskell

Simon Thompson

s.j.thompson@kent.ac.uk

Section 1: introduction

Masterclass

What this is and isn't all about.

Not teaching new stuff …

… putting it into practice in a bigger example.

What I assume you know

Data types: including tuples, lists, numbers and Booleans.

Haskell functions and modules.

The Haskell read / evaluate / print loop.

Using pattern matching and recursion.

Using functions on lists.

What you will learn

Using data types to represent structured data.

Processing structured data in a variety of ways.

Types: using `type` and `data`.

Building a suite of functions and an API.

Extending an existing program.

Language processing

Common example: handling a “little language”.

XML, SOAP, REST, HTML 5, …

Text processing, building/making systems, data analytics, testing, …

Domain-specific languages …

… as well as “good old fashioned compilers”.

Numerical expressions

Expressions like (2+(3*4)) …

… or (2+(3*v)), where v is a variable.

What can we do with expressions?

Evaluate them

$(2+(3*4))$ has the value 14 …

… and, if v is -2, then $(2+(3*v))$ has the value -4.

Simplify them

(0+(1*v)) simplifies to v.

Compile them for a virtual machine

$(2+(3*v))$ compiles to the instruction sequence

PUSH 2, PUSH 3, FETCH v, MUL_2, ADD_2

Section 2: representing structured data

How to represent numerical expressions?

Expressions like (2+(3*4)) …

… or (2+(3*v)), where v is a variable.

Strings?

(2+(3*4)) could be represented as the string (list) `"(2+(3*4))"` …

… why is that a bad idea?

Strings? No!

(2+(3*4)) could be represented as the string (list) `"(2+(3*4))"` …

… why is that a bad idea?

Because when we read (2+(3*4)) we see a **structure**:

… it's the addition of 2 and (3*4),

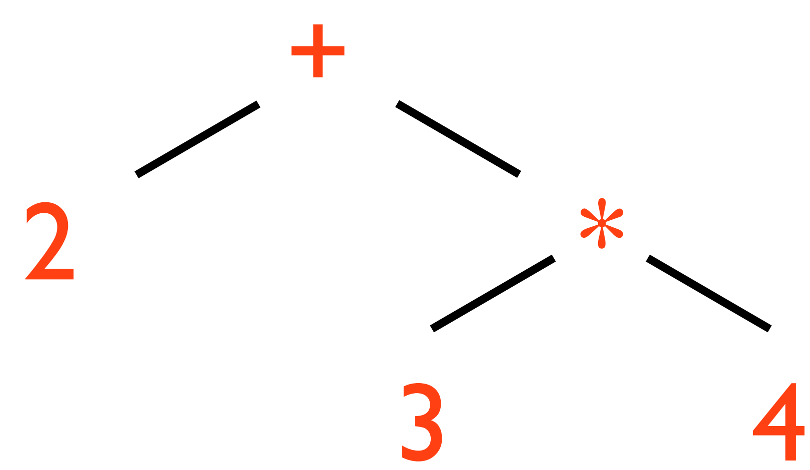
… and (3*4) is itself the multiplication of 3 and 4.

How do we represent them in a program?

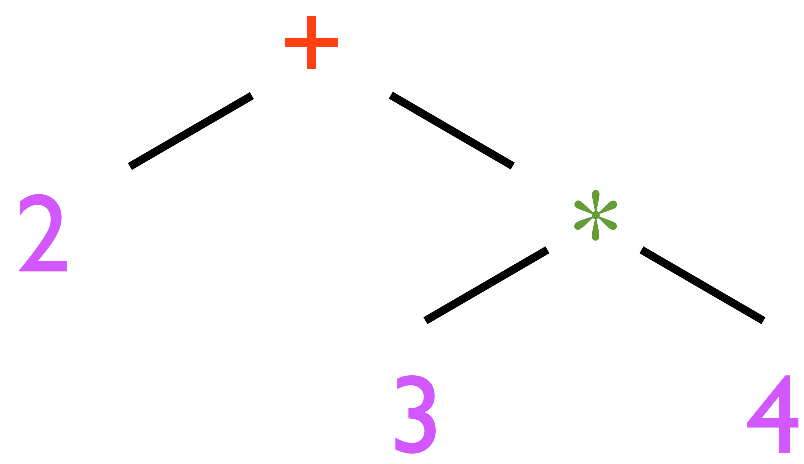
Because when we read (2+(3*4)) we see a **structure**:

… it's the addition of 2 and (3*4),

… and (3*4) is itself the multiplication of 3 and 4.



How do we represent them in a program?




```
Add (Num 2) (Mul (Num 3) (Num 4))
```

Defining a type for expressions: `Expr`

```
data Expr =
    Num Int
  | Var Char
  | Add Expr Expr
  | Mul Expr Expr
    deriving (Eq,Ord,Show)
```

Converting from `String` to `Expr`

Going from

```
"(2+(3*4))"
```

to

```
Add (Num 2) (Mul (Num 3) (Num 4))
```

is called parsing.

Converting from `Expr` to `String`

Going from

```
Add (Num 2) (Mul (Num 3) (Num 4))
```

to

```
"(2+(3*4))"
```

is called (pretty) printing.

Section 3: using recursion: pretty printing

Converting from `Expr` to `String`

Going from

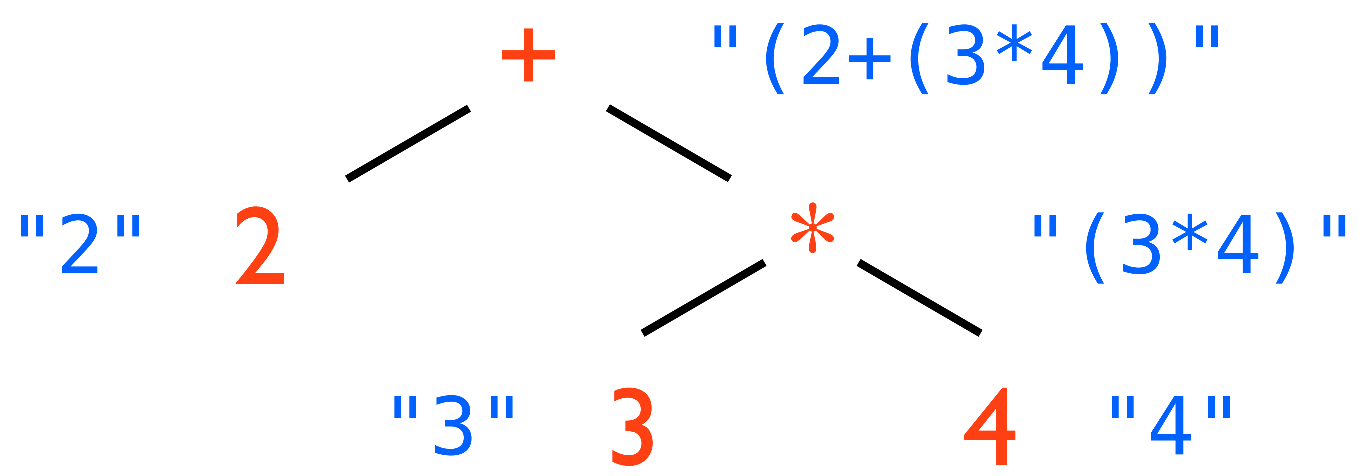
```
Add (Num 2) (Mul (Num 3) (Num 4))
```

to

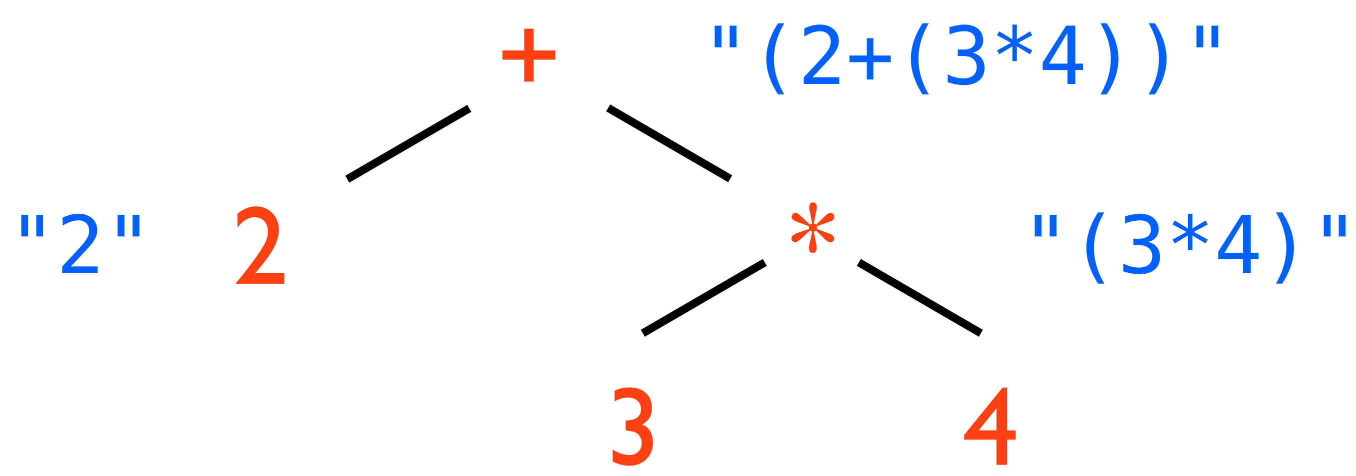
```
"(2+(3*4))"
```

is called (pretty) printing.

Bottom up …



Top down …



Printing

```haskell
print :: Expr -> String
```

Printing

```
print :: Expr -> String
print (Num n) = show n
```

Printing

```haskell
print :: Expr -> String

print (Num n) = show n
print (Var v) = [v]
```

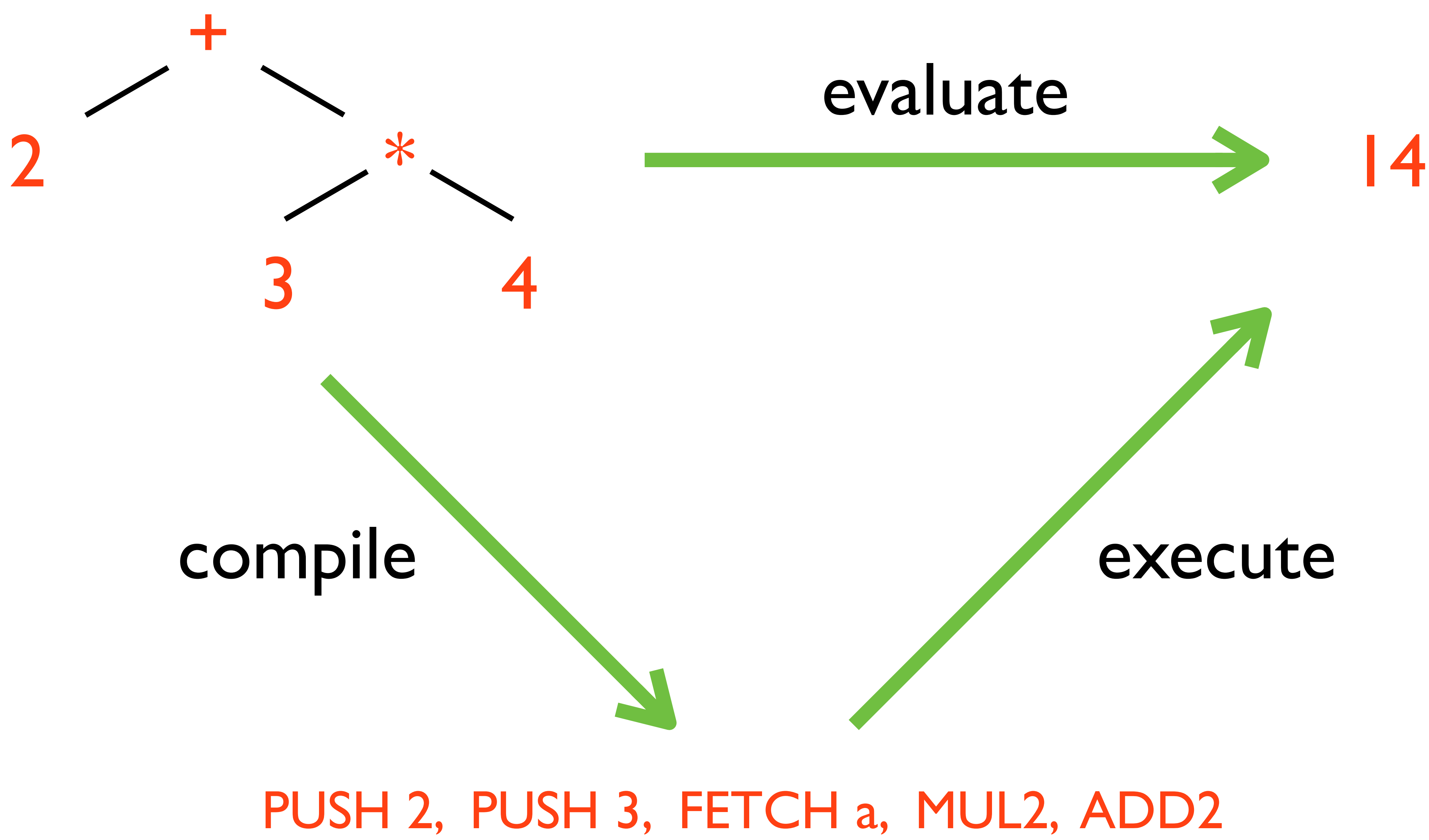
Printing

```
print :: Expr -> String

print (Num n) = show n
print (Var v) = [v]
print (Add e1 e2) =
    "(" ++ print e1 ++ "+" ++ print e2 ++")"
```

Printing

```
print :: Expr -> String

print (Num n) = show n
print (Var v) = [v]
print (Add e1 e2) =
    "(" ++ print e1 ++ "+" ++ print e2 ++")"
print (Mul e1 e2) =
    "(" ++ print e1 ++ "*" ++ print e2 ++")"
```

Section 4: evaluating expressions


+
2
*
3
4
evaluate
14
compile
execute
PUSH 2, PUSH 3, FETCH a, MUL2, ADD2

Evaluating an expression

```haskell
eval :: Expr -> Int
```

Evaluating an expression

```
eval :: Expr -> Int

eval (Num n) = n
```

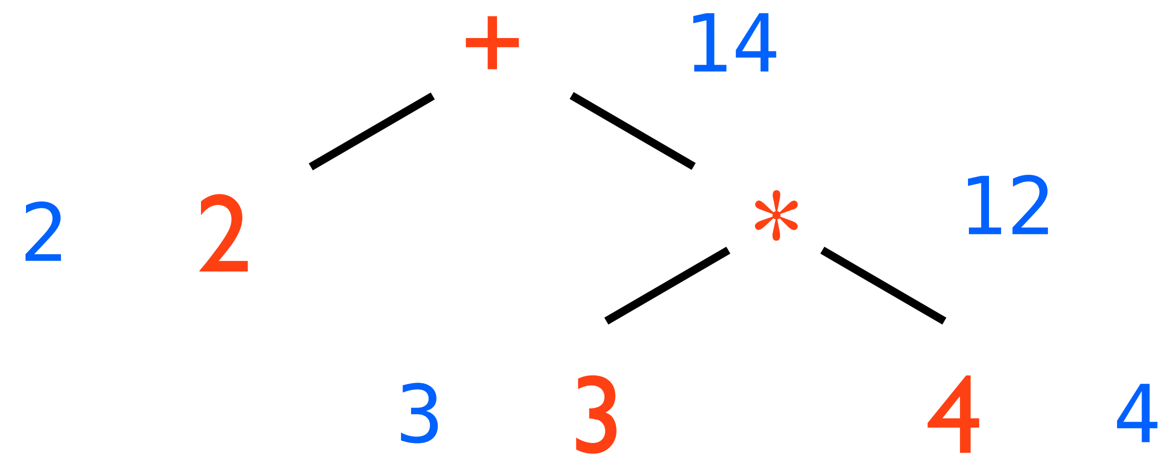
Evaluating an expression

```haskell
eval :: Expr -> Int

eval (Num n) = n
eval (Add e1 e2) = eval e1 + eval e2
```

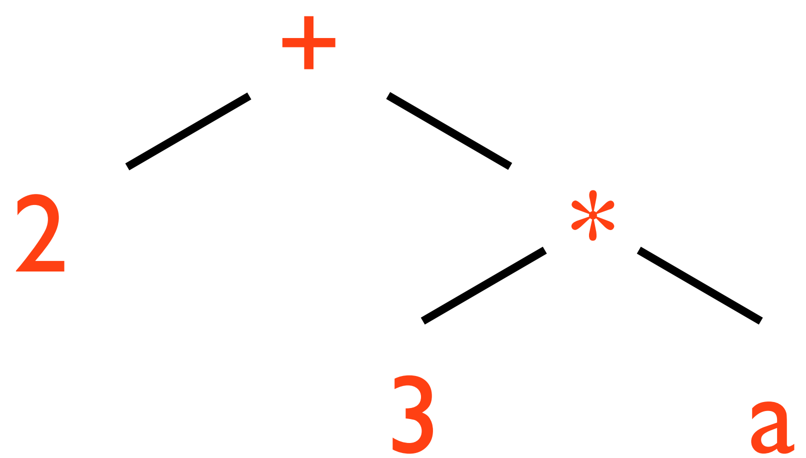
Evaluating an expression

```haskell
eval :: Expr -> Int

eval (Num n) = n
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
```

Animating the calculation



A problem



Evaluating an expression with variables

```haskell
eval :: Env -> Expr -> Int

eval env (Num n) = n
eval env (Var v) = lookup v env
eval env (Add e1 e2) = eval env e1 + eval env e2
eval env (Mul e1 e2) = eval env e1 * eval env e2
```

Evaluating an expression with variables

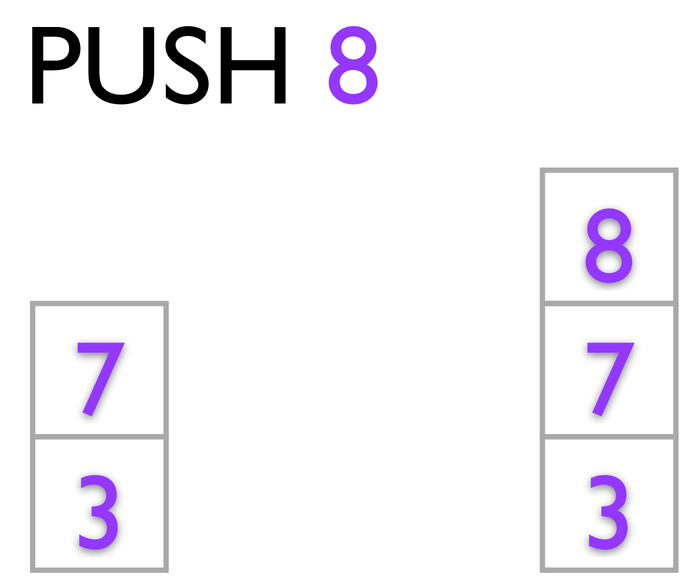
```
type Env = [(Char,Int)]

env1 = [ ('a',12), ('b', -11) ]

lookup :: Char -> Env -> Int

lookup v []
     = 0
lookup v ((x,n):ps) =
    if x==v then n else lookup v ps
```

Section 5: compiling and running on a VM

A stack virtual machine

The virtual machine has a stack, which is manipulated by the machine instructions.

For example, the **PUSH N** instruction pushes the integer **N** onto the top of the stack.



PUSH 8

	8
7	7
3	3

ADD$_2$

8	
7	15
3	3

FETCH a

	4
7	7
3	3

a	b	...
4	0	...

MUL$_2$

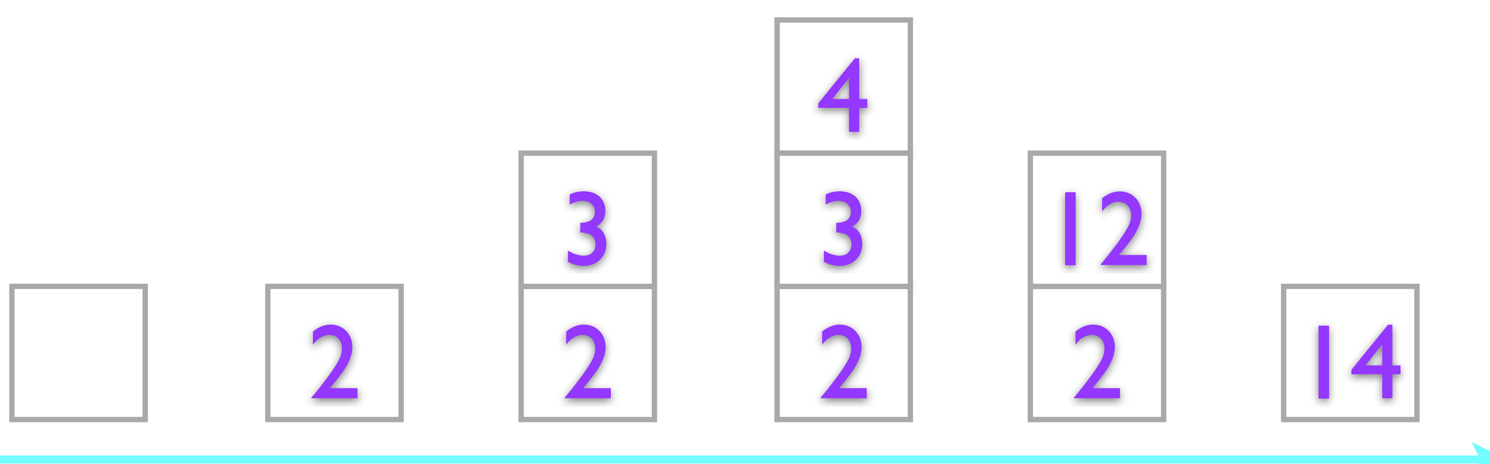
8	
2	16
3	3

A stack virtual machine

(2+(3*a)) compiles to the instruction sequence

PUSH 2, PUSH 3, FETCH a, MUL_2, ADD_2

a	b	…
4	0	…



Doing it in Haskell

We need to show how it is all implemented in Haskell.

- how to model machine instructions
- how to model the running of the machine
- how to compile an expression into a sequence of instructions.

Instructions, programs and stacks

```haskell
data Instr =
    Push Int
  | Fetch Char
  | Add2
  | Mul2

type Program = [Instr]

type Stack = [Int]
```

Compiling expressions; running programs

```
compile :: Expr -> Program

run :: Program -> Env -> Int
```

Compiling expressions; running programs

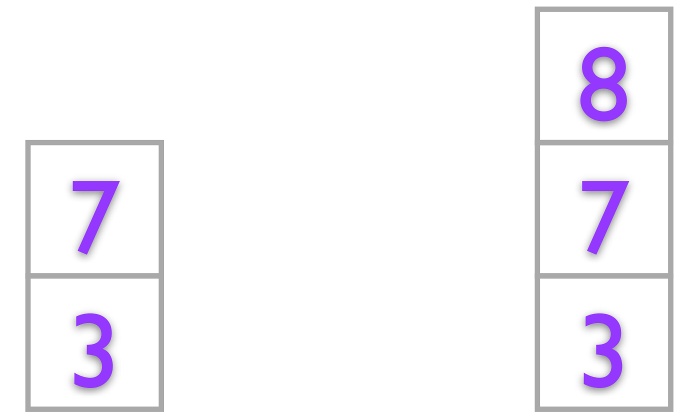
```
compile :: Expr -> Program

run :: Program -> Env -> Int

run :: Program -> Env -> Stack -> Int
```

Running the stack machine

```
run :: Program -> Env -> Stack -> Int
```

Running the stack machine

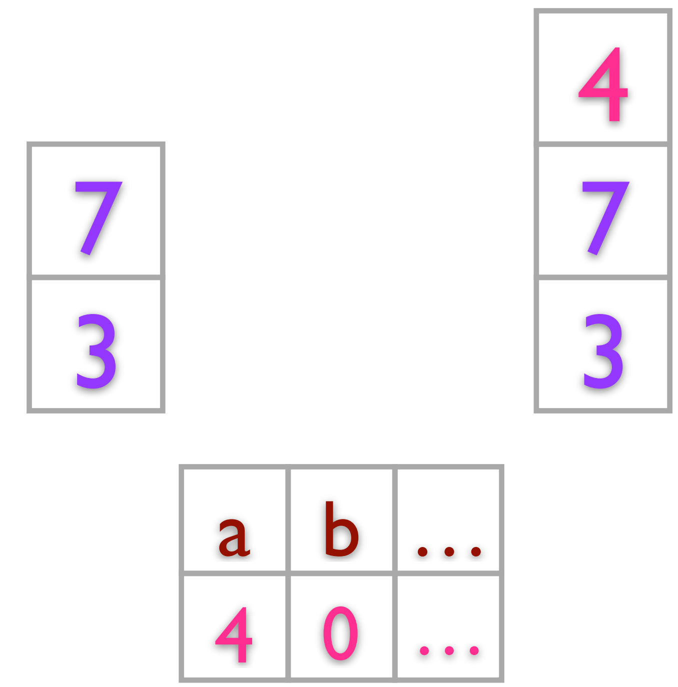
PUSH 8




```
run :: Program -> Env -> Stack -> Int

run (Push n : cont) env stack =
    run cont env (n:stack)
```

Running the stack machine

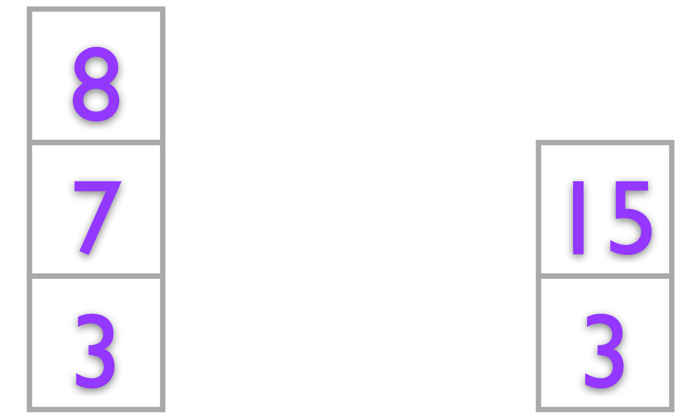
FETCH a




```haskell
run :: Program -> Env -> Stack -> Int

run (Push n : cont) env stack =
    run cont env (n:stack)
run (Fetch v : cont) env stack =
    run cont env (lookup v env : stack)
```

Running the stack machine

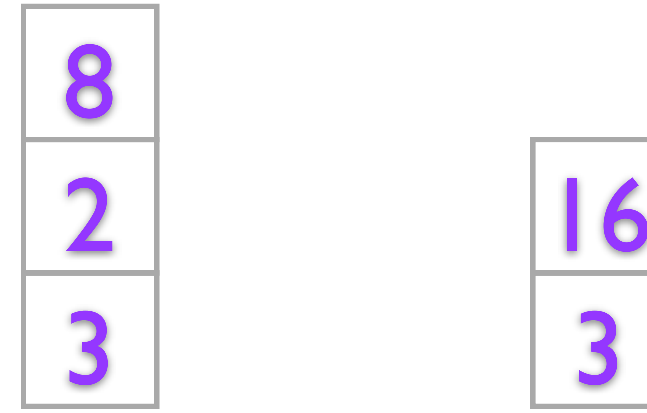
ADD$_2$




```haskell
run :: Program -> Env -> Stack -> Int

run (Push n : cont) env stack =
    run cont env (n:stack)
run (Fetch v : cont) env stack =
    run cont env (lookup v env : stack)
run (Add2 : cont) env (n1:n2:rest) =
    run cont env (n1+n2:rest)
```

Running the stack machine

MUL_2




```haskell
run :: Program -> Env -> Stack -> Int

run (Push n : cont) env stack =
    run cont env (n:stack)
run (Fetch v : cont) env stack =
    run cont env (lookup v env : stack)
run (Add2 : cont) env (n1:n2:rest) =
    run cont env (n1+n2:rest)
run (Mul2 : cont) env (n1:n2:rest) =
    run cont env (n1*n2:rest)
```

Running the stack machine

```haskell
run :: Program -> Env -> Stack -> Int

run (Push n : cont) env stack =
    run cont env (n:stack)
run (Fetch v : cont) env stack =
    run cont env (lookup v env : stack)
run (Add2 : cont) env (n1:n2:rest) =
    run cont env (n1+n2:rest)
run (Mul2 : cont) env (n1:n2:rest) =
    run cont env (n1*n2:rest)
run [] env (n:_) =
    n
```

Compilation

Numbers and (values of) variables go on the stack.

To perform an add, evaluate the two sub-expressions, putting each of the results on the stack, then add the values on top of the stack.

```
compile :: Expr -> Program

compile (Num n) = [Push n]
compile (Var v) = [Fetch v]
compile (Add e1 e2) = compile e2 ++ compile e1 ++ [Add2]
compile (Mul e1 e2) = compile e2 ++ compile e1 ++ [Mul2]
```

Lessons learned

Some of the lessons we can learn from this section are

- flexibility of data representations
- type definitions and specifications
- pattern matching
- tail recursion
- missing cases - let it fail

Section 6: parsing

Converting to and from `Expr`

Going from `"(2+(3*4))"`

to `Add (Num 2) (Mul (Num 3) (Num 4))` is called parsing.

Going from `Add (Num 2) (Mul (Num 3) (Num 4))` to

`"(2+(3*4))"` is called (pretty) printing.

Parsing

Examples

```
parse("(2+(3*4))") = Add (Num 2) (Mul (Num 3) (Num 4))

parse("2") = (Num 2)

parse("a") = (Var 'a')
```

Parsing

Examples

`parse("(2+(3*4))") = Add (Num 2) (Mul (Num 3) (Num 4))`

`parse("2") = (Num 2)`

`parse("a") = (Var 'a')`

But what about

`parse("2+(3*4))") = (Num 2)`

We've lost `"+(3*4))"`, which is still to be processed.

Getting the type right

```
parse :: [Char] -> (Expr,[Char])
```

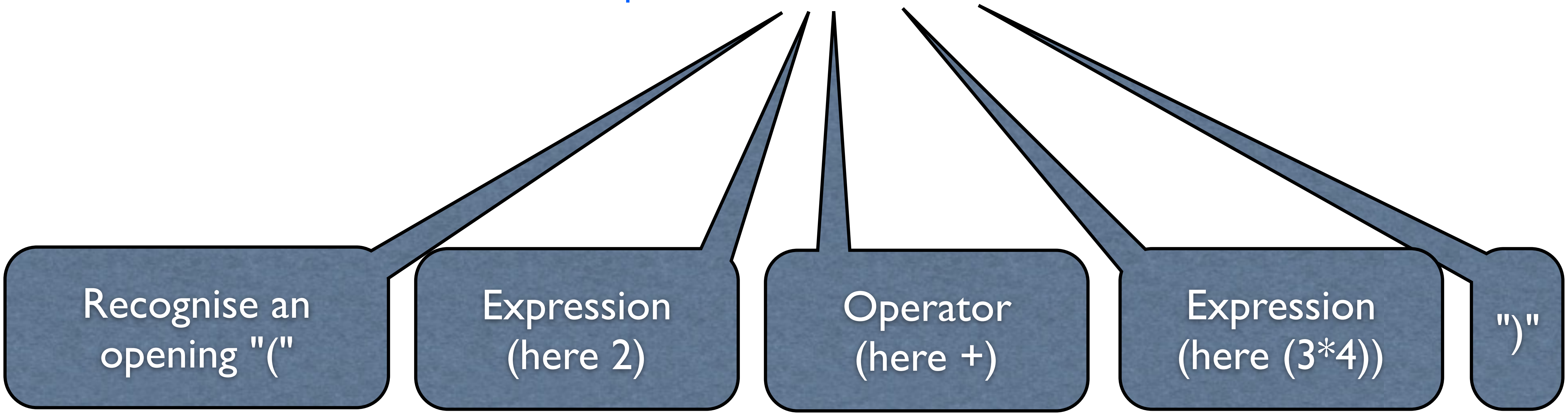
Examples

```
parse("(2+(3*4))") = (Add (Num 2) (Mul (Num 3) (Num 4)), "")
parse("2")         = ((Num 2), "")
parse("2+(3*4))") = ((Num 2), "+(3*4))")
```

Return a pair {expression, what's left of the input}.

The algorithm, informally

```
parse("(2+(3*4))")
```




```
Add (Num 2) (Mul (Num 3) (Num 4))
```

Predictive, top-down, parsing

```haskell
parse :: [Char] -> (Expr,[Char])

parse ('(':rest) =
    let (e1,rest1)  = parse rest
        (op:rest2)  = rest1
        (e2,rest3)  = parse rest2
        (')':rest4) = rest3 in
    ((case op of
        '+' -> Add e1 e2
        '*' -> Mul e1 e2),
     rest4)
```

Predictive, top-down, parsing

```
parse :: [Char] -> (Expr,[Char])

parse ('(':rest)
    let (e1,rest1)      parse rest
        (op:rest2)    = rest1
        (e2,rest3)    = parse rest2
        (')':rest4) = rest3 in
    ((case op of
        '+' -> Add e1 e2
        '*' -> Mul e1 e2),
     rest4)
```

Predictive, top-down, parsing

```
parse :: [Char] -> (Expr,[Char])

parse ('(':rest) =
    let (e1,rest1)  = parse rest
        (op:rest2)  = rest1
        (e2,rest3)  = parse rest2
        (')':rest4) = rest3 in
    ((case op of
        '+' -> Add e1 e2
        '*' -> Mul e1 e2),
     rest4)
```

Predictive, top-down, parsing

```haskell
parse :: [Char] -> (Expr,[Char])

parse ('(':rest) =
    let (e1,rest1)  = parse rest
        (op:rest2)  = rest1
        (e2,rest3)  = parse rest2
        (')':rest4) = rest3 in
    ((case op of
        '+' -> Add e1 e2
        '*' -> Mul e1 e2),
     rest4)
```

Predictive, top-down, parsing

```
parse :: [Char] -> (Expr,[Char])

parse ('(':rest) =
    let (e1,rest1)  = parse rest
        (op:rest2)  = rest1
        (e2,rest3)  = parse rest2
        (')':rest4) = rest3 in
    ((case op of
        '+' -> Add e1 e2
        '*' -> Mul e1 e2),
     rest4)
```

Predictive, top-down, parsing

```
parse :: [Char] -> (Expr,[Char])

parse ('(':rest) =
    let (e1,rest1)  = parse rest
        (op:rest2)  = rest1
        (e2,rest3)  = parse rest2
        (')':rest4) = rest3 in
    ((case op of
        '+' -> Add e1 e2
        '*' -> Mul e1 e2),
     rest4)
```

Predictive, top-down, parsing

```
parse :: [Char] -> (Expr,[Char])

parse ('(':rest) =
    let (e1,rest1)  = parse rest
        (op:rest2)  = rest1
        (e2,rest3)  = parse rest2
        (')':rest4) = rest3 in
    ((case op of
        '+' -> Add e1 e2
        '*' -> Mul e1 e2),
     rest4)
```

Predictive, top-down, parsing

```
parse :: [Char] -> (Expr,[Char])

parse ('(':rest) =
    let (e1,rest1)  = parse rest
        (op:rest2)  = rest1
        (e2,rest3)  = parse rest2
        (')':rest4) = rest3 in
    ((case op of
        '+' -> Add e1 e2
        '*' -> Mul e1 e2),
     rest4)
```

Section 7: taking things further

Going further

Simplification: (0+(1*v)) simplifies to v.

More operations: subtraction, div/rem, unary minus.

Setting variables: let v=e_1 in e_2

Defining functions for yourself: let f=(\x -> e_1) in e_2

Adding other types: if b then e_1 else e_2

Changing the syntax: e.g. operator precedence – BODMAS.

Simplification

```
simplify :: Expr -> Expr

simplify (Add e1 (Num 0)) = simplify e1

simplify (Mul (Num 1) e2) = simplify e2

simplify (Mul (Num 0) e2) = Num 0
```

Simplification

```haskell
simplify :: Expr -> Expr

simplify (Add e1 (Num 0)) = simplify e1

simplify (Mul (Num 1) e2) = simplify e2

simplify (Mul (Num 0) e2) = Num 0

simplify (Add e1 e2) =
    if not(s1==e1 && s2==e2)
        then
            simplify (Add s1 s2)
        else
            Add s1 s2
      where
        s1 = simplify e1
        s2 = simplify e2
```

Lessons learned

Some of the lessons we can learn

- flexibility of data representations
- type definitions and specifications
- type-driven development
- pattern matching
- patterns of recursion