Language processing

Simon Thompson
s.j.thompson@kent.ac.uk

CO545 Lecture 10

Section 1: langauges are everywhere

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

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),

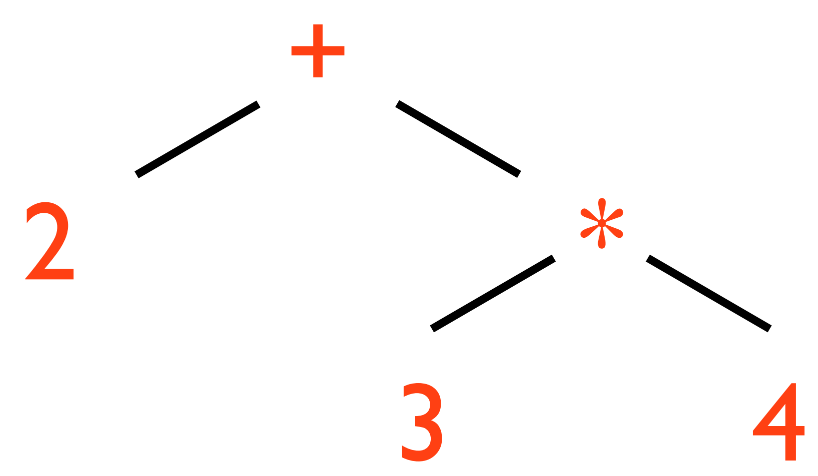
… and (3*4) is itself the multiplication of 3 and 4.

How do we represent them in a program?

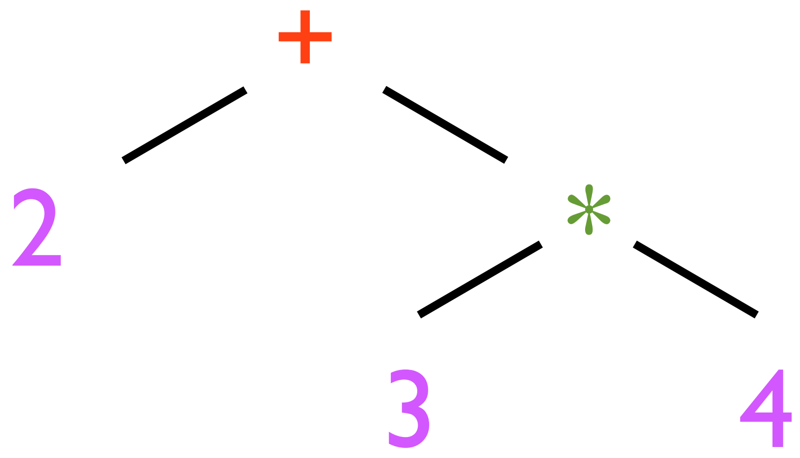
Because when we read (2+(3*4)) we see a **structure**:

… it's the addition of 2 and (3*4),

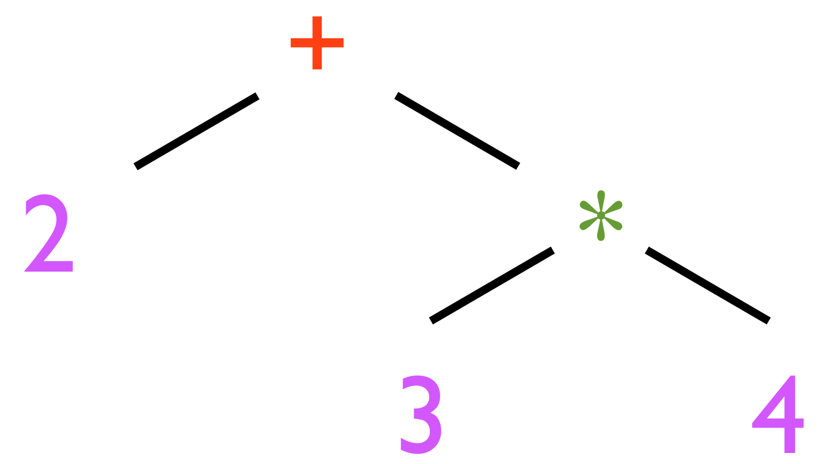
… and (3*4) is itself the multiplication of 3 and 4.



How do we represent them in a program?



How do we represent them in a program?




```
{add, {num,2}, {mul, {num,3}, {num,4}}}
```

Defining a type for expressions, `expr()`

```
-type expr() :: {'num',integer()}
             |  {'var',atom()}
             |  {'add',expr(),expr()}
             |  {'mul',expr(),expr()}.
```

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

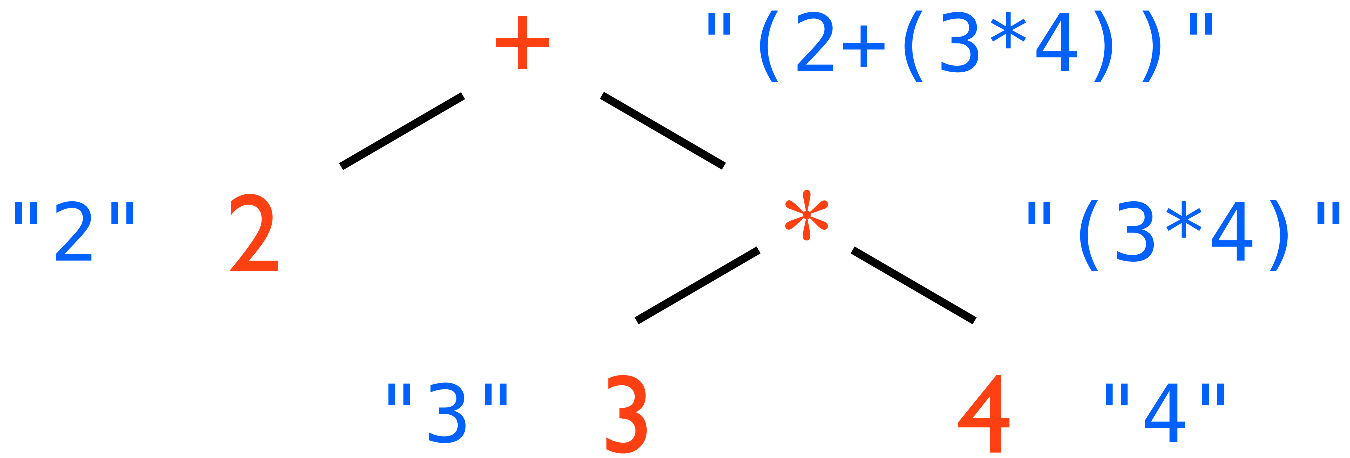
```
{add,{num,2},{mul,{num,3},{num,4}}}
```

to

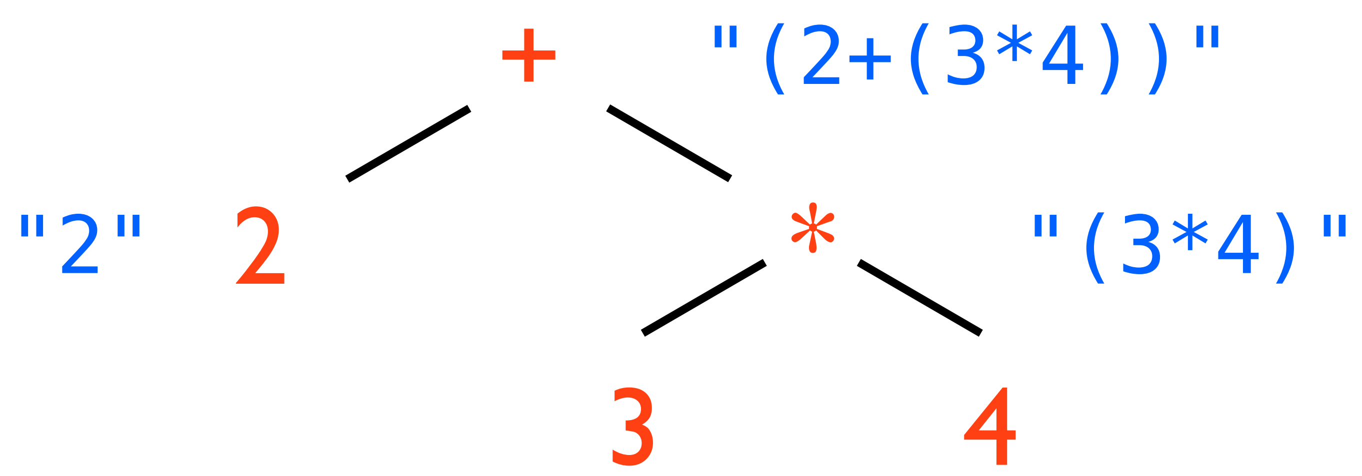
```
"(2+(3*4))"
```

is called (pretty) printing.

Bottom up …



Top down …



Printing

Turn an expression into a string.

```
-spec print(expr()) -> string().
```

Printing

Turn an expression into a string.

```
-spec print(expr()) -> string().

print({num,N}) ->
    ... ;
print({var,A}) ->
    ... ;
print({add,E1,E2}) ->
    ... ;
print({mul,E1,E2}) ->
    ... .
```

Printing

Turn an expression into a string.

```
-spec print(expr()) -> string().

print({num,N}) ->
    integer_to_list(N);
print({var,A}) ->
    ... ;
print({add,E1,E2}) ->
    ... ;
print({mul,E1,E2}) ->
    ... .
```

Printing

Turn an expression into a string.

```
-spec print(expr()) -> string().

print({num,N}) ->
    integer_to_list(N);
print({var,A}) ->
    atom_to_list(A);
print({add,E1,E2}) ->
    ... ;
print({mul,E1,E2}) ->
    ... .
```

Printing

Turn an expression into a string.

```
-spec print(expr()) -> string().

print({num,N}) ->
    integer_to_list(N);
print({var,A}) ->
    atom_to_list(A);
print({add,E1,E2}) ->
    “(" ++ print(E1) ++ "+" ++ print(E2) ++ ")";
print({mul,E1,E2}) ->
    ... .
```

Printing

Turn an expression into a string.

```
-spec print(expr()) -> string().

print({num,N}) ->
    integer_to_list(N);
print({var,A}) ->
    atom_to_list(A);
print({add,E1,E2}) ->
    “(" ++ print(E1) ++ "+" ++ print(E2) ++ ")";
print({mul,E1,E2}) ->
    “(" ++ print(E1) ++ "*" ++ print(E2) ++ ")".
```

Working it out

```
print({num,N}) ->
    integer_to_list(N);
print({var,A}) ->
    atom_to_list(A);
print({add,E1,E2}) ->
    “(" ++ print(E1) ++ "+" ++ print(E2) ++ ")";
print({mul,E1,E2}) ->
    “(" ++ print(E1) ++ "*" ++ print(E2) ++ ")".

print({add,{num,2},{mul,{num,3},{num,4}}})
```

Working it out

```
print({num,N}) ->
    integer_to_list(N);
print({var,A}) ->
    atom_to_list(A);
print({add,E1,E2}) ->
    “(" ++ print(E1) ++ "+" ++ print(E2) ++ ")";
print({mul,E1,E2}) ->
    “(" ++ print(E1) ++ "*" ++ print(E2) ++ ")".

print({add,{num,2},{mul,{num,3},{num,4}}})
```

Working it out

```
print({num,N}) ->
    integer_to_list(N);
print({var,A}) ->
    atom_to_list(A);
print({add,E1,E2}) ->
    “(" ++ print(E1) ++ "+" ++ print(E2) ++ ")";
print({mul,E1,E2}) ->
    “(" ++ print(E1) ++ "*" ++ print(E2) ++ ")".

print({add,{num,2},{mul,{num,3},{num,4}}})
 -> "("++ print({num,2}) ++"+"++ print({mul,{num,3},{num,4}}) ++")"
```

Working it out

```
print({num,N}) ->
    integer_to_list(N);
print({var,A}) ->
    atom_to_list(A);
print({add,E1,E2}) ->
    “(" ++ print(E1) ++ "+" ++ print(E2) ++ ")";
print({mul,E1,E2}) ->
    “(" ++ print(E1) ++ "*" ++ print(E2) ++ ")".

print({add,{num,2},{mul,{num,3},{num,4}}})
 -> "("++ print({num,2}) ++"+"++ print({mul,{num,3},{num,4}}) ++")"
```

Working it out

```
print({num,N}) ->
    integer_to_list(N);
print({var,A}) ->
    atom_to_list(A);
print({add,E1,E2}) ->
    “(" ++ print(E1) ++ "+" ++ print(E2) ++ ")";
print({mul,E1,E2}) ->
    “(" ++ print(E1) ++ "*" ++ print(E2) ++ ")".

print({add,{num,2},{mul,{num,3},{num,4}}})
 -> "("++ print({num,2}) ++"+"++ print({mul,{num,3},{num,4}}) ++")"
```

Working it out

```
print({num,N}) ->
    integer_to_list(N);
print({var,A}) ->
    atom_to_list(A);
print({add,E1,E2}) ->
    “(" ++ print(E1) ++ "+" ++ print(E2) ++ ")";
print({mul,E1,E2}) ->
    “(" ++ print(E1) ++ "*" ++ print(E2) ++ ")".

print({add,{num,2},{mul,{num,3},{num,4}}})
 -> "("++ print({num,2}) ++"+"++ print({mul,{num,3},{num,4}}) ++")"
 -> "("++"2"++"+"++ "("++ print({num,3}) ++"*"++ print({num,4}) ++”)"++")
```

Working it out

```
print({num,N}) ->
    integer_to_list(N);
print({var,A}) ->
    atom_to_list(A);
print({add,E1,E2}) ->
    “(" ++ print(E1) ++ "+" ++ print(E2) ++ ")";
print({mul,E1,E2}) ->
    “(" ++ print(E1) ++ "*" ++ print(E2) ++ ")".

print({add,{num,2},{mul,{num,3},{num,4}}})
 -> "("++ print({num,2}) ++"+"++ print({mul,{num,3},{num,4}}) ++")"
 -> "("++"2"++"+"++ "("++ print({num,3}) ++"*"++ print({num,4}) ++”)"++")
```

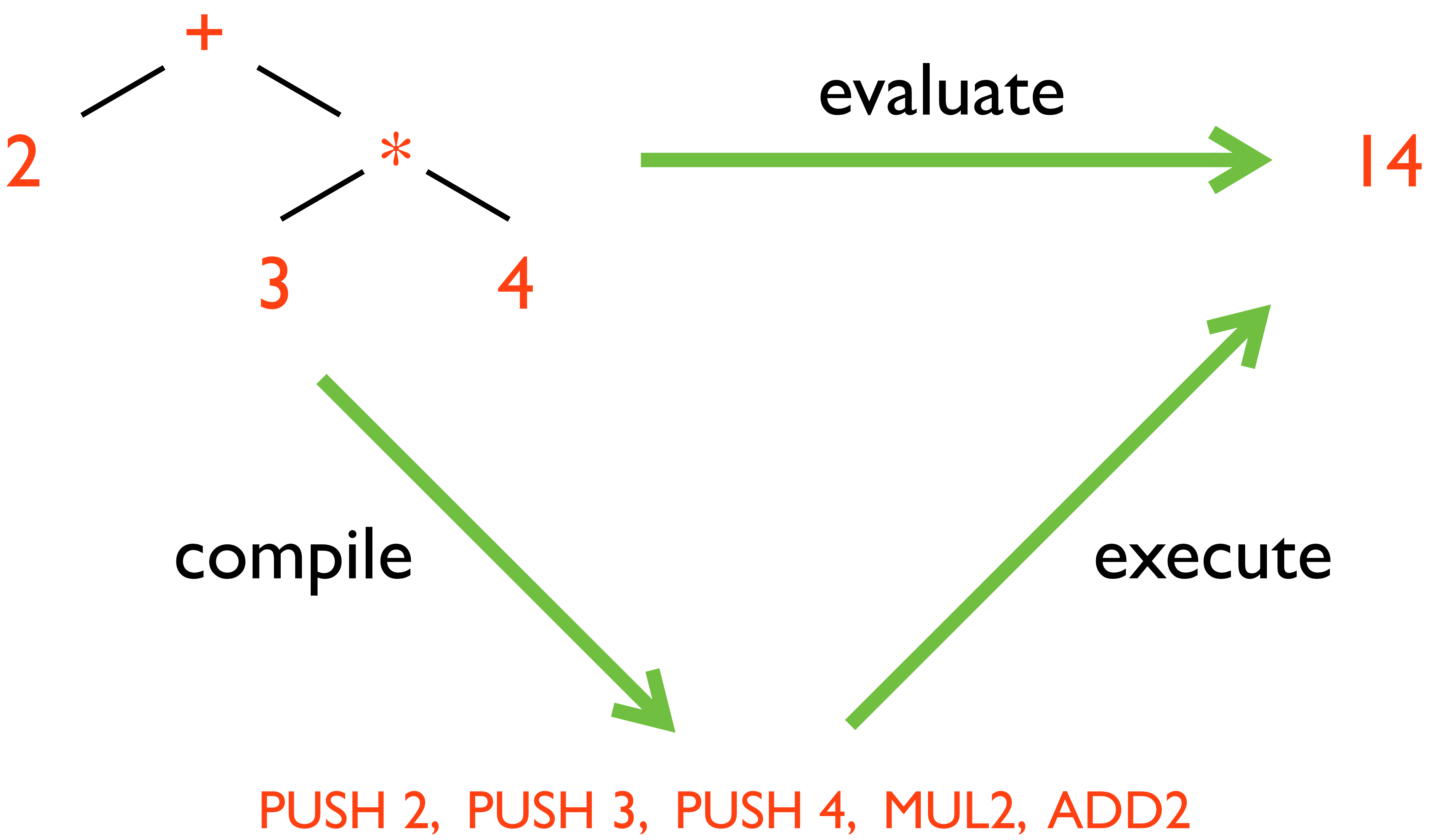
Working it out

```
print({num,N}) ->
    integer_to_list(N);
print({var,A}) ->
    atom_to_list(A);
print({add,E1,E2}) ->
    “(" ++ print(E1) ++ "+" ++ print(E2) ++ ")";
print({mul,E1,E2}) ->
    “(" ++ print(E1) ++ "*" ++ print(E2) ++ ")".

print({add,{num,2},{mul,{num,3},{num,4}}})
 -> "("++ print({num,2}) ++"+"++ print({mul,{num,3},{num,4}}) ++")"
 -> "("++"2"++"+"++ "("++ print({num,3}) ++"*"++ print({num,4}) ++")"++")
 -> "("++"2"++"+"++"("++"3"++"*"++"4"++”)"++")"
```

Working it out

```
print({num,N}) ->
    integer_to_list(N);
print({var,A}) ->
    atom_to_list(A);
print({add,E1,E2}) ->
    “(" ++ print(E1) ++ "+" ++ print(E2) ++ ")";
print({mul,E1,E2}) ->
    “(" ++ print(E1) ++ "*" ++ print(E2) ++ ")".

print({add,{num,2},{mul,{num,3},{num,4}}})
 -> "("++ print({num,2}) ++"+"++ print({mul,{num,3},{num,4}}) ++")"
 -> "("++"2"++"+"++ "("++ print({num,3}) ++"*"++ print({num,4}) ++")"++")
 -> "("++"2"++"+"++"("++"3"++"*"++"4"++”)"++")"
 -> "(2+(3*4))"
```

Section 4: evaluating expressions


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

Defining evaluation

Turn an expression into a number, by working out its value.

```
-spec eval(expr()) -> integer().
```

Defining evaluation

Turn an expression into a number, by working out its value.

```
-spec eval(expr()) -> integer().

 eval({num,N}) ->
     ... ;
 eval({add,E1,E2}) ->
     ... ;
 eval({mul,E1,E2}) ->
     ... .
```

Defining evaluation

Turn an expression into a number, by working out its value.

```
-spec eval(expr()) -> integer().

 eval({num,N}) ->
     N;
 eval({add,E1,E2}) ->
     ... ;
 eval({mul,E1,E2}) ->
     ... .
```

Defining evaluation

Turn an expression into a number, by working out its value.

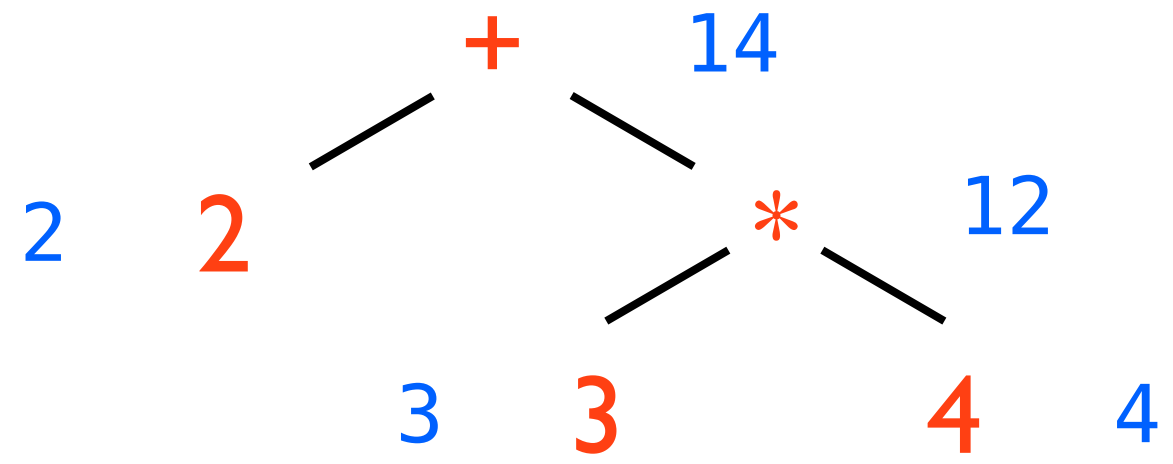
```
-spec eval(expr()) -> integer().

 eval({num,N}) ->
     N;
 eval({add,E1,E2}) ->
     eval(E1) + eval(E2);
 eval({mul,E1,E2}) ->
     ... .
```

Defining evaluation

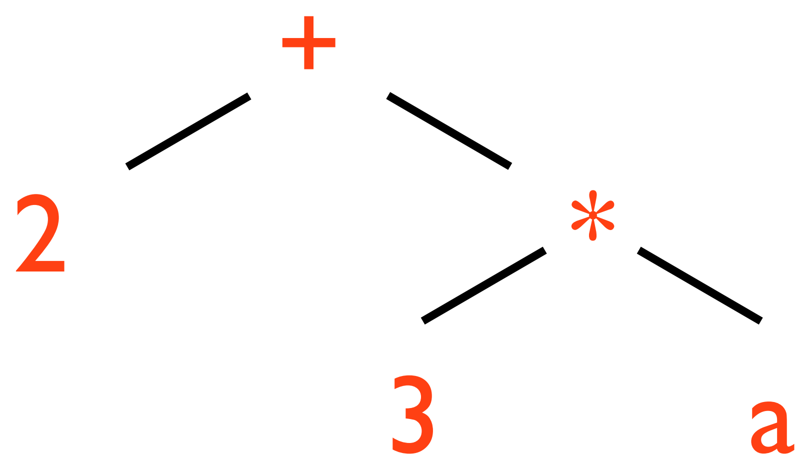
Turn an expression into a number, by working out its value.

```
-spec eval(expr()) -> integer().

 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



Evaluation with variables

We need somewhere to `lookup` their values, e.g. `[{a,23},{v,17}]`

```
-type env() :: [{atom(),integer()}].

-spec eval(env(),expr()) -> integer().

eval(_Env,{num,N}) -> N;

eval(Env,{var,A}) ->
    lookup(A,Env);

eval(Env,{add,E1,E2}) ->
    eval(Env,E1) + eval(Env,E2);

eval(Env,{mul,E1,E2}) ->
    eval(Env,E1) * eval(Env,E2).
```

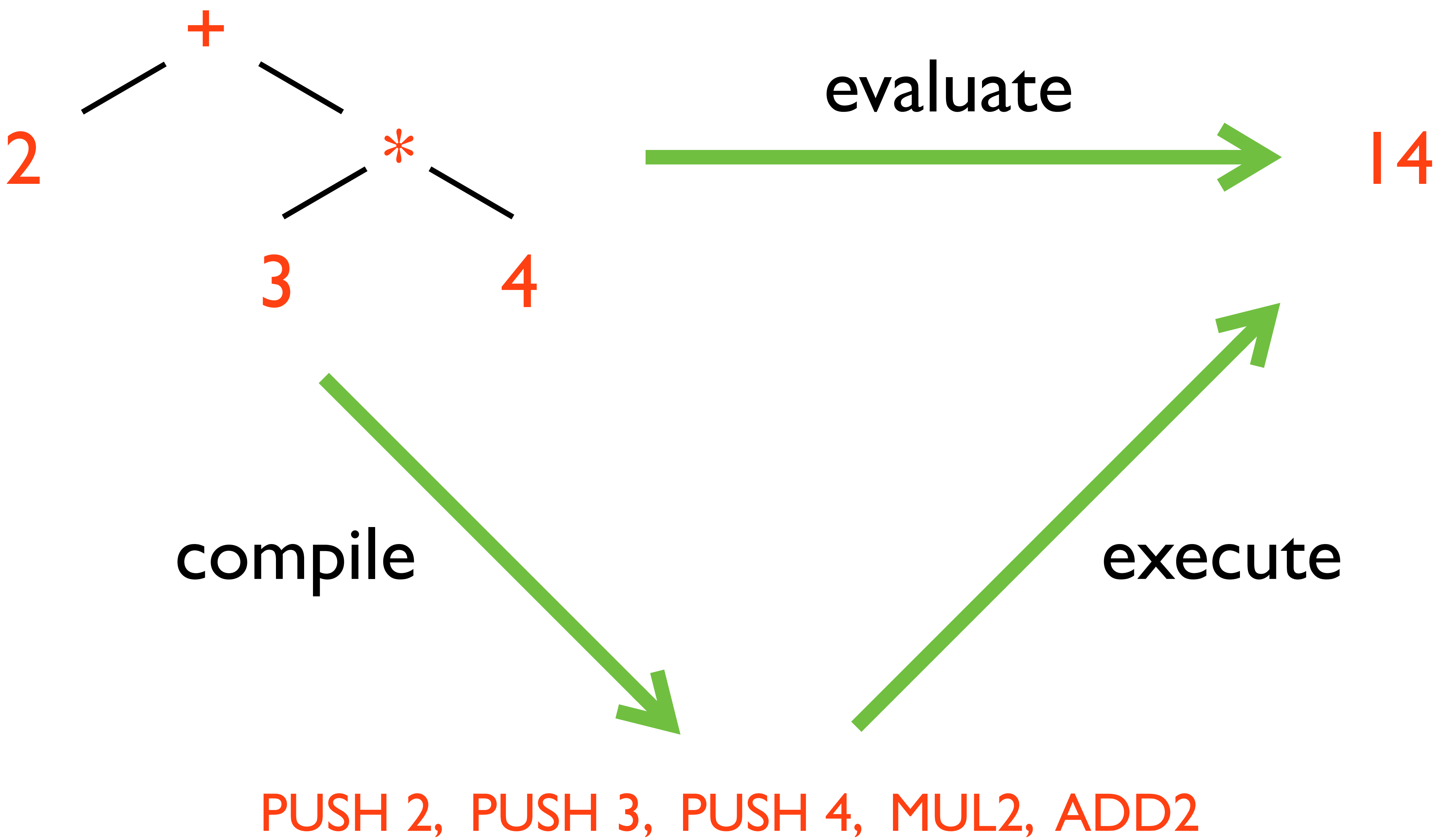
Evaluation with variables

We need somewhere to `lookup` their values, e.g. `[{a,23},{v,17}]`.

```
-type env() :: [{atom(),integer()}].

-spec lookup(atom(),env()) -> integer().

lookup(A,[{A,V}|_]) ->
    V;
lookup(A,[_|Rest]) ->
    lookup(A,Rest).
```

Fails when an atom not present e.g. `lookup(foo,[{a,23},{v,17}])`.

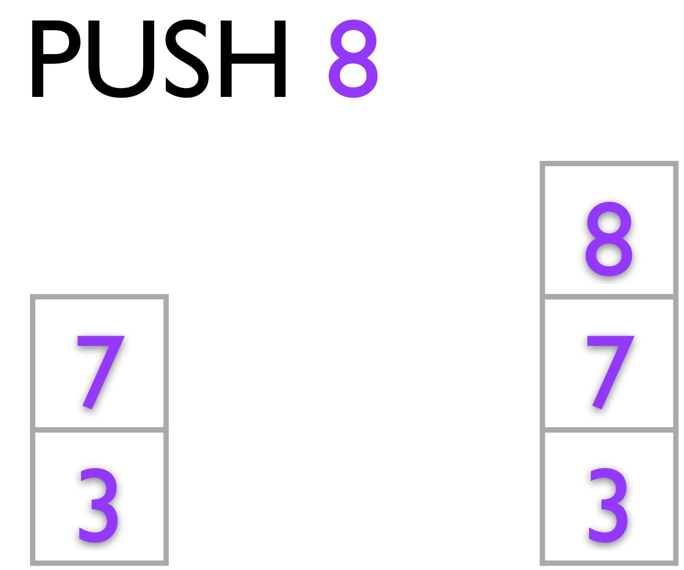
Section 5: compiling and running on a VM


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

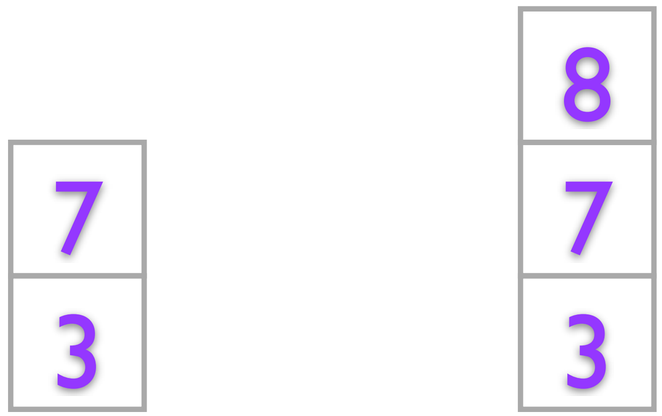
A stack virtual machine

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

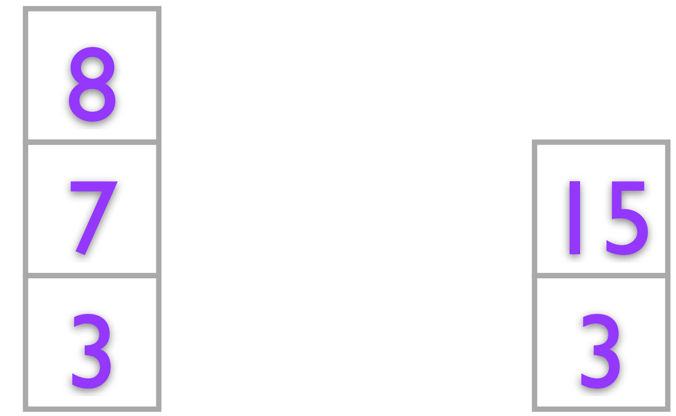
For example, the **PUSH N** instruction pushes the integer **N** onto the top of the stack.



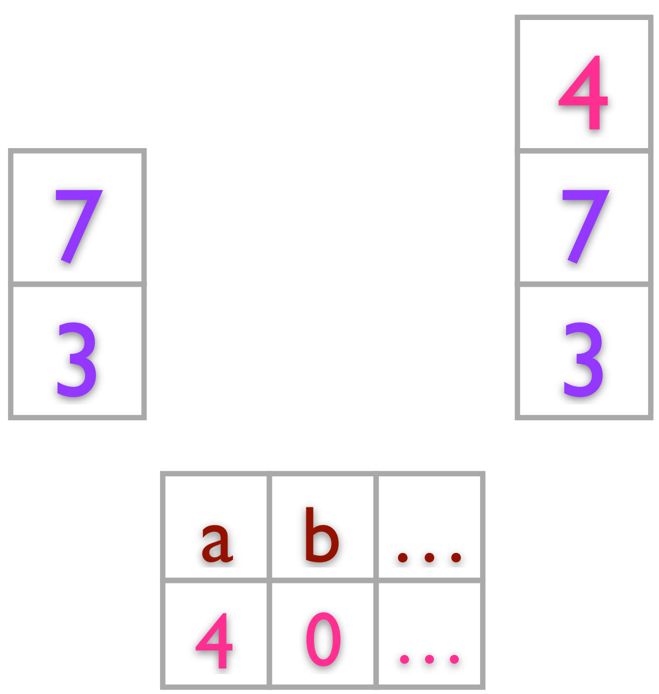
PUSH 8



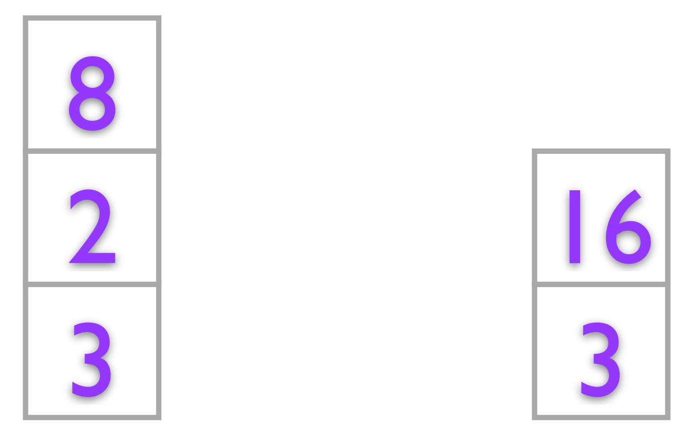
ADD$_2$



FETCH a



MUL$_2$

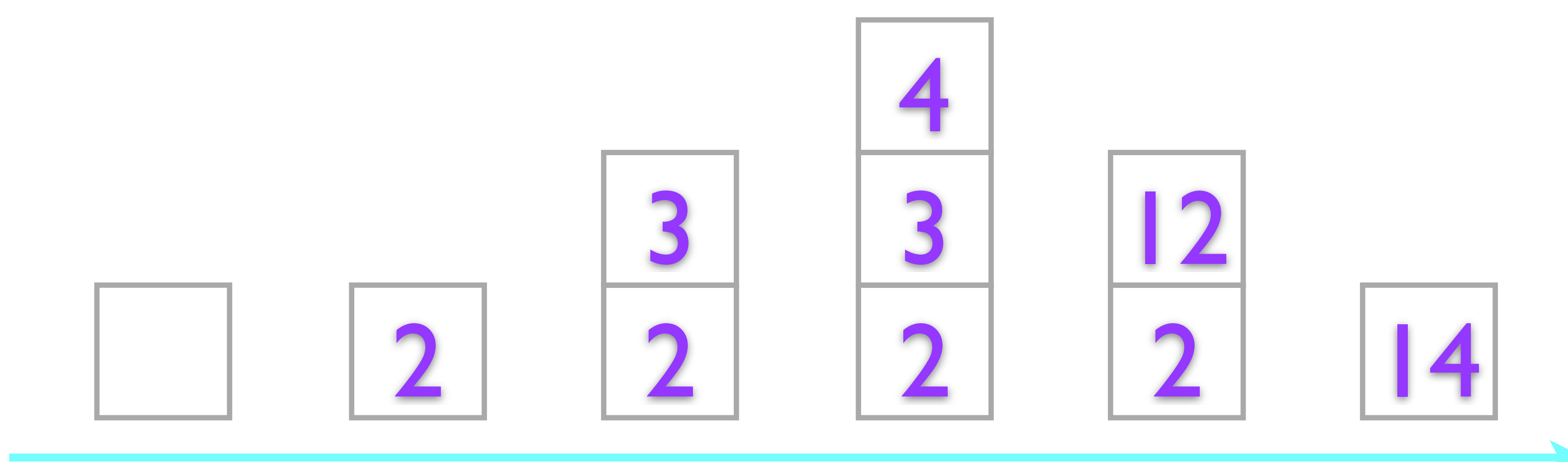


A stack virtual machine

(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 Erlang

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

- 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

```
-type instr() :: {'push',integer()}
              |  {'fetch',atom()}
              |  {'add2'}
              |  {‘mul2'}.

-type program() :: [instr()].

-type stack() :: [integer()].
```

Compiling expressions; running programs

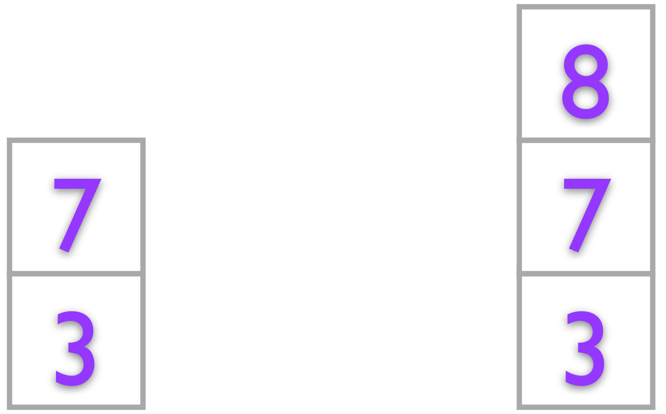
```
-spec compile(expr()) -> program().

-spec run(program(),env()) -> integer().
```

Compiling expressions; running programs

```
-spec compile(expr()) -> program().

-spec run(program(),env()) -> integer().

-spec run(program(),env(),stack()) -> integer().
```

Running the stack machine

PUSH 8

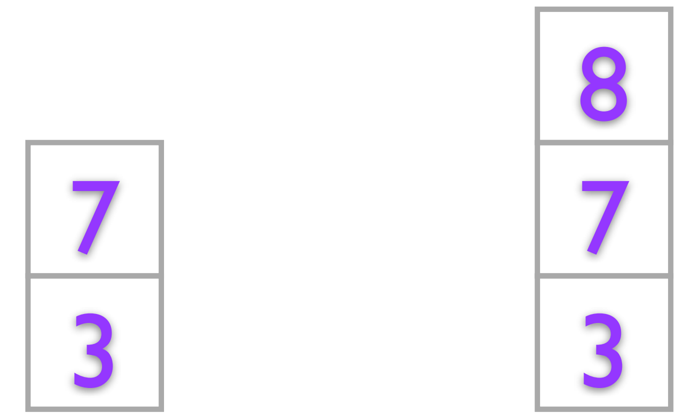


The stack machine `run` function is defined using pattern matching and tail recursion.

```
run([{push, N} | Continue], Env, Stack) ->
   … ;

run([{fetch, A} | Continue], Env, Stack) ->
   … ;

run([{add2} | Continue], Env, [N1,N2|Stack]) ->
   … ;

run([{mul2} | Continue], Env ,[N1,N2|Stack]) ->
   … ;

run([],_Env,[N]) -> … .
```

Running the stack machine

PUSH 8

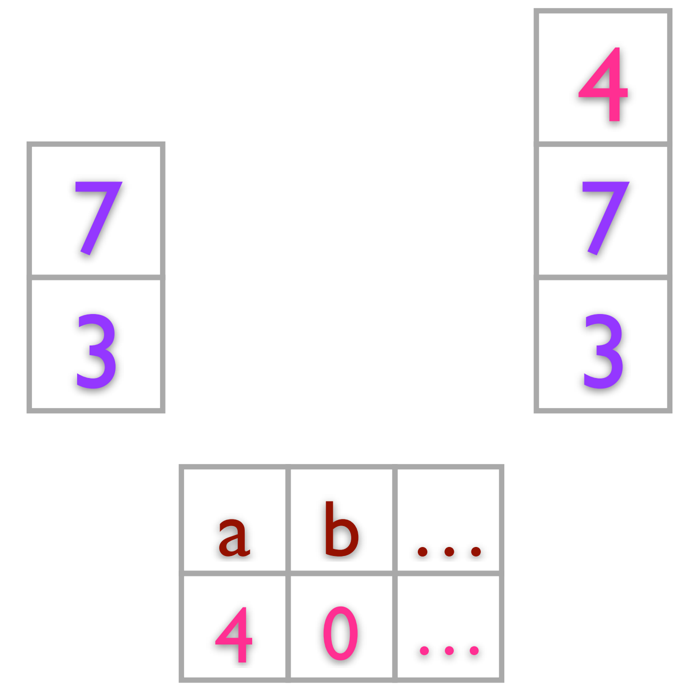


The stack machine `run` function is defined using pattern matching and tail recursion.

```
run([{push, N} | Continue], Env, Stack) ->
   run(Continue, Env, [N | Stack]);

run([{fetch, A} | Continue], Env, Stack) ->
   … ;

run([{add2} | Continue], Env, [N1,N2|Stack]) ->
   … ;

run([{mul2} | Continue], Env ,[N1,N2|Stack]) ->
   … ;

run([],_Env,[N]) -> … .
```

Running the stack machine

FETCH a

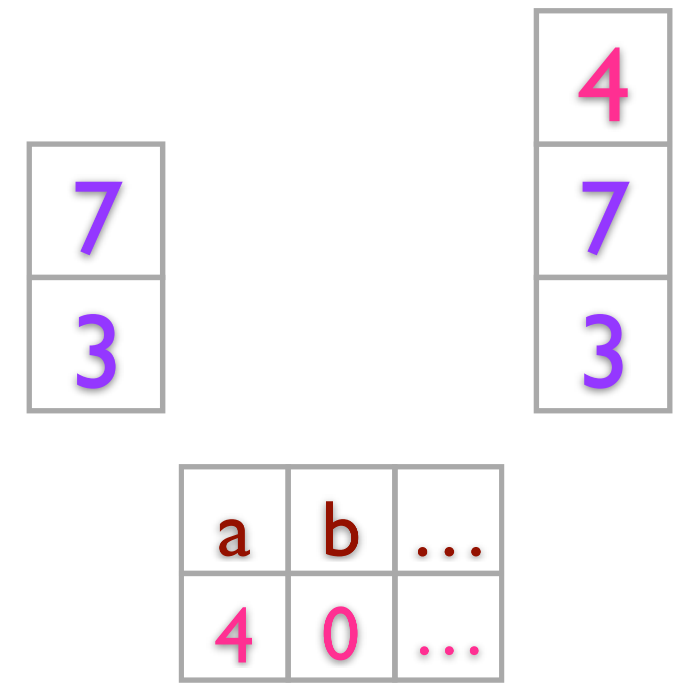


The stack machine `run` function is defined using pattern matching and tail recursion.

```
run([{push, N} | Continue], Env, Stack) ->
   run(Continue, Env, [N | Stack]);

run([{fetch, A} | Continue], Env, Stack) ->
   … ;

run([{add2} | Continue], Env, [N1,N2|Stack]) ->
   … ;

run([{mul2} | Continue], Env ,[N1,N2|Stack]) ->
   … ;

run([],_Env,[N]) -> … .
```

Running the stack machine

FETCH a

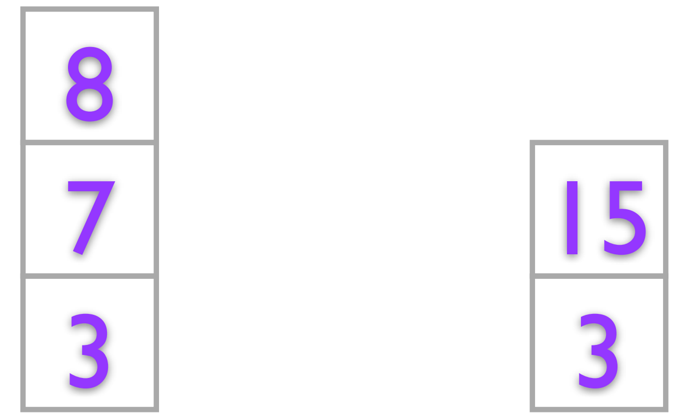


The stack machine `run` function is defined using pattern matching and tail recursion.

```
run([{push, N} | Continue], Env, Stack) ->
   run(Continue, Env, [N | Stack]);

run([{fetch, A} | Continue], Env, Stack) ->
   run(Continue, Env, [lookup(A,Env) | Stack]);

run([{add2} | Continue], Env, [N1,N2|Stack]) ->
   … ;

run([{mul2} | Continue], Env ,[N1,N2|Stack]) ->
   … ;

run([],_Env,[N]) -> … .
```

Running the stack machine

ADD2

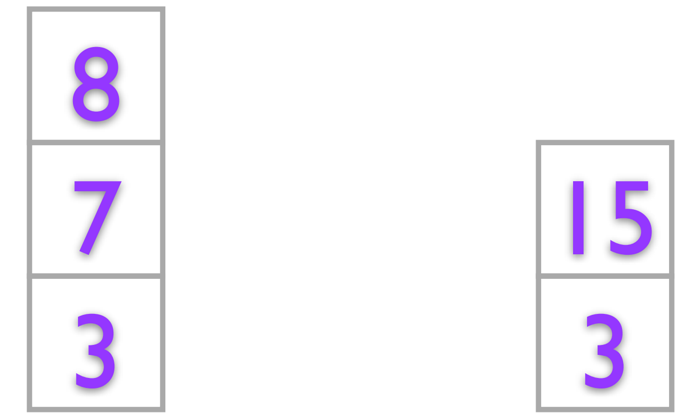


The stack machine `run` function is defined using pattern matching and tail recursion.

```
run([{push, N} | Continue], Env, Stack) ->
   run(Continue, Env, [N | Stack]);

run([{fetch, A} | Continue], Env, Stack) ->
   run(Continue, Env, [lookup(A,Env) | Stack]);

run([{add2} | Continue], Env, [N1,N2|Stack]) ->
   … ;

run([{mul2} | Continue], Env ,[N1,N2|Stack]) ->
   … ;

run([],_Env,[N]) -> … .
```

Running the stack machine

ADD2

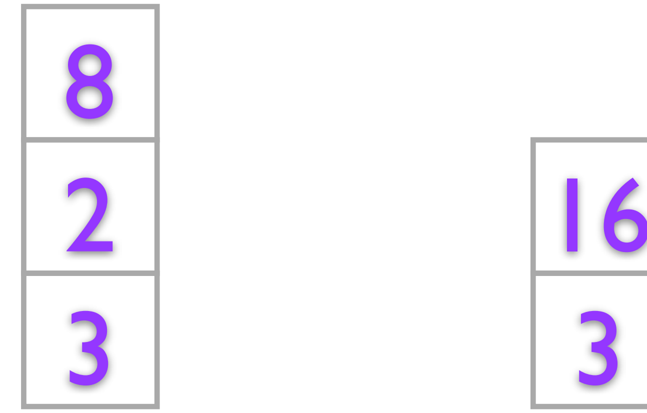


The stack machine `run` function is defined using pattern matching and tail recursion.

```
run([{push, N} | Continue], Env, Stack) ->
   run(Continue, Env, [N | Stack]);

run([{fetch, A} | Continue], Env, Stack) ->
   run(Continue, Env, [lookup(A,Env) | Stack]);

run([{add2} | Continue], Env, [N1,N2|Stack]) ->
   run(Continue, Env, [(N1+N2) | Stack]);

run([{mul2} | Continue], Env ,[N1,N2|Stack]) ->
   … ;

run([],_Env,[N]) -> … .
```

Running the stack machine

MUL$_2$

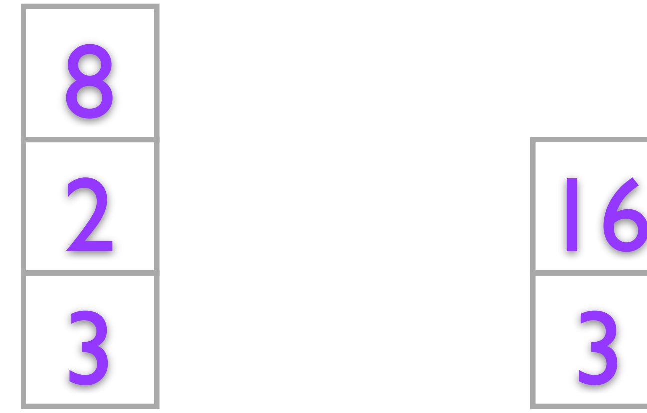


The stack machine run function is defined using pattern matching and tail recursion.

```
run([{push, N} | Continue], Env, Stack) ->
   run(Continue, Env, [N | Stack]);

run([{fetch, A} | Continue], Env, Stack) ->
   run(Continue, Env, [lookup(A,Env) | Stack]);

run([{add2} | Continue], Env, [N1,N2|Stack]) ->
   run(Continue, Env, [(N1+N2) | Stack]);

run([{mul2} | Continue], Env ,[N1,N2|Stack]) ->
   … ;

run([],_Env,[N]) -> … .
```

Running the stack machine

MUL$_2$



The stack machine `run` function is defined using pattern matching and tail recursion.

```
run([{push, N} | Continue], Env, Stack) ->
   run(Continue, Env, [N | Stack]);

run([{fetch, A} | Continue], Env, Stack) ->
   run(Continue, Env, [lookup(A,Env) | Stack]);

run([{add2} | Continue], Env, [N1,N2|Stack]) ->
   run(Continue, Env, [(N1+N2) | Stack]);

run([{mul2} | Continue], Env ,[N1,N2|Stack]) ->
   run(Continue, Env, [(N1*N2) | Stack]);

run([],_Env,[N]) -> … .
```

Running the stack machine

The stack machine `run` function is defined using pattern matching and tail recursion.

```
run([{push, N} | Continue], Env, Stack) ->
   run(Continue, Env, [N | Stack]);

run([{fetch, A} | Continue], Env, Stack) ->
   run(Continue, Env, [lookup(A,Env) | Stack]);

run([{add2} | Continue], Env, [N1,N2|Stack]) ->
   run(Continue, Env, [(N1+N2) | Stack]);

run([{mul2} | Continue], Env ,[N1,N2|Stack]) ->
   run(Continue, Env, [(N1*N2) | Stack]);

run([],_Env,[N]) -> … .
```

Running the stack machine

The stack machine `run` function is defined using pattern matching and tail recursion.

```
run([{push, N} | Continue], Env, Stack) ->
   run(Continue, Env, [N | Stack]);

run([{fetch, A} | Continue], Env, Stack) ->
   run(Continue, Env, [lookup(A,Env) | Stack]);

run([{add2} | Continue], Env, [N1,N2|Stack]) ->
   run(Continue, Env, [(N1+N2) | Stack]);

run([{mul2} | Continue], Env ,[N1,N2|Stack]) ->
   run(Continue, Env, [(N1*N2) | Stack]);

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.

```
-spec compile(expr()) -> program().

compile({num,N}) ->
    [{push, N}];

compile({var,A}) ->
    [{fetch, A}];

compile({add,E1,E2}) ->
    compile(E1) ++ compile(E2) ++ [{add2}];

compile({mul,E1,E2}) ->
    compile(E1) ++ compile(E2) ++ [{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

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.

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

```
-spec parse(string()) -> {expr(), string()}.
```

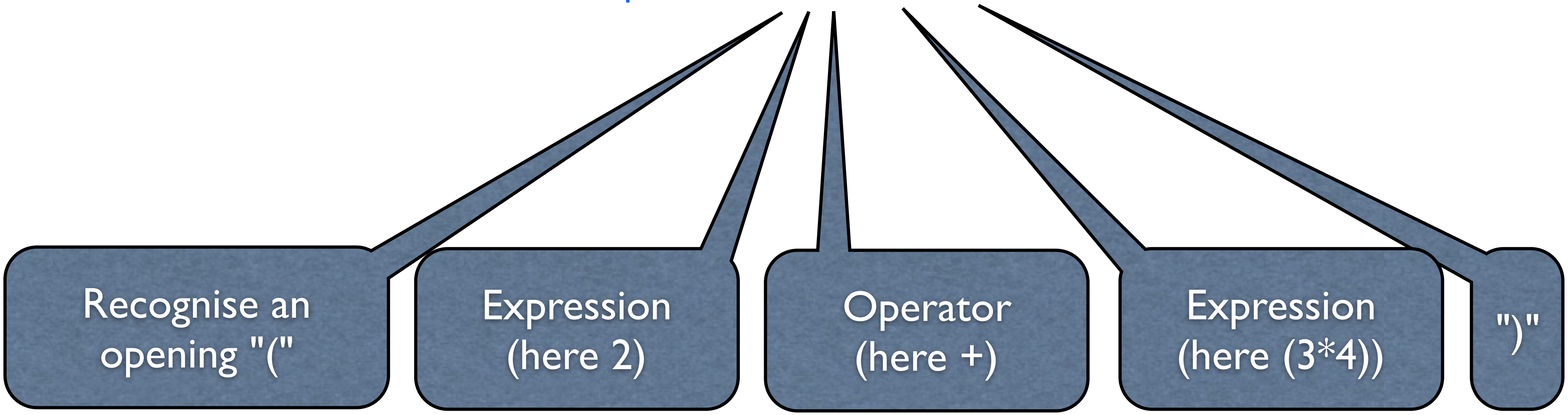
Examples

```
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

Recognise "(", an expression, an operator, an expression and finally ")".

```
-spec parse(string()) -> {expr(), string()}.

parse([$(|Rest]) ->                                 % starts with a '('
      {E1,Rest1}     = parse(Rest),                 % then an expression
      [Op|Rest2]     = Rest1,                       % then an operator, '+' or '*'
      {E2,Rest3}     = parse(Rest2),                % then another expression
      [$)|RestFinal] = Rest3,                       % starts with a ')'
      {case Op of
           $+ -> {add,E1,E2};
           $* -> {mul,E1,E2}
        end,
       RestFinal};
```

Predictive, top-down, parsing

Recognise "(", an expression, an operator, an expression and finally ")".

```
-spec parse(string()) -> {expr(), string()}.

parse([$(|Rest]) ->                                   % starts with a '('
      {E1,Rest1}     = parse(Rest),                   % then an expression
      [Op|Rest2]     = Rest1,                         % then an operator, '+' or '*'
      {E2,Rest3}     = parse(Rest2),                  % then another expression
      [$)|RestFinal] = Rest3,                         % starts with a ')'
      {case Op of
           $+ -> {add,E1,E2};
           $* -> {mul,E1,E2}
        end,
       RestFinal};
```

Predictive, top-down, parsing

Recognise "(", an expression, an operator, an expression and finally ")".

```
-spec parse(string()) -> {expr(), string()}.

parse([$(|Rest]) ->                                    % starts with a '('
      {E1,Rest1}     = parse(Rest),                    % then an expression
      [Op|Rest2]     = Rest1,                          % then an operator, '+' or '*'
      {E2,Rest3}     = parse(Rest2),                   % then another expression
      [$)|RestFinal] = Rest3,                          % starts with a ')'
      {case Op of
           $+ -> {add,E1,E2};
           $* -> {mul,E1,E2}
        end,
       RestFinal};
```

Predictive, top-down, parsing

Recognise `"("`, an expression, an operator, an expression and finally `")"`.

```
-spec parse(string()) -> {expr(), string()}.

parse([$(|Rest]) ->                                % starts with a '('
      {E1,Rest1}     = parse(Rest),                % then an expression
      [Op|Rest2]     = Rest1,                      % then an operator, '+' or '*'
      {E2,Rest3}     = parse(Rest2),               % then another expression
      [$)|RestFinal] = Rest3,                      % starts with a ')'
      {case Op of
           $+ -> {add,E1,E2};
           $* -> {mul,E1,E2}
        end,
       RestFinal};
```

Predictive, top-down, parsing

Recognise `"("`, an expression, an operator, an expression and finally `")"`.

```
-spec parse(string()) -> {expr(), string()}.

parse([$(|Rest]) ->                                 % starts with a '('
      {E1,Rest1}     = parse(Rest),                 % then an expression
      [Op|Rest2]     = Rest1,                       % then an operator, '+' or '*'
      {E2,Rest3}     = parse(Rest2),                % then another expression
      [$)|RestFinal] = Rest3,                       % starts with a ')'
      {case Op of
           $+ -> {add,E1,E2};
           $* -> {mul,E1,E2}
        end,
       RestFinal};
```

Predictive, top-down, parsing

Recognise "(", an expression, an operator, an expression and finally ")".

```
-spec parse(string()) -> {expr(), string()}.

parse([$(|Rest]) ->                                    % starts with a '('
      {E1,Rest1}     = parse(Rest),                    % then an expression
      [Op|Rest2]     = Rest1,                          % then an operator, '+' or '*'
      {E2,Rest3}     = parse(Rest2),                   % then another expression
      [$)|RestFinal] = Rest3,                          % starts with a ')'
      {case Op of
           $+ -> {add,E1,E2};
           $* -> {mul,E1,E2}
        end,
       RestFinal};
```

Predictive, top-down, parsing

Recognise `"("`, an expression, an operator, an expression and finally `")"`.

```
-spec parse(string()) -> {expr(), string()}.

parse([$(|Rest]) ->                                % starts with a '('
      {E1,Rest1}     = parse(Rest),                % then an expression
      [Op|Rest2]     = Rest1,                      % then an operator, '+' or '*'
      {E2,Rest3}     = parse(Rest2),               % then another expression
      [$)|RestFinal] = Rest3,                      % starts with a ')'
      {case Op of
           $+ -> {add,E1,E2};
           $* -> {mul,E1,E2}
        end,
       RestFinal};
```

Recognising numbers and literals

Examples

```
parse(“-123)”) =  {{num,-123},”)”}

parse("variable+3") = {{var,variable},”+3”}

parse("a") = {var,a}
```

In both cases we want to recognise the longest sequence of digits or characters …

Recognising numbers and literals

Get the longest initial segment of a list with a given property:

```
-spec get_while(fun((T) -> boolean()),[T]) -> {[T],[T]}.
```

Recognising numbers and literals

Get the longest initial segment of a list with a given property:

```
-spec get_while(fun((T) -> boolean()),[T]) -> {[T],[T]}.
```

Recognising numbers and literals

Get the longest initial segment of a list with a given property:

```
-spec get_while(fun((T) -> boolean()),[T]) -> {[T],[T]}.
```

where we represent a property as a boolean-valued function.

Recognising numbers and literals

```
get_while(P,[Ch|Rest]) ->
  case P(Ch) of
    true ->
      {Succeeds,Remainder} = get_while(P,Rest),
      {[Ch|Succeeds],Remainder};
    false ->
      {[],[Ch|Rest]}
  end;

get_while(_P,[]) ->
  {[],[]}.
```

Recognising literals

Literals

```
parse([Ch|Rest])  when $a =< Ch andalso Ch =< $z ->
    {Succeeds,Remainder} = get_while(fun is_alpha/1,Rest),
    {{var, list_to_atom([Ch|Succeeds])}, Remainder}.
```

Testing for a small alphabetic character …

```
is_alpha(Ch) -> $a =< Ch andalso Ch =< $z.
```