Matt Might
University of Utah
matt.might.net

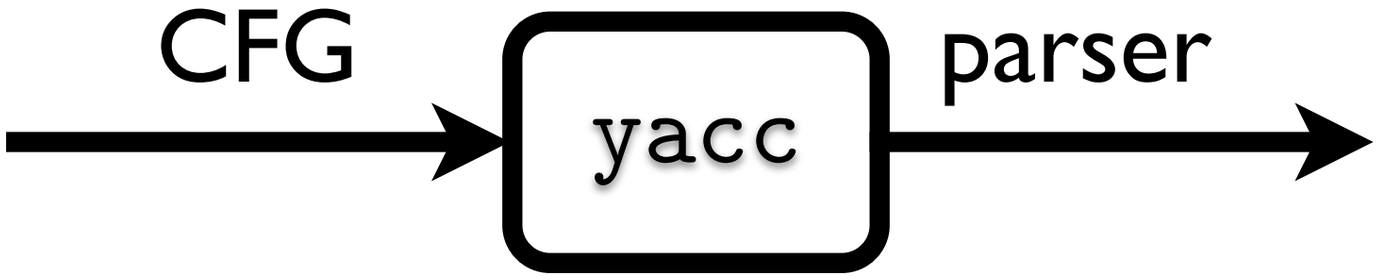
yacc

Reminder!

- Midterm next Thursday
- Midterm review Tuesday
- “Best” of P1 - Tuesday

Project 2: Out

Parser generators


CFG
yacc
parser

yacc

bison

```
%{
prologue code
%}

bison declarations
%%
grammar rules
%%

epilogue code
```

Rule format

$$
\begin{aligned}
&\mathit{nonterm}\ :\ \mathit{exp}_1\ \ldots\ \mathit{exp}_n\ \{\ \mathit{action}\ \}\\
&\quad |\ \mathit{exp}_1\ \ldots\ \mathit{exp}_m\ \{\ \mathit{action}\ \}\\
&\quad \ldots\\
&\quad ;
\end{aligned}
$$

```
<sexp>   ::= "(" <sexpl> ")"
           |  SYMBOL
           |  INTEGER
           |  #t
           |  #f

<sexpl>  ::= <sexp> <sexpl>
           |
```

```
start : sexp
      ;

sexp : INT
     | SYM
     | 't'
     | 'f'
     | '(' sexpl ')'
     ;

sexpl : sexp sexpl
      |
      ;
```

```
start : sexp { result = $1 ; }
      ;

sexp : INT { $$ = integer($1) ; }
     | SYM { $$ = symbol($1) ; }
     | 't' { $$ = SX_TRUE ; }
     | 'f' { $$ = SX_FALSE ; }
     | '(' sexpl ')' { $$ = $2 ; }
     ;

sexpl : sexp sexpl { $$ = cons($1, $2) ; }
      |            { $$ = SX_NIL ; }
      ;
```

```
start : exp { printf("ans: %i\n", $1 ); }
      ;

exp : exp '+' term { $$ = $1 + $3 ; }
    | term         { $$ = $1 ; }
    ;

term : term '*' factor { $$ = $1 * $3 ; }
     | factor          { $$ = $1 ; }
     ;

factor : INT { $$ = $1 ; }
       | SYM { $$ = lookup($1) ; }
       | '(' exp ')' { $$ = $2 ; }
       ;
```

lex & yacc

```
bison -d file.y
```

file.tab.h

```
%token id
```

%token EQUALS

```
%token <field> id
```

```
%union {
    type field ;
    . . .
}
```

```
%union {
  char* id ;
    int    z ;
}
```

```
%token <z> NUM
```

```
%token <id> ID
```

```
%type <field> nt ...
```

```
%type <z> exp term
```

```
start : exp { printf("ans: %i\n", $1 ); }
      ;

exp : exp '+' term { $$ = $1 + $3 ; }
    | term         { $$ = $1 ; }
    ;

term : term '*' factor { $$ = $1 * $3 ; }
     | factor          { $$ = $1 ; }
     ;

factor : INT { $$ = $1 ; }
       | SYM { $$ = lookup($1) ; }
       | '(' exp ')' { $$ = $2 ; }
       ;
```

```
%union {
  int    integer ;
  char* symbol ;
}

%type <integer> exp term factor

%token <integer> INT
%token <symbol>  SYM
```

```
%union {
  int   integer ;
  char* symbol ;
}

%type <integer> exp term factor

%token <integer> INT
%token <symbol>  SYM

%%

start : exp { printf("ans: %i\n", $1 ); }
      ;

exp : exp '+' term { $$ = $1 + $3 ; }
    | term         { $$ = $1 ; }
    ;

term : term '*' factor { $$ = $1 * $3 ; }
     | factor          { $$ = $1 ; }
     ;

factor : INT { $$ = $1 ; }
       | SYM { $$ = lookup($1) ; }
       | '(' exp ')' { $$ = $2 ; }
       ;
```

exp.l

```
%{

#include <stdio.h>
#include <string.h>

#include "exp.tab.h"

%}
```

exp.tab.h

exp.tab.h

```
/* Tokens.  */
#define INT 258
#define SYM 259
```

exp.tab.h

```
typedef union YYSTYPE
{
  int   integer ;
  char* symbol ;
}

extern YYSTYPE yylval;
```

```
%option noyywrap

ws [ \t\r\n]

id [A-Za-z_][A-Za-z_0-9]*

nzd    [1-9]
digit  [0-9]

decint -?{nzd}{digit}*|0+
```

```
%%

<INITIAL>{ws}          {}
<INITIAL>"("           {return '(';}
<INITIAL>")"           {return ')';}
<INITIAL>"*"           {return '*';}
<INITIAL>"+"           {return '+';}
<INITIAL>{decint}      {yylval.integer = atoi(yytext);
                        return INT;}
<INITIAL>{id}          {yylval.symbol = strdup(yytext);
                        return SYM;}
<INITIAL><<EOF>>       {return 0;}

.   {printf("error: unknown char: '%c'\n",*yytext);
     exit(-1);}

%%
```

Can we simplify grammar?

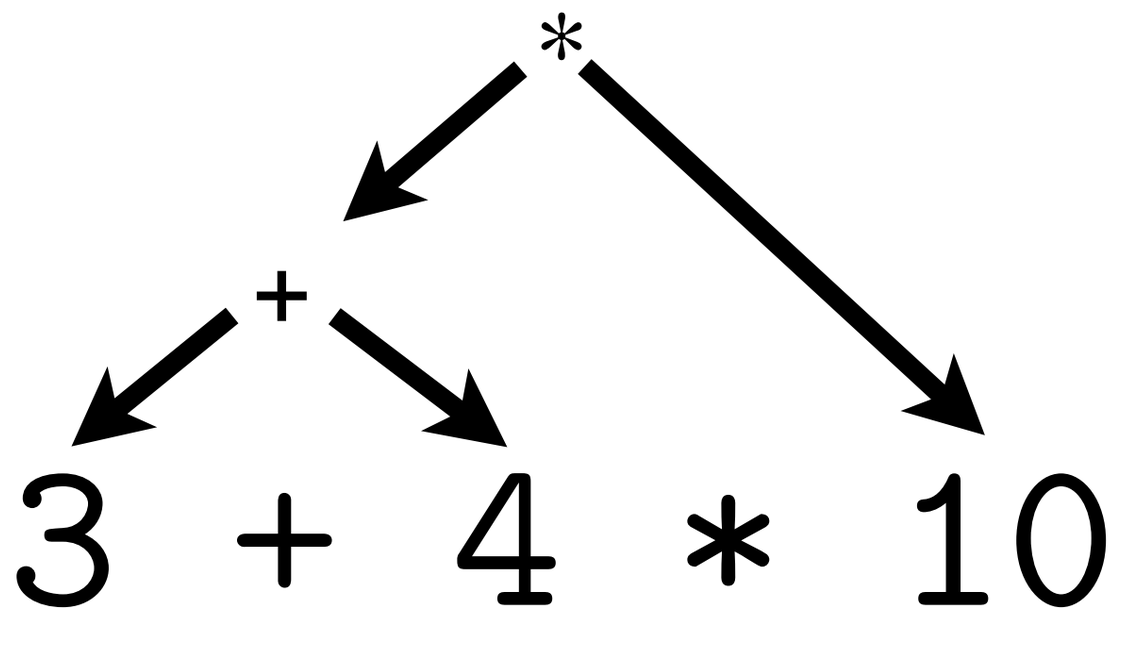
```
<exp> ::= <exp> ‘*’ <exp>
        |  <exp> ‘+’ <exp>
        |  ‘(’ <exp> ‘)’
        |  <id>
        |  <num>
```

```
start : exp { printf("ans: %i\n", $1 ) ; }
      ;

exp : exp '+' exp { $$ = $1 + $3 ; }
| exp '*' exp { $$ = $1 * $3 ; }
| INT { $$ = $1 ; }
| SYM { $$ = lookup($1) ; }
| '(' exp ')' { $$ = $2 ; }
;
```

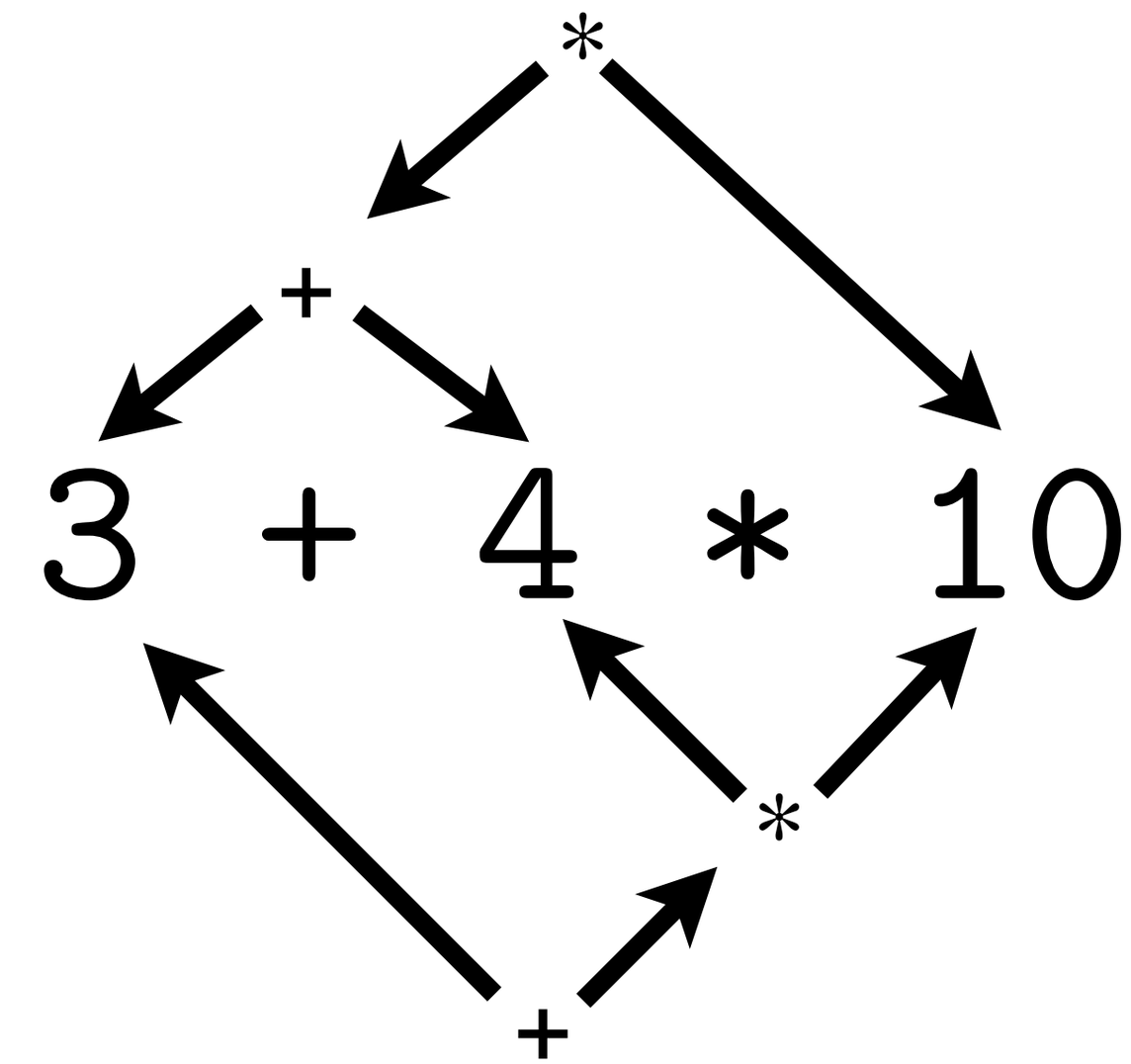
Ambiguous grammar!

```
3 + 4 * 10
```

(3 + 4) * 10



```
(3 + 4) * 10
```




```
3 + (4 * 10)
```

If the same string has multiple parse trees, the grammar is ambiguous.

```
% bison -d exp.y
exp.y: conflicts: 4 shift/reduce
```

```
% wc -l exp.output
189 exp.output
```

exp.output

```
State 11 conflicts: 2 shift/reduce
State 12 conflicts: 2 shift/reduce
...
```

```
state 11

    2 exp: exp . '+' exp  [$end, '+', '*', ')']
    2    | exp '+' exp .  [$end, '+', '*', ')']
    3    | exp . '*' exp

'+'  shift, and go to state 8
'*'  shift, and go to state 9

'+'        [reduce using rule 2 (exp)]
'*'        [reduce using rule 2 (exp)]
$default  reduce using rule 2 (exp)
```

```
%left '+'
%left '*'
```

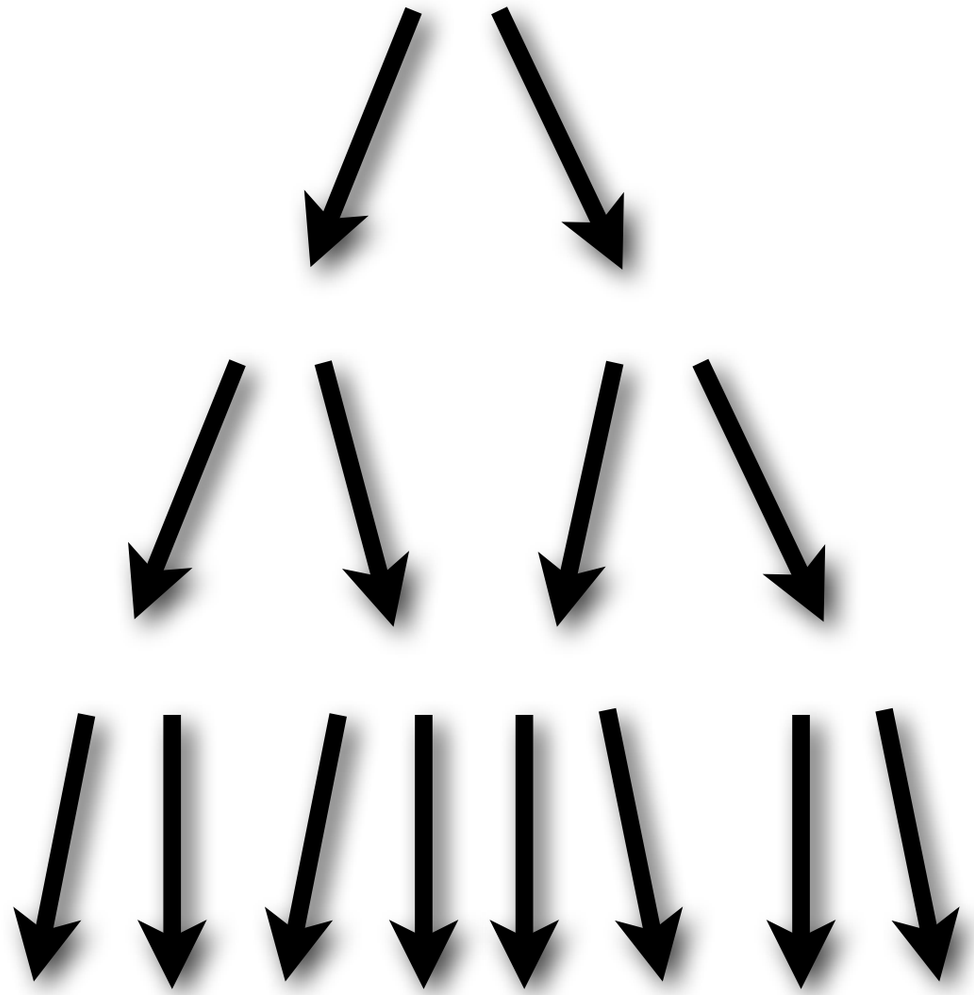
Ugh...

How yacc thinks

Top-down *v.* bottom-up

Top-down

Top-down



Top-down drawbacks

- No left-recursive grammars
- Explosive growth in lookahead
- Need lots of refactoring

```
<E> ::= <T> + <E>
      |  <T>

<T> ::= <F> * <T>
      |  <F>

<F> ::= ( <E> )
      |  NUM
```

```
<E> ::= <E> + <T>
      |  <T>

<T> ::= <T> * <F>
      |  <F>

<F> ::= ( <E> )
      |  NUM
```

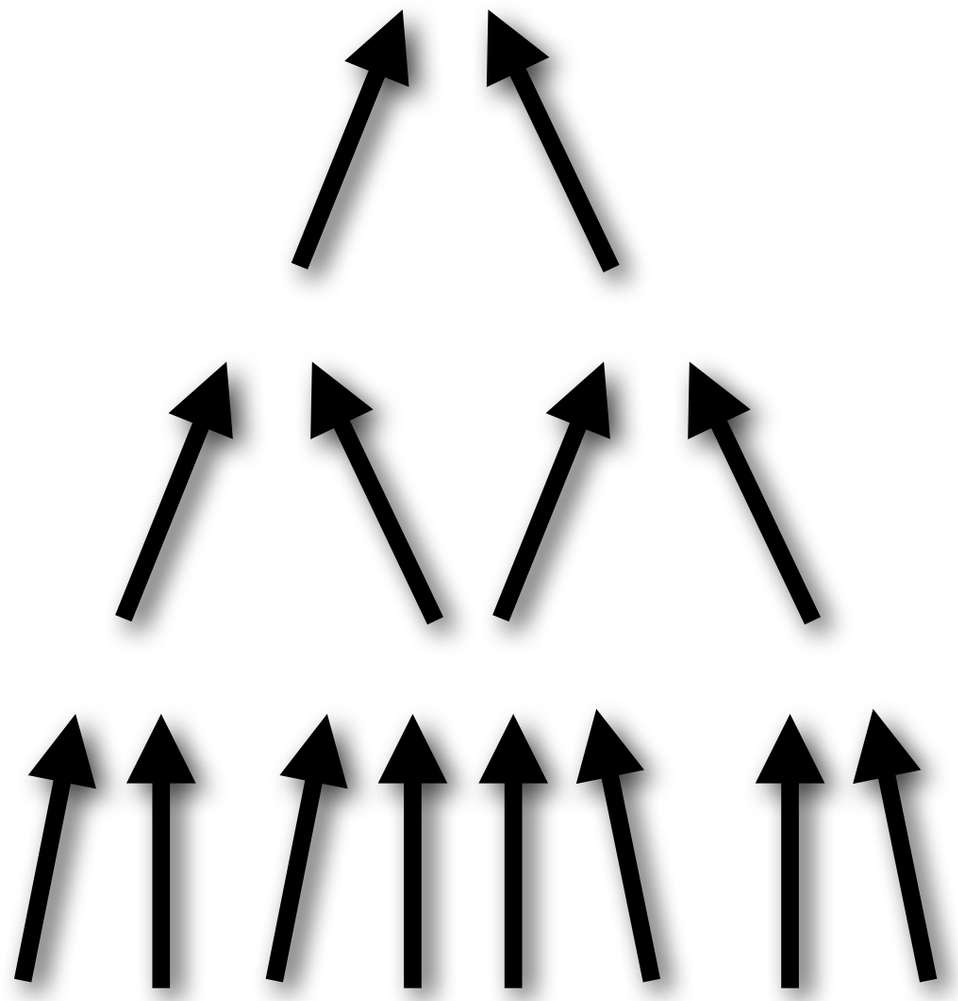
```
<E> ::= <T> + <E>
      |  <T>

<T> ::= <F> * <T>
      |  <F>

<F> ::= ( <E> )
      |  NUM
```

```
<E> ::= <E> + <T>
      |  <T>

<T> ::= <T> * <F>
      |  <F>

<F> ::= ( <E> )
      |  NUM
```

Bottom-up parsing

Bottom-up

Bottom-up



Bottom-up example

$$(3 + 7) * 4$$

$$(F + 7) * 4$$

$$(F + F) * 4$$

$$(T + T) * 4$$

$$(T + E) * 4$$

$$(E) * 4$$

$$F * 4$$

$$F * F$$

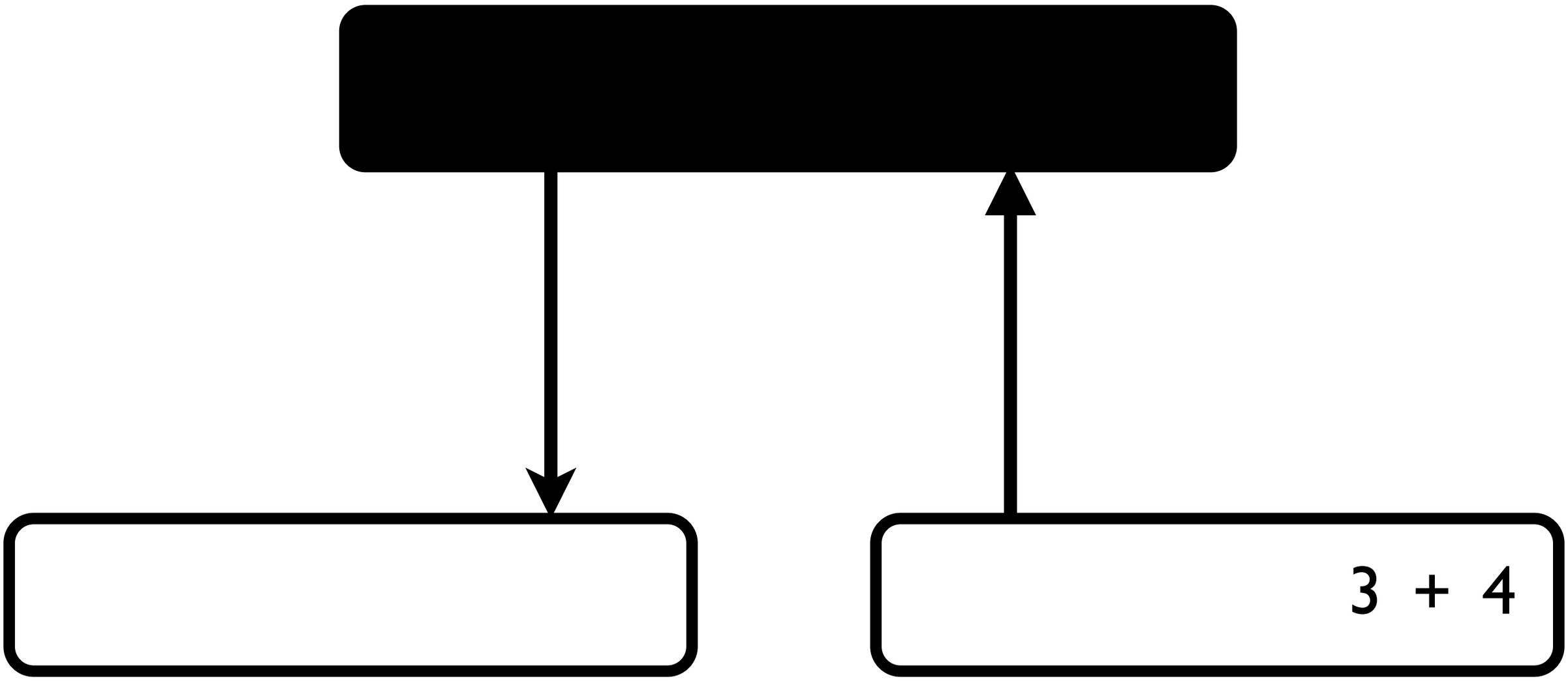
$$F * T$$

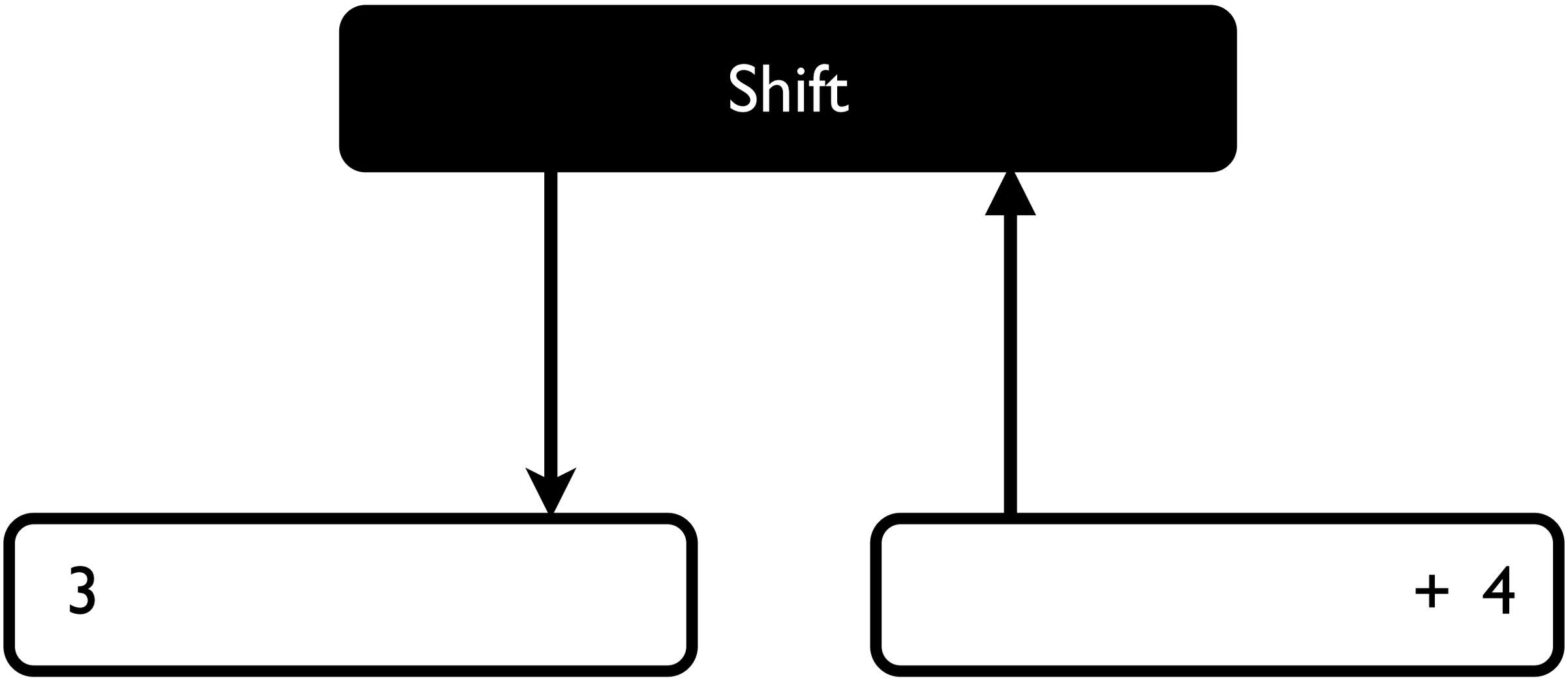
T

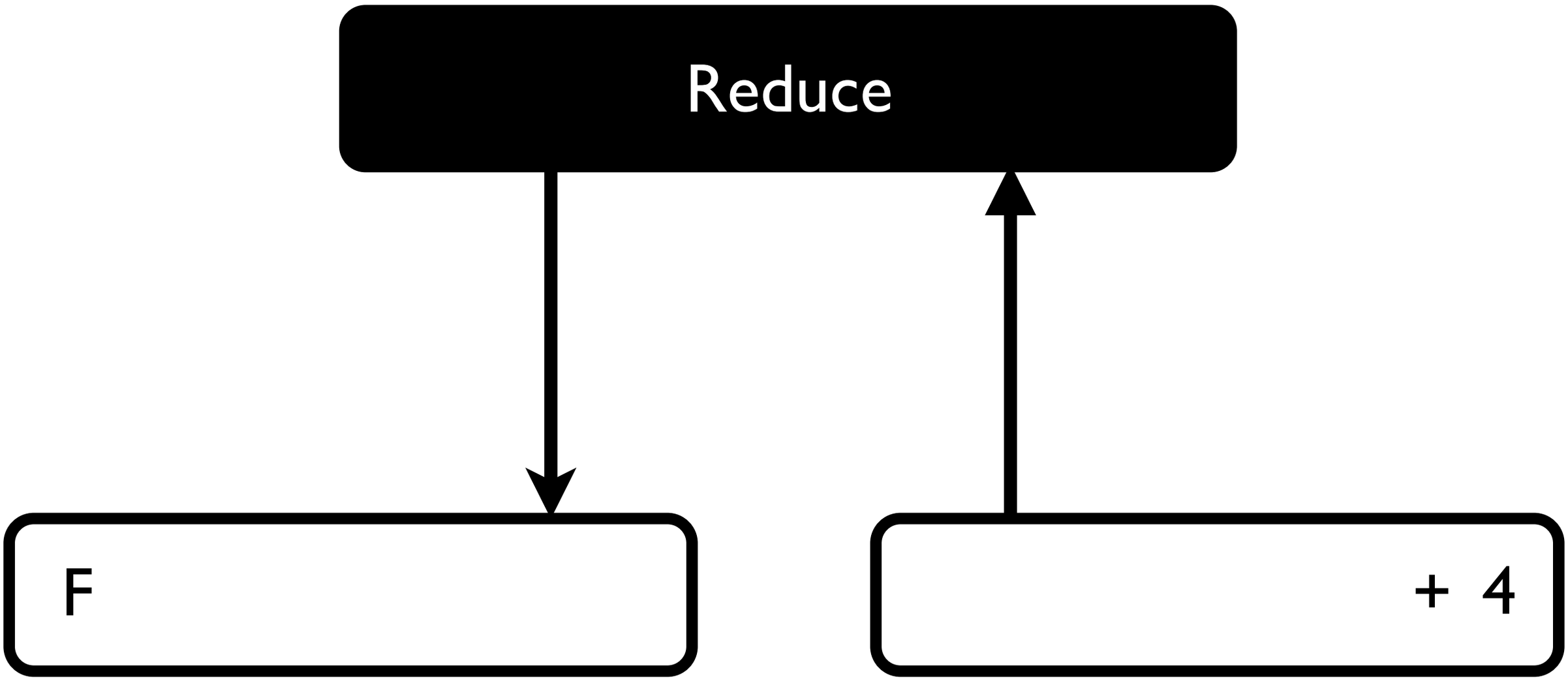
$$E$$

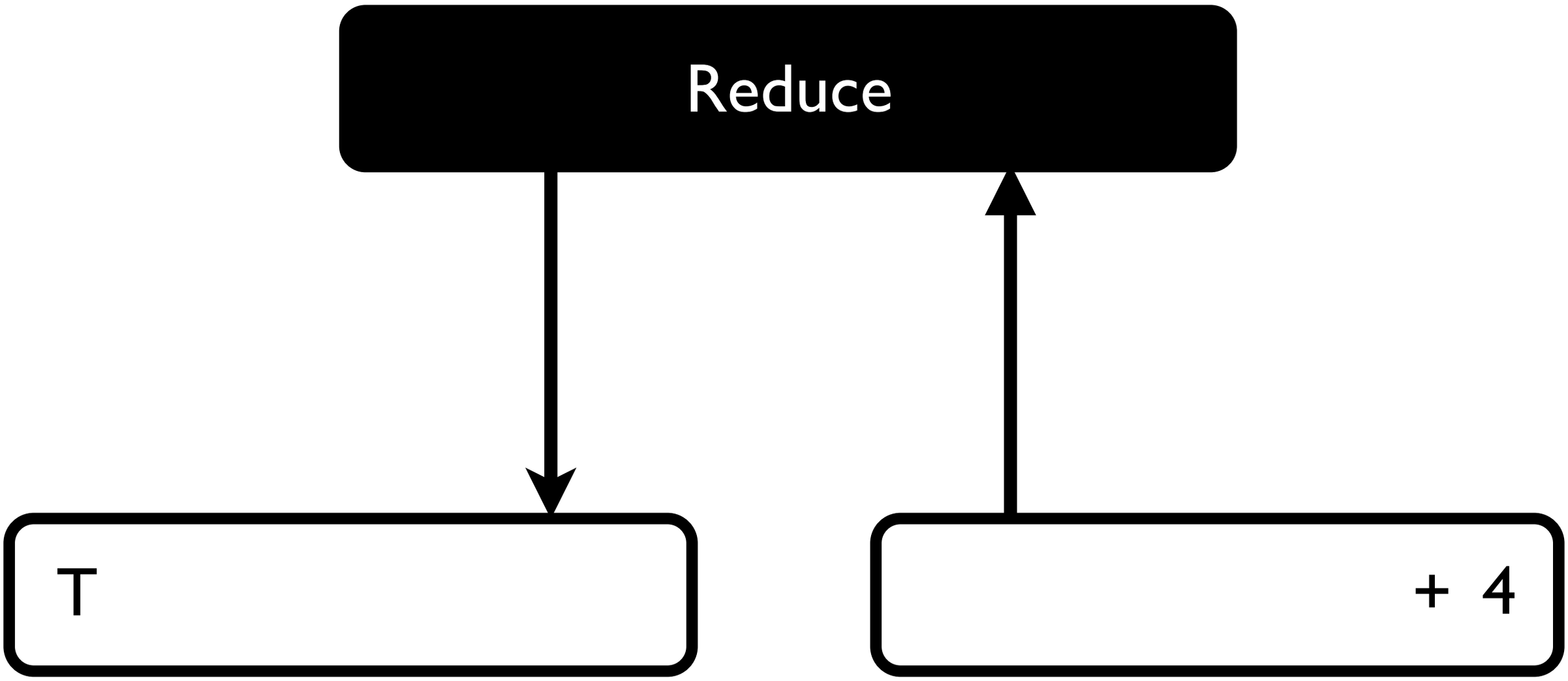
Bottom-up parsing

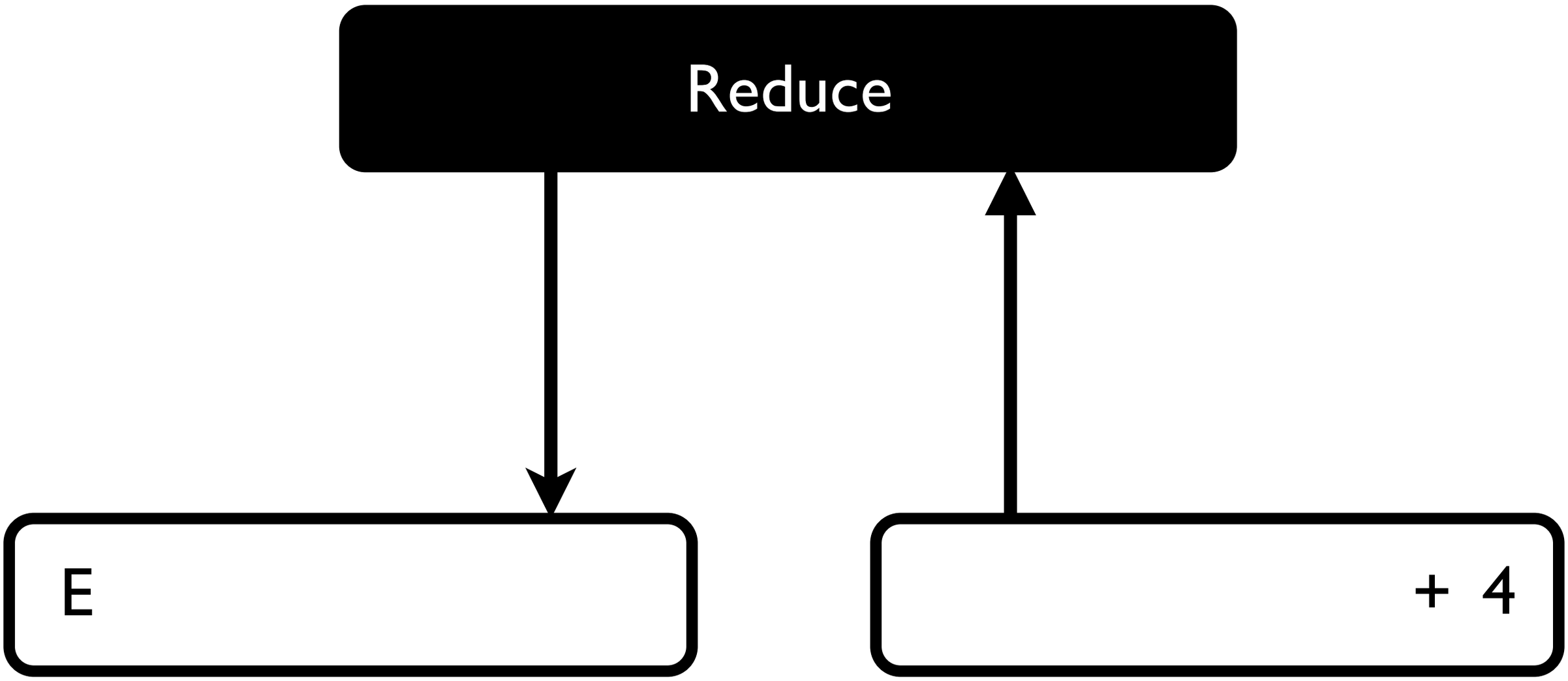
- State: Stack and remaining input
- Two choices: Shift or reduce
- Shift: Push the next character
- Reduce by rule: Pop stack; push symbol

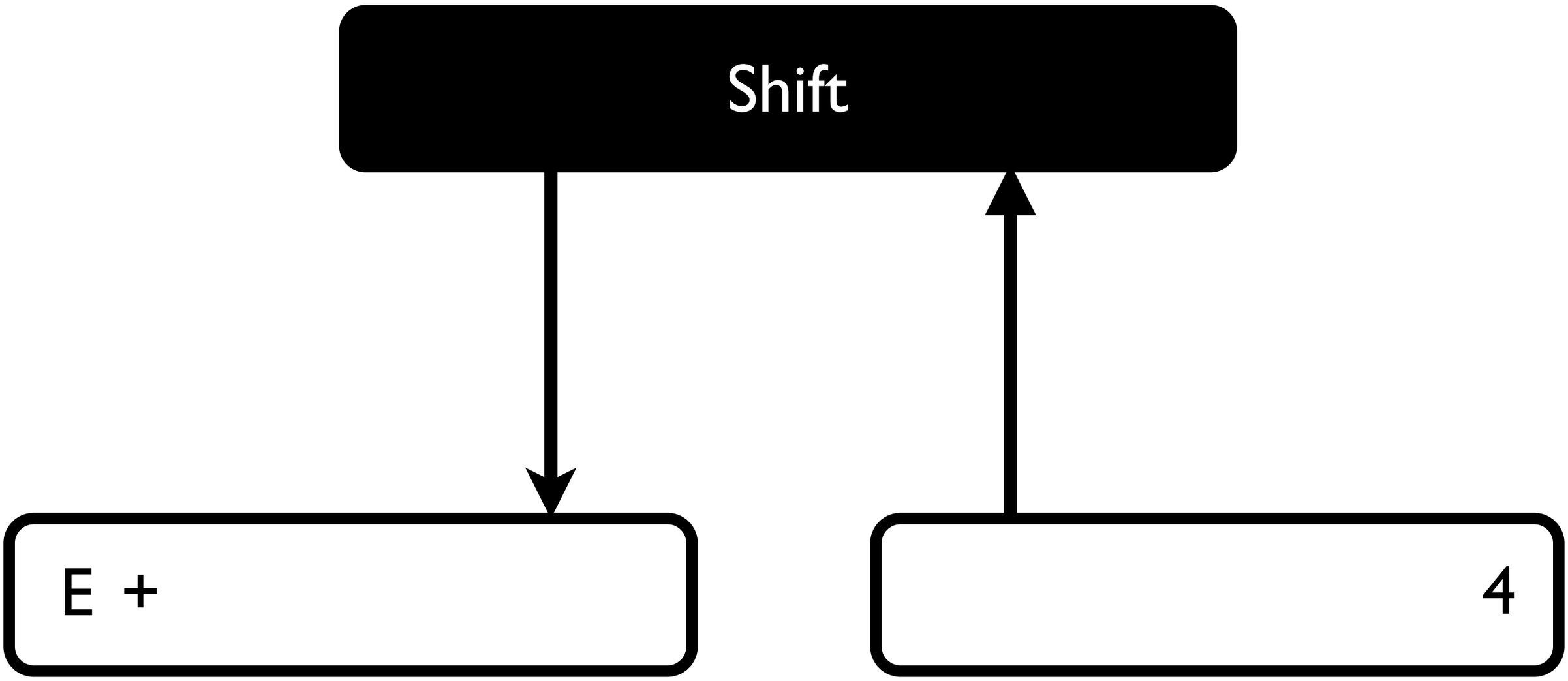

3 + 4

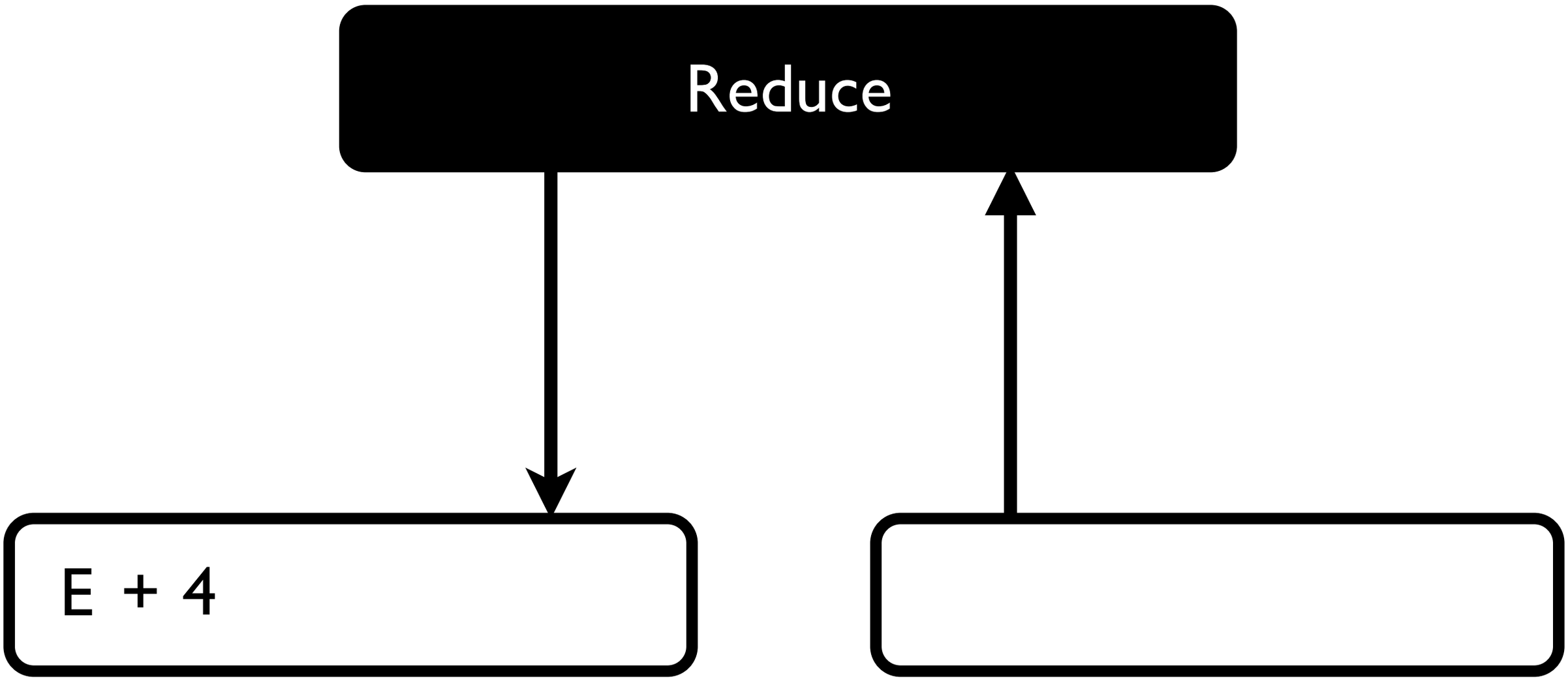

Shift
3
+ 4

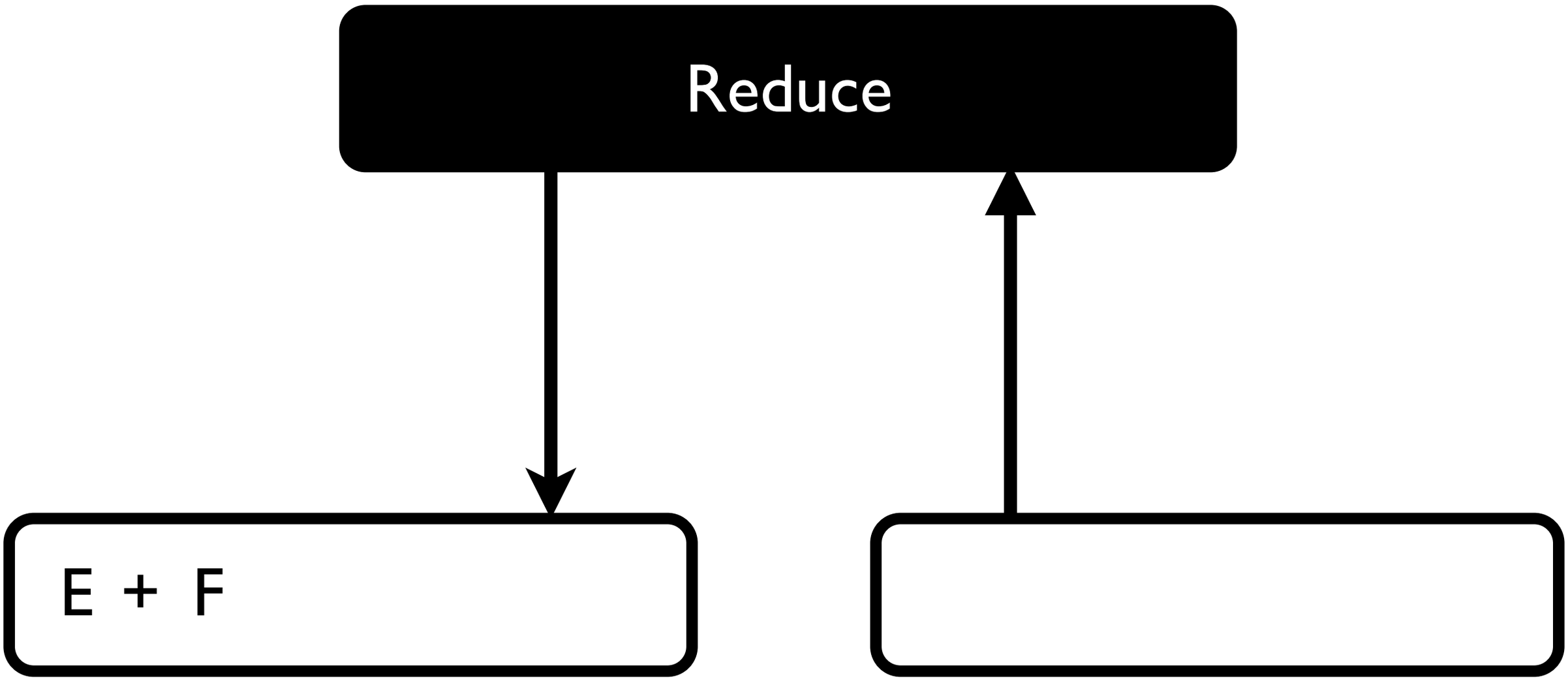

Reduce
F
+ 4

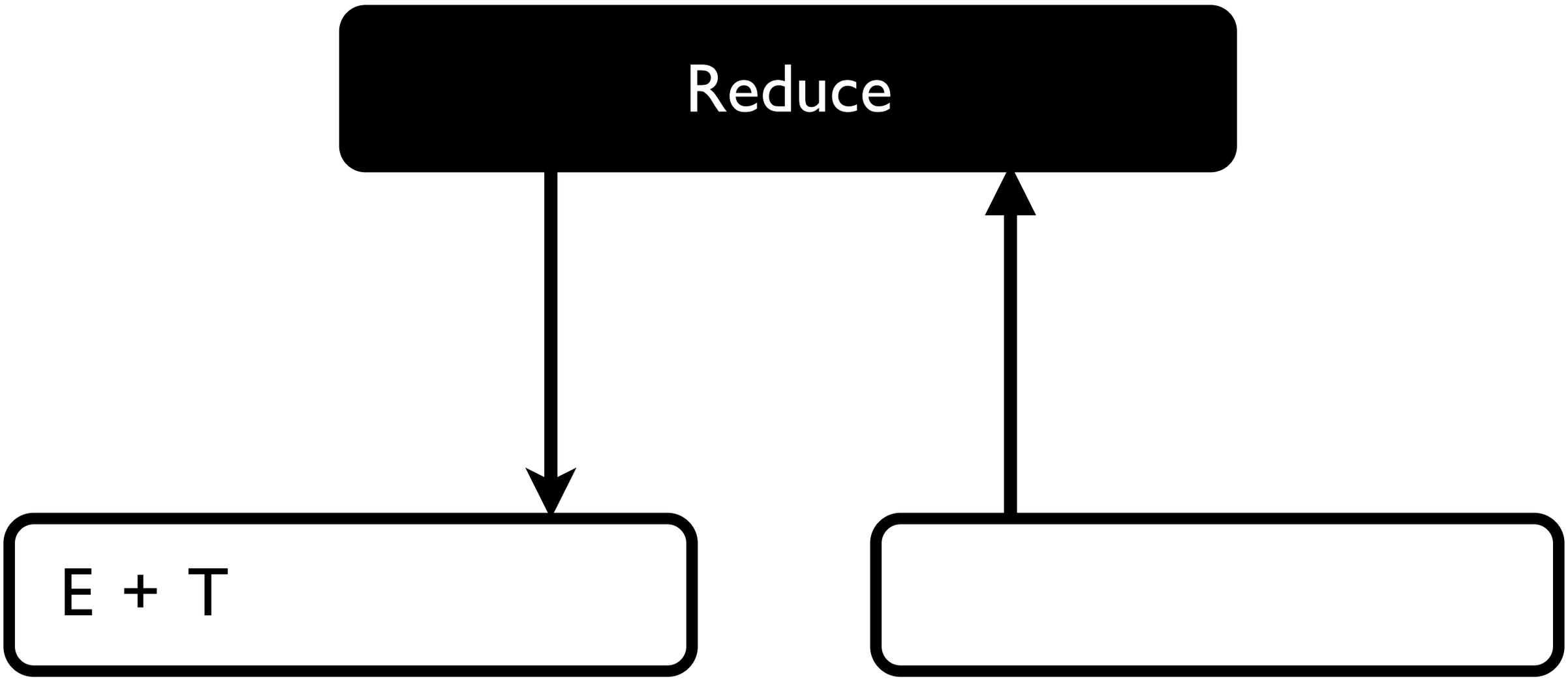

Reduce
T
+ 4


Reduce
E
+ 4


Shift
E +
4


Reduce
E + 4


Reduce
E + F

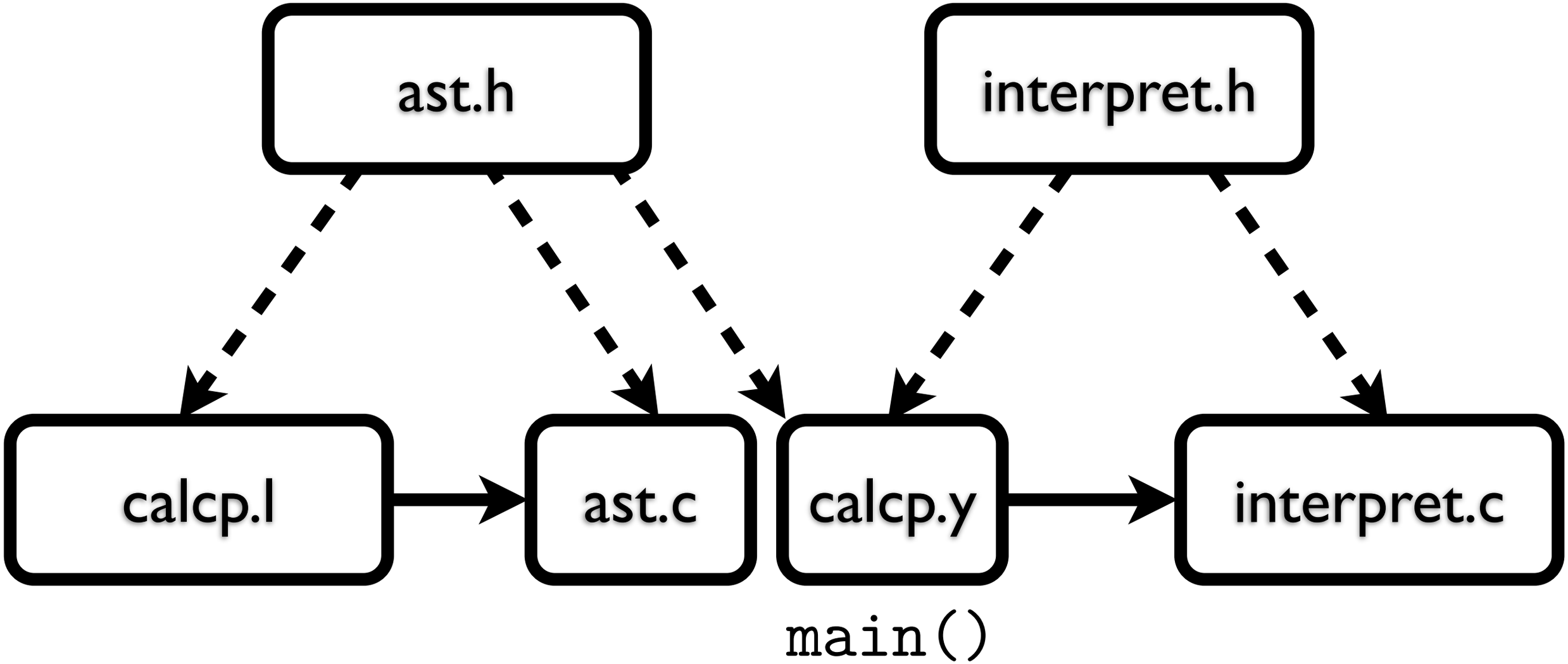

Reduce
E + T

Reduce

E

Exercise: `calc+`

```
pi = 3 ;
A(r) = pi * r * r ;
g(x,y) = x + A(y) ;
--
g(3,4)
```


ast.h
interpret.h
calcp.l
ast.c
calcp.y
interpret.c
main()

```
<prog> ::= { <decl> } '--' <exp>

<decl> ::= ID '=' <exp> ';'
         |  ID '(' <formals> ) '=' <exp> ';'

<formals> ::= ID { ',' ID }
            |

<exp> ::= <exp> '+' <term>
        |  <term>
```

```
<term> ::= <term> '*' <factor>
         |  <factor>

<factor> ::= INT
           |  ID
           |  ID '(' <args> ')'
           |  '(' <exp> ')'

<args> ::= <exp> { , <exp> }
         |
```

+ Identifiers match [A-Za-z_][A-Za-z_0-9]*

+ Numerals are decimal integers

+ Whitespace is ignored, except to separate tokens.

+ Tokens that act as separators are:

  ( ) = ; , * + --

Questions?