

Programming Abstractions

Lecture 16: Backtracking continued and grammars

Stephen Checkoway

Announcements

Due dates for homework 5, 6, 7, and 8 changed!

New dates:

- Homework 5: Friday, November 19
- Homework 6: Friday, December 03
- Homework 7: Friday, December 17
- Homework 8: Friday, January 07

Backtracking in Racket

```
; sofar is the list of steps so far in reverse order
; curr is the current value to try
(define (backtrack params sofar curr)
  (cond [<sofar is a complete solution> (reverse sofar)]
        [<curr is out of the range of possible values> #f]
        [(feasible sofar curr)
         (let ([res (backtrack params
                               (cons curr sofar)
                               <first value for next step>))])
          (if res
              res
              (backtrack params sofar <value after curr>)))]
        [else (backtrack params sofar <value after curr>)])
```

Using backtrack

(Of course, you'll write specific backtrack and feasible functions for each problem)

(backtrack params empty **⟨first value for first step⟩**)

n-queens

(single solution)

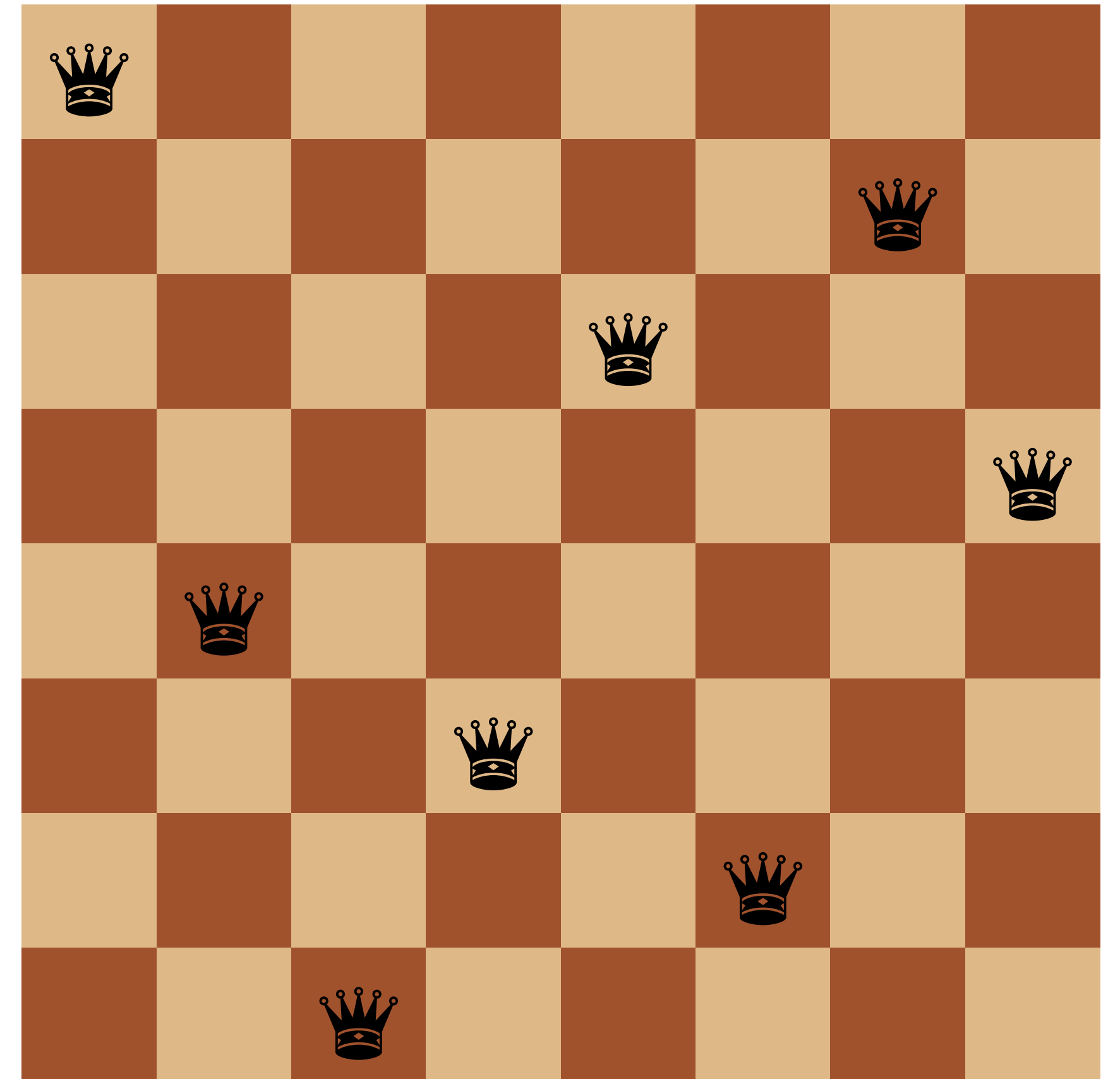
First, how should we represent a solution?

- A list of row-column pairs like
`' ((0 0) (4 1) (7 2) (5 3)
 (2 4) (6 5) (1 6) (3 7))`
- A list of rows like `' (0 4 7 5 2 6 1 3)`

Either works and we can easily convert from one to the other

- `(map list list-of-rows (range n))`
- `(map first list-of-pairs)`
The list must be sorted by column first

Let's use a list of rows



Careful!

Our normal procedure for constructing the list of steps prepends the current step to our partial solution

- `(bt (cons curr sofar) initial)`

This means our partial solution will be in reverse order which means we need to

- reverse our final result so it's in the correct order; and
- write our (*feasible?* sofar curr) procedure keeping this in mind

n-queens

```
(define (bt n sofar curr)
  (cond [(is-complete? sofar) (reverse sofar)]
        [(out-of-range? curr) #f]
        [(feasible? sofar curr)
         (let ([res (bt n (cons curr sofar) initial)])
           (if res
               res
               (bt n sofar (next curr))))])
        [else (bt n sofar (next curr))]))

(define (n-queens n)
  (bt n empty initial))
```

What's our `initial` value?

```
(define (bt n sofar curr)
  (cond [(is-complete? sofar) (reverse sofar)]
        [(out-of-range? curr) #f]
        [(feasible? sofar curr)
         (let ([res (bt n (cons curr sofar) initial)])
           (if res
               res
               (bt n sofar (next curr))))])
        [else (bt n sofar (next curr))]))

(define (n-queens n)
  (bt n empty initial))
```

A. 0

D. $n-1$

B. 1

E. $n+1$

C. n

What's our `(next curr)` procedure?

```
(define (bt n sofar curr)
  (cond [(is-complete? sofar) (reverse sofar)]
        [(out-of-range? curr) #f]
        [(feasible? sofar curr)
         (let ([res (bt n (cons curr sofar) initial)])
           (if res
               res
               (bt n sofar (next curr))))])
        [else (bt n sofar (next curr))])
```

```
(define (n-queens n)
  (bt n empty initial))
```

A. `(add1 curr)`

D. `(modulo (add1 curr) (add1 n))`

B. `(add1 (modulo curr n))`

E. More than one of the above

C. `(modulo (add1 curr) n)`

What's our `(is-complete? sofar)` procedure?

```
(define (bt n sofar curr)
  (cond [(is-complete? sofar) (reverse sofar)]
        [(out-of-range? curr) #f]
        [(feasible? sofar curr)
         (let ([res (bt n (cons curr sofar) initial)])
           (if res
               res
               (bt n sofar (next curr))))])
        [else (bt n sofar (next curr))]))
```

```
(define (n-queens n)
  (bt n empty initial))
```

A. `(feasible? sofar null)`

D. `(= (length sofar) (sub1 n))`

B. `(= (length sofar) n)`

E. More than one of the above

C. `(= (length sofar) (add1 n))`

What's our `(out-of-range? curr)` procedure?

```
(define (bt n sofar curr)
  (cond [(is-complete? sofar) (reverse sofar)]
        [(out-of-range? curr) #f]
        [(feasible? sofar curr)
         (let ([res (bt n (cons curr sofar) initial)])
           (if res
               res
               (bt n sofar (next curr))))])
        [else (bt n sofar (next curr))])
```

```
(define (n-queens n)
  (bt n empty initial))
```

A. `(< curr n)`

D. `(< n 0)`

B. `(= curr n)`

E. `(not (integer? curr))`

C. `(> curr n)`

feasible?

There are three conditions

- No two queens share the same column
 - Easy, we're picking one queen per column so this is always satisfied
- No two queens share the same row
 - We'll need to check that `sofar` doesn't already contain `curr`
- No two queens share the same diagonal
 - Two diagonals to check: up-left from `curr` and down-left from `curr`
 - Lots of ways to do this, here's one: move left through columns; up through rows

```
(define (up-left-ok? queen-rows row)
  (cond [(empty? queen-rows) #t]
        [(= (first queen-rows) row) #f]
        [else (up-left-ok? (rest queen-rows) (sub1 row))]))
(up-left-ok? sofar (sub1 curr))
```

feasible?

There are three conditions

- No two queens share the same column
 - Easy, we're picking one queen per column so this is always satisfied
- No two queens share the same row
 - We'll need to check that `sofar` doesn't already contain `curr`
- No two queens share the same diagonal
 - Two diagonals to check: up-left from `curr` and down-left from `curr`
 - Lots of ways to do this, here's one: move left through columns; up through rows

```
(define (up-left-ok? queen-rows row)
  (cond [(empty? queen-rows) #t]
        [(= (first queen-rows) row) #f]
        [else (up-left-ok? (rest queen-rows) (sub1 row))]))
(up-left-ok? sofar (sub1 curr))
```

Move left through
reversed columns

feasible?

There are three conditions

- No two queens share the same column
 - Easy, we're picking one queen per column so this is always satisfied
- No two queens share the same row
 - We'll need to check that `sofar` doesn't already contain `curr`
- No two queens share the same diagonal
 - Two diagonals to check: up-left from `curr` and down-left from `curr`
 - Lots of ways to do this, here's one: move left through columns; up through rows

```
(define (up-left-ok? queen-rows row)
  (cond [(empty? queen-rows) #t]
        [(= (first queen-rows) row) #f]
        [else (up-left-ok? (rest queen-rows) (sub1 row))]))
(up-left-ok? sofar (sub1 curr))
```

Move left through
reversed columns

Move up through rows

At various points, the backtracking algorithm needs to choose the next value to try for the current step or it needs to backtrack to a previous step.

When does it need to backtrack to a previous step?

- A. It backtracks each time it encounters a partial solution that isn't feasible
- B. It backtracks whenever there are no more choices for the current step
- C. It backtracks when the choice it makes for the final step leads to an invalid solution
- D. It backtracks after each invalid choice
- E. All of the above

One common variant: all solutions

Rather than using `#f` to signal failure, we'll use `empty` to indicate the set of solutions is empty

Key differences

- Rather than stopping after a single solution is found, keep going
- Each call will return a list of solutions
- When we have a feasible solution, we need to get all the solutions both using the feasible one and not

All solutions in Racket

```
(define (all-sol params sofar curr)
  (cond [<sofar is a complete solution> (list (reverse sofar))]
        [<curr is out of the range of possible values> '()]
        [(feasible sofar curr)
         (let ([res1 (all-sol params
                               (cons curr sofar)
                               <first value for next step>))]
              [res2 (all-sol params sofar <value after curr>)]))
         (append res1 res2)])
        [else (all-sol params sofar <value after curr>)]))

(all-sol params empty <first value for first step>)
```

Permutations of $\{0, 1, \dots, n-1\}$

(Not the most efficient way)

Let's compute all permutations of $\{0, 1, \dots, n-1\}$ using backtracking

```
(define (bt n sofar curr)
  (cond [(is-complete? sofar) (list sofar)]
        [(out-of-range? curr) empty]
        [(feasible? sofar curr)
         (let ([with-curr (bt n (cons curr sofar) initial)]
               [without-curr (bt n sofar (next curr))])
           (append with-curr without-curr))]
        [else (bt n sofar (next curr))]))

(define (all-perms n)
  (bt n empty initial))
```

We just need to deal with the **problem-specific parts**

n-queens all solutions

No harder than getting one solution, we just need to plug in the **usual parts**

```
(define (bt n sofar curr)
  (cond [(is-complete? sofar) (list (reverse sofar))]
        [(out-of-range? curr) empty]
        [(feasible? sofar curr)
         (let ([with-curr (bt n (cons curr sofar) initial)]
               [without-curr (bt n sofar (next curr))])
           (append with-curr without-curr))]
        [else (bt n sofar (next curr))]))

(define (all-queens n)
  (bt n empty initial))
```


A quick introduction to grammars

Alphabets and words

An **alphabet** Σ is a finite, nonempty set of symbols

- $\{0, 1\}$ is a binary alphabet
- The set of emoji is an alphabet
- The set of English words is an alphabet

A **word** (also called a string) w over an alphabet Σ is a finite (possibly-empty) sequence of symbols from the alphabet

- The empty word, ε , consisting of no symbols is a word over every alphabet
- 001101 is a word over $\{0, 1\}$
-  is a word over the emoji alphabet
- functional programming is great is a word over English

Languages

A **language** is a (possibly infinite) set of words over an alphabet

There's a whole lot we can do studying languages as mathematical objects

We're not going to do that in this course, take theory of computation to find out more!

For a given programming language (like Scheme) the alphabet is the set of keywords, identifiers, and symbols in the language

- This is a bit of a simplification because there are infinitely many possible identifiers but alphabets must be finite

A word (or string) over this alphabet is in the programming language if it is a syntactically valid program

Syntactically valid?

Consider the invalid Scheme program

```
(let ([x 5]
      [y 32])
  (+ z 2))
```

This is *syntactically* valid (i.e., it's a word in the Scheme language) but *semantically* meaningless as we don't have a binding for the identifier `z`

Grammars

A grammar for a language is a (mathematical) tool for specifying which words over the alphabet belong to the language

Grammars are often used to determine the meaning of words in the language

Grammars

For example, consider the arithmetic expression $a+b^*c$ as a word over the alphabet consisting of variables and arithmetic operators

- ▶ We can write many different grammars that will let us determine if a given word is a valid expression (i.e., is in the language of valid expressions)
- ▶ With a careful choice of grammars we can determine that this means $a+(b^*c)$ and not $(a+b)^*c$

Grammars are very old, dating back to at least Yāska the 4th c. BCE

Mathematical representation of grammars

A grammar G is a 4-tuple $G = (V, \Sigma, S, R)$ where

- V is a finite, nonempty set of *nonterminals*, also called variables
- Σ is an alphabet of *terminal* symbols
- $S \in V$ is the *start* nonterminal
- R is a finite set of *production rules*

(Terminal symbols are distinct from nonterminals)

In English, we might have nonterminals like *NOUN*, *VERB*, *NP*, etc.

We often write nonterminals in upper-case and terminals in lower-case

Production rules

Nonterminals are expanded using production rules to sequences of terminals and nonterminals

A production rule looks has the form

$$A \rightarrow \alpha$$

where A is a nonterminal and α is a (possibly-empty) word over $\Sigma \cup V$

Here's an example for Scheme

$$EXP \rightarrow (\text{if } EXP \text{ } EXP \text{ } EXP)$$

This says that wherever we have an expression, we can expand it to an if-then-else expression which starts with (followed by if and then three more expressions and lastly)

Example grammar for arithmetic

$EXP \rightarrow EXP + TERM$

$EXP \rightarrow TERM$

$TERM \rightarrow TERM * FACTOR$

$TERM \rightarrow FACTOR$

$FACTOR \rightarrow (EXP)$

$FACTOR \rightarrow \text{number}$

Compact form:

$EXP \rightarrow EXP + TERM \mid TERM$

$TERM \rightarrow TERM * FACTOR \mid FACTOR$

$FACTOR \rightarrow (EXP) \mid \text{number}$

Derivations

A derivation with a grammar starts with a nonterminal and replaces nonterminals one at a time until only a sequence of terminals remains

Derivation example

Left-most derivation of $3 + 4 * 50$

EXP \Rightarrow

EXP \rightarrow *EXP* + *TERM* | *TERM*

TERM \rightarrow *TERM* * *FACTOR* | *FACTOR*

FACTOR \rightarrow (*EXP*) | number

Derivation example

Left-most derivation of $3 + 4 * 50$

$EXP \Rightarrow EXP + TERM$

$EXP \rightarrow EXP + TERM \mid TERM$

$TERM \rightarrow TERM * FACTOR \mid FACTOR$

$FACTOR \rightarrow (EXP) \mid \text{number}$

Derivation example

Left-most derivation of $3 + 4 * 50$

$EXP \Rightarrow EXP + TERM$

$\Rightarrow TERM + TERM$

$EXP \rightarrow EXP + TERM \mid TERM$

$TERM \rightarrow TERM * FACTOR \mid FACTOR$

$FACTOR \rightarrow (EXP) \mid \text{number}$

Derivation example

Left-most derivation of $3 + 4 * 50$

$EXP \Rightarrow EXP + TERM$

$\Rightarrow TERM + TERM$

$\Rightarrow \textcolor{brown}{FACTOR} + TERM$

$EXP \rightarrow EXP + TERM \mid TERM$

$TERM \rightarrow TERM * FACTOR \mid FACTOR$

$\textcolor{brown}{FACTOR} \rightarrow (EXP) \mid \textcolor{brown}{number}$

Derivation example

Left-most derivation of $3 + 4 * 50$

$EXP \Rightarrow EXP + TERM$

$\Rightarrow TERM + TERM$

$\Rightarrow FACTOR + TERM$

$\Rightarrow 3 + TERM$

$EXP \rightarrow EXP + TERM \mid TERM$

$TERM \rightarrow TERM * FACTOR \mid FACTOR$

$FACTOR \rightarrow (EXP) \mid \text{number}$

Derivation example

Left-most derivation of $3 + 4 * 50$

$EXP \Rightarrow EXP + TERM$

$\Rightarrow TERM + TERM$

$\Rightarrow FACTOR + TERM$

$\Rightarrow 3 + TERM$

$\Rightarrow 3 + TERM * FACTOR$

$EXP \rightarrow EXP + TERM \mid TERM$

$TERM \rightarrow TERM * FACTOR \mid FACTOR$

$FACTOR \rightarrow (EXP) \mid \text{number}$

Derivation example

Left-most derivation of $3 + 4 * 50$

$EXP \Rightarrow EXP + TERM$

$\Rightarrow TERM + TERM$

$\Rightarrow FACTOR + TERM$

$\Rightarrow 3 + TERM$

$\Rightarrow 3 + TERM * FACTOR$

$\Rightarrow 3 + FACTOR * FACTOR$

$EXP \rightarrow EXP + TERM \mid TERM$

$TERM \rightarrow TERM * FACTOR \mid FACTOR$

$FACTOR \rightarrow (EXP) \mid \text{number}$

Derivation example

Left-most derivation of $3 + 4 * 50$

$EXP \Rightarrow EXP + TERM$

$\Rightarrow TERM + TERM$

$\Rightarrow FACTOR + TERM$

$\Rightarrow 3 + TERM$

$\Rightarrow 3 + TERM * FACTOR$

$\Rightarrow 3 + FACTOR * FACTOR$

$\Rightarrow 3 + 4 * FACTOR$

$EXP \rightarrow EXP + TERM \mid TERM$

$TERM \rightarrow TERM * FACTOR \mid FACTOR$

$FACTOR \rightarrow (EXP) \mid \text{number}$

Derivation example

Left-most derivation of $3 + 4 * 50$

$EXP \Rightarrow EXP + TERM$

$\Rightarrow TERM + TERM$

$\Rightarrow FACTOR + TERM$

$\Rightarrow 3 + TERM$

$\Rightarrow 3 + TERM * FACTOR$

$\Rightarrow 3 + FACTOR * FACTOR$

$\Rightarrow 3 + 4 * FACTOR$

$\Rightarrow 3 + 4 * 50$

$EXP \rightarrow EXP + TERM \mid TERM$

$TERM \rightarrow TERM * FACTOR \mid FACTOR$

$FACTOR \rightarrow (EXP) \mid \text{number}$

Parse tree

Corresponds to the left-most derivation

$EXP \Rightarrow EXP + TERM$

$\Rightarrow TERM + TERM$

$\Rightarrow FACTOR + TERM$

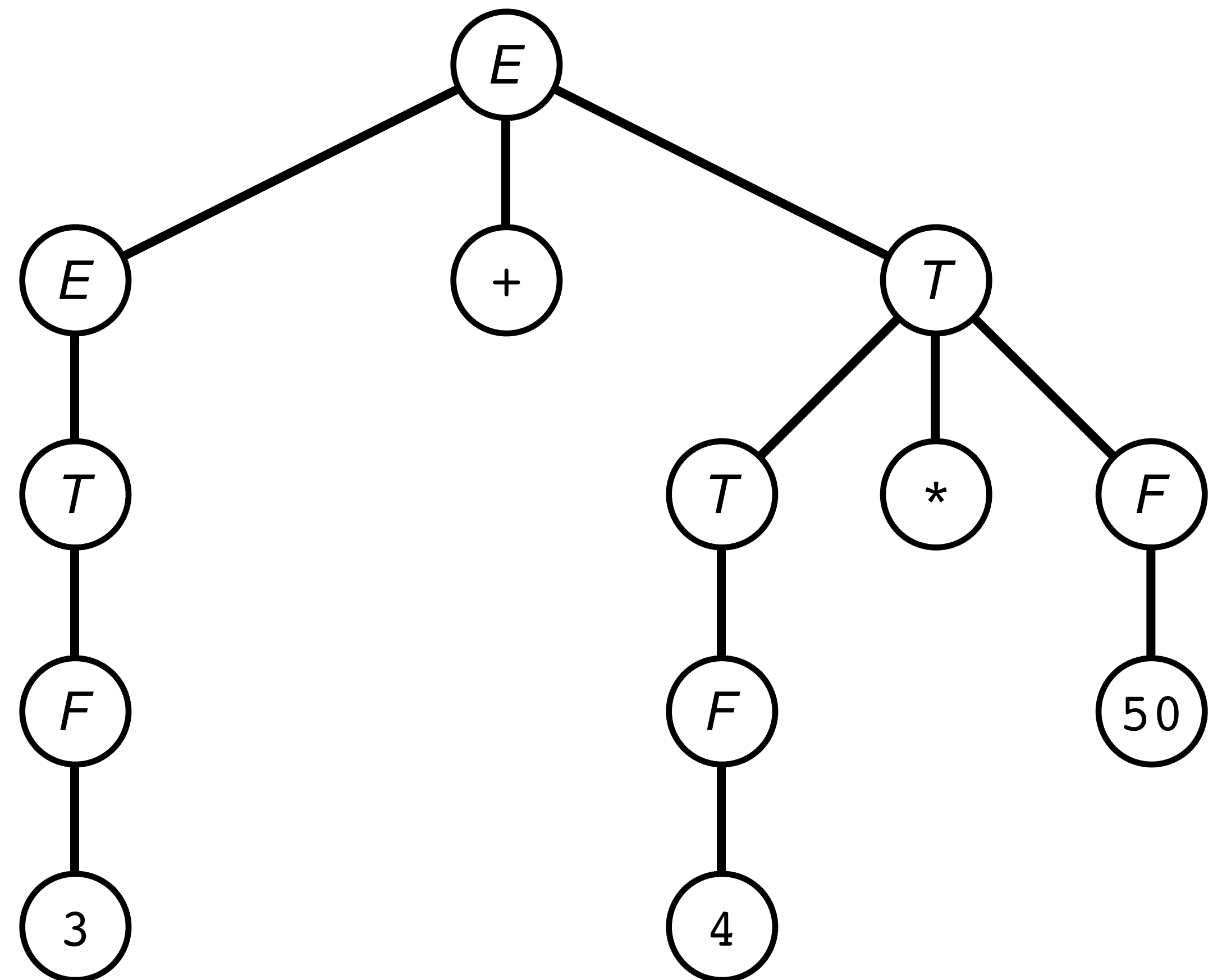
$\Rightarrow 3 + TERM$

$\Rightarrow 3 + TERM * FACTOR$

$\Rightarrow 3 + FACTOR * FACTOR$

$\Rightarrow 3 + 4 * FACTOR$

$\Rightarrow 3 + 4 * 50$

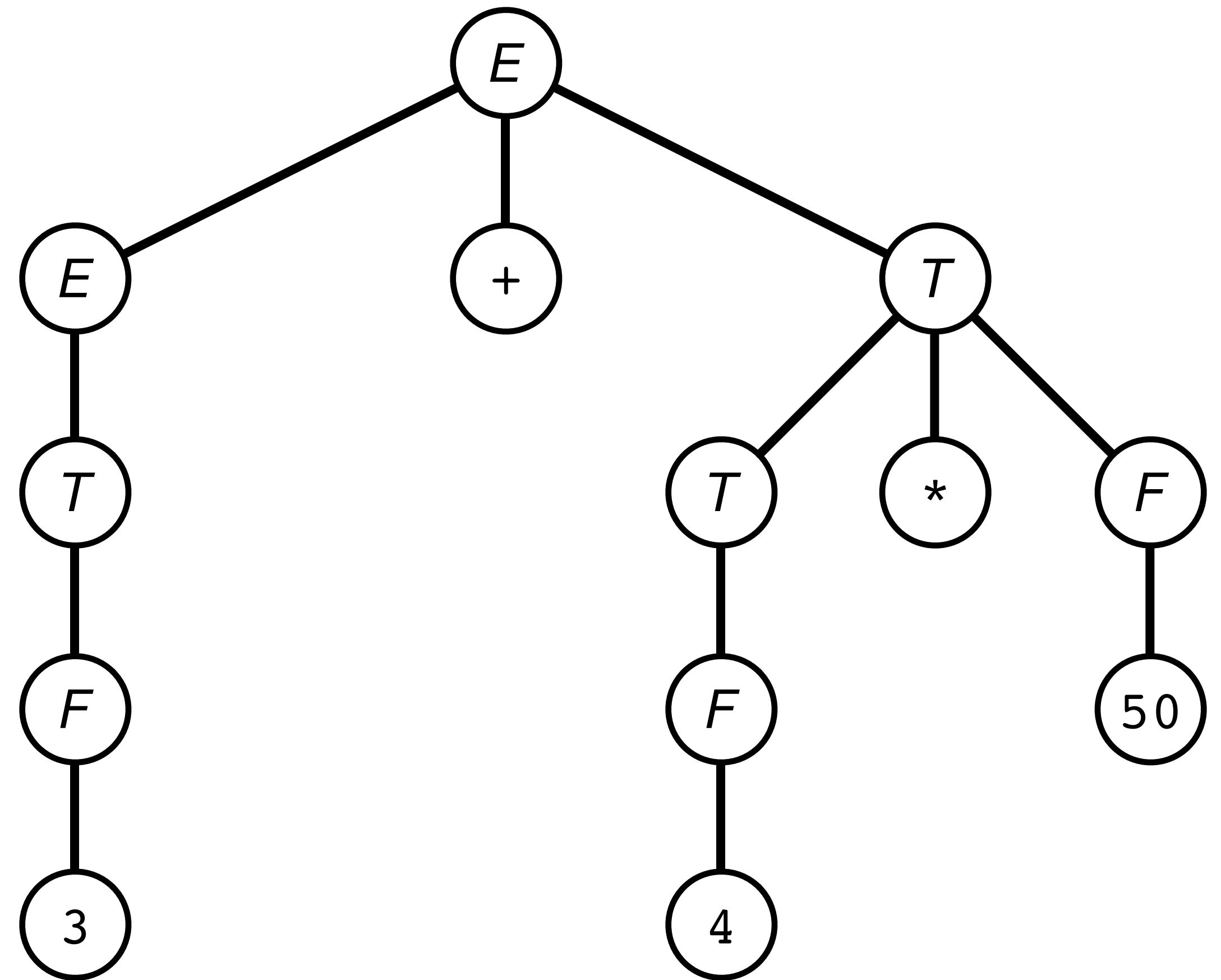


Note that the derived expression is a left-to-right traversal of the leaves

Parse tree

The structure of the tree encodes the order of operation

It's clear that we have to evaluate the $4 * 50$ before we can add to the 3



The language generated by a grammar

One nonterminal is designated as the start nonterminal

- Typically, this is the nonterminal on the left-hand side of the first production rule

The language *generated* by the grammar is the set of words over the terminal alphabet which can be derived by the production rules, starting with the start nonterminal

Given our grammar for arithmetic

- $1 * (2 + 3)$ is in the language generated by the grammar
- $85 + * 10$ is not

Why do we care (in 275)?

We're going to specify a grammar for MiniScheme

We'll use this to

- specify what needs to be implemented in each part
- a guide for how MiniScheme should be parsed

There's a strong connection between grammars and parsing which we won't explore in this course

A convenient shorthand

It's often useful to say that a particular terminal or nonterminal can appear 0 or more times

$$A \rightarrow xA \mid \varepsilon$$

where x is either a terminal or nonterminal and ε represents the empty word

Similarly, it's often useful to say that a particular terminal or nonterminal can appear 1 or more times

$$A \rightarrow xA \mid x$$

We write x^* or x^+ as a shorthand for these constructs

A full grammar for Minischeme

$EXP \rightarrow$ number
| symbol
| (if $EXP\ EXP\ EXP$)
| (let ($LET-BINDINGS$) EXP)
| (letrec ($LET-BINDINGS$) EXP)
| (lambda ($PARAMS$) EXP)
| (set! symbol EXP)
| (begin EXP^*)
| (EXP^+)

$LET-BINDINGS \rightarrow LET-BINDING^*$

$LET-BINDING \rightarrow$ [symbol EXP]

$PARAMS \rightarrow$ symbol^{*}