Graph Reachability: Getting from a to b

One approach (among many!)

✓ Keep a collection of “partial” paths starting at a
  ▶ For efficiency, we might represent a path as a list in reverse order.

✓ Repeatedly take out a path (the “active path”)
  ▶ If it ends at the destination b, we’re done!
  ▶ If not, find reasonable ways to make it one step longer, add them to our collection.
Graph Reachability: Getting from \textit{a} to \textit{b}

One approach (among many!)

✔ Keep a collection of “partial” paths starting at \textit{a}
  ▶ For efficiency, we might represent a path as a list in reverse order.

✔ Repeatedly take out a path (the “active path”)
  ▶ If it ends at the destination \textit{b}, we’re done!
  ▶ If not, find reasonable ways to make it one step longer, add them to our collection.

What’s “reasonable?”

✔ Don’t create paths with cycles (loops)

✔ Or, don’t create new paths to places you’ve already reached.
  (Requires keeping track separately of which nodes have been reached.)
(define (reach a b G)
  (reach-help b (list (list a)) G))

(define (reach-help dst path-list G)
  (if (null? path-list)
      #f
      (let ((active-path (first path-list))
            (remaining-paths (rest path-list)))
        ; If active path contains the destination, return #t
        ; Otherwise, expand the active path by one node for
        ; each kid, ignoring kids that create cycles, making
        ; a list of 0 or more new paths.
        ; Append this new list of paths onto the remaining
        ; paths and recurse.
        ; At least one helper function (to build the list
        ; of new paths) is recommended.
        ...
        ...))
Weighted Graphs

Find the shortest path from Bucharest to Arad.
Dijkstra's Algorithm

Greedy/Best-First Search

One implementation strategy:

✓ A collection of partial paths (with lengths)
✓ Always pick the shortest partial path as your “active path”.
✓ Stop when the active path reaches your destination.

Note:

✓ The first time an active path ends at any node \( n \), this will be the shortest path from a to \( n \).
✓ We could tweak the code to find shortest paths from a to all the nodes in the graph!

How to get from a to e?
An example interaction

Welcome to DrRacket, version 5.1.3 [3m].
Language: racket; memory limit: 128 MB.
> (read-eval-print)

input> 5 + 4
result: 9

input> 5 * 5
result: 25

input> 5 + 3 * 2
result: 11

input> (5 + 3) * 2
result: 16

Looks simple. How to implement it? What are the pieces?
**Structure of a Generic Interpreter**

1. **Lexical Analyzer (a.k.a. Tokenizer, Lexer, Scanner)**
   - Turns input (a linear sequence of characters) into a linear sequence of tokens/lexemes.

2. **Parser**
   - Turns the linear sequence of tokens into a structured representation of the program (usually a tree).

3. **Evaluator**
   - Process/interpret/run the structured representation to compute a final answer
   - May need to maintain an “environment” that associates program variables to their values.

4. **Printer**
   - Display the final result.
Minimath: 32+2 * 5

1. Lexical Analyzer (a.k.a. Tokenizer, Lexer, Scanner)

2. Parser

3. Evaluator

4. Printer
Racket: (average 44 (+ 15 15))

1. Lexical Analyzer (a.k.a. Tokenizer, Lexer, Scanner)

2. Parser

3. Evaluator

4. Printer
Syntax vs. Semantics
Syntax vs. Semantics

✓ Syntax: Formation Rules
  ▶ Colorless green ideas sleep furiously.
  ▶ (+ 3 true)

✓ Semantics: Meaning
SPECIFYING SYNTAX VIA CFGs

A context-free grammar is a set of rules for producing a set of strings (a language).
Specifying Syntax via CFGs

A context-free grammar is a set of rules for producing a set of strings (a language).

\[
S \rightarrow V + S \mid V \\
V \rightarrow 0 \mid 1 \mid 2 \mid \cdots \mid 9
\]

Ingredients:

✓ Nonterminals: \( S, V \)
✓ Terminals: +, 0, 1, 2, …, 9
✓ Production rules: (see above)
✓ Where to start: \( S \)

Show how to produce 4 starting from \( S \).
Show how to produce 4 + 5 starting from \( S \).
What other strings can we produce?
Using Structure to Clarify Meaning

He gave her cat food.
He gave her cat food.
PARSE TREES

The parse tree of a string makes explicit how a string was produced:

✓ Root is the start symbol
✓ When we apply a rule, items on the right-hand-side become children

Parse trees for 4 and 4+5?

\[
S \rightarrow V + S \mid V \\
V \rightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9
\]
Parse Trees

Show the parse tree for $9 - 3 + 2$

$$S \rightarrow V + S \mid V - S \mid V$$

$$V \rightarrow 0 \mid 1 \mid 2 \mid \cdots \mid 9$$
“Quiz:” Grammars

\[
P \rightarrow S \cdot P \mid S \div P \mid V \\
S \rightarrow V + S \mid V - S \mid V \\
V \rightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9
\]

1. Draw the parse tree for
   \[7 \cdot 3 + 9 \div 2\]

2. How could we change the grammar to get a parse tree with the usual mathematical meaning of this expression?

3. What could we add to permit parentheses ( and ) to group sub-expressions?