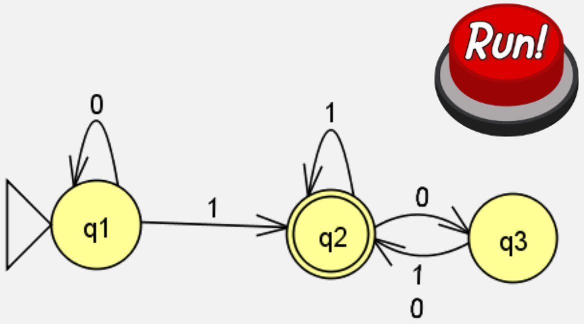
CS622

Computing With DFAs

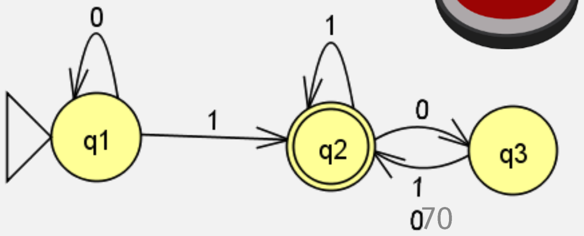
Friday, February 2, 2024

UMass Boston Computer Science



Announcements

- HW 1
  - Due: Wed 2/7 12pm (noon)



A Computation Model is … (from lecture 1)

- Some **definitions** …

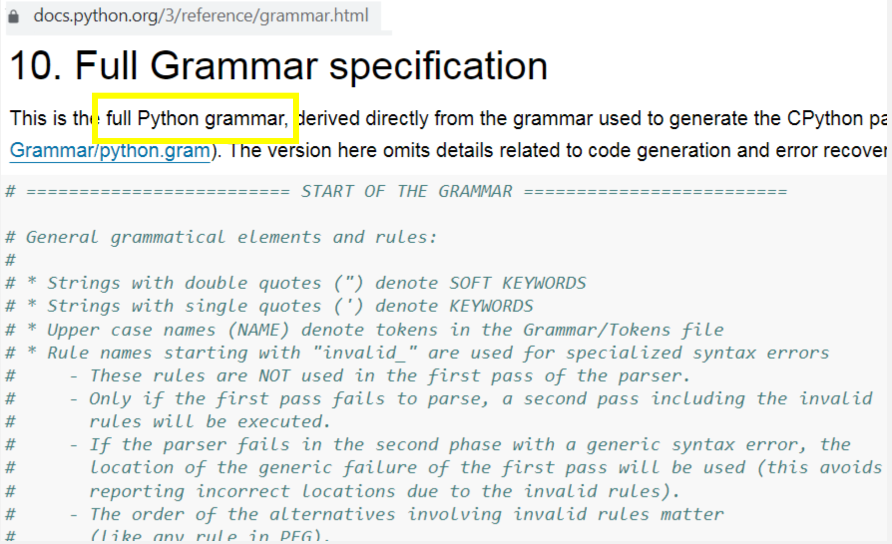
e.g., A **Natural Number** is either
- Zero
- a **Natural Number** + 1

- And **rules** that describe how to **compute** with the **definitions** …

To **add** two **Natural Numbers:**
1. Add the ones place of each num
2. Carry anything over 10
3. Repeat for each of remaining digits …

A Computation Model is … (from lecture 1)

- Some **definitions** …



docs.python.org/3/reference/grammar.html

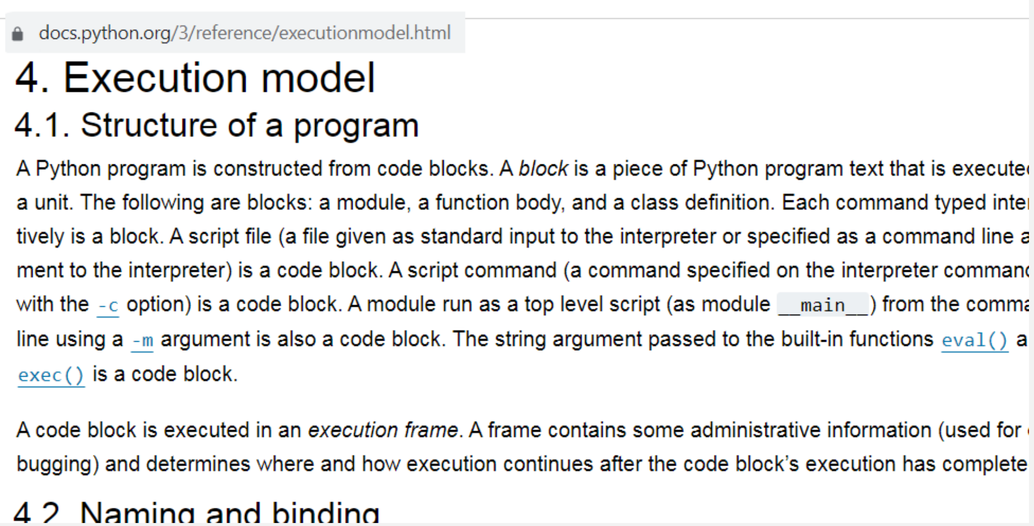
10. Full Grammar specification

This is the full Python grammar, derived directly from the grammar used to generate the CPython pa
Grammar/python.gram). The version here omits details related to code generation and error recover

```
# ======================== START OF THE GRAMMAR ========================

# General grammatical elements and rules:
#
# * Strings with double quotes (") denote SOFT KEYWORDS
# * Strings with single quotes (') denote KEYWORDS
# * Upper case names (NAME) denote tokens in the Grammar/Tokens file
# * Rule names starting with "invalid_" are used for specialized syntax errors
#     - These rules are NOT used in the first pass of the parser.
#     - Only if the first pass fails to parse, a second pass including the invalid
#       rules will be executed.
#     - If the parser fails in the second phase with a generic syntax error, the
#       location of the generic failure of the first pass will be used (this avoids
#       reporting incorrect locations due to the invalid rules).
#     - The order of the alternatives involving invalid rules matter
#       (like any rule in PEG).
```

- And **rules** that describe how to **compute** with the **definitions** …



docs.python.org/3/reference/executionmodel.html

4. Execution model

4.1. Structure of a program

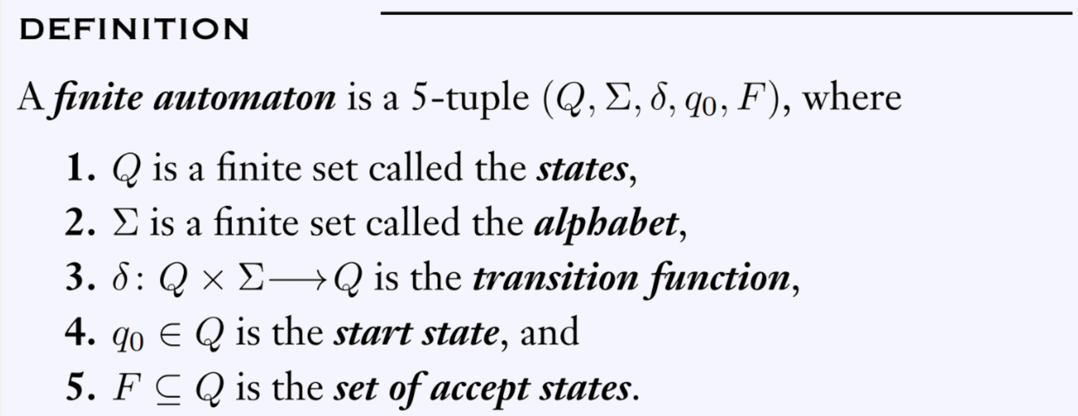
A Python program is constructed from code blocks. A *block* is a piece of Python program text that is execute
a unit. The following are blocks: a module, a function body, and a class definition. Each command typed inte
tively is a block. A script file (a file given as standard input to the interpreter or specified as a command line a
ment to the interpreter) is a code block. A script command (a command specified on the interpreter command
with the `-c` option) is a code block. A module run as a top level script (as module `__main__`) from the comma
line using a `-m` argument is also a code block. The string argument passed to the built-in functions `eval()` a
`exec()` is a code block.

A code block is executed in an *execution frame*. A frame contains some administrative information (used for
bugging) and determines where and how execution continues after the code block's execution has complete

4.2. Naming and binding

A Computation Model is … (from lecture 1)

- Some **definitions** …



DEFINITION

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

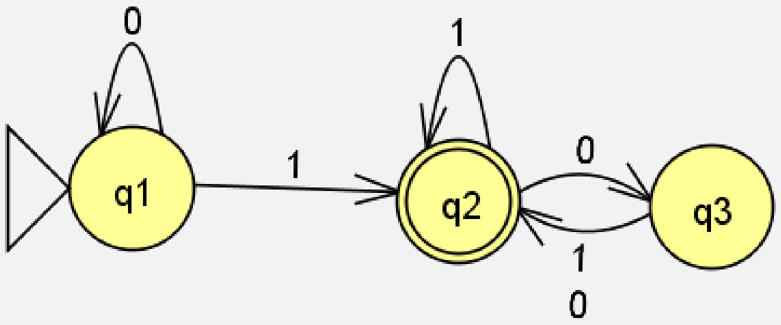
1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta\colon Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.

- And **rules** that describe how to **compute** with the **definitions** …

???

Computation with DFAs (JFLAP demo)

- DFA:



- Input: “`1101`”

HINT: always work out concrete examples to understand how a machine works

DFA Computation Rules

Informally

Given

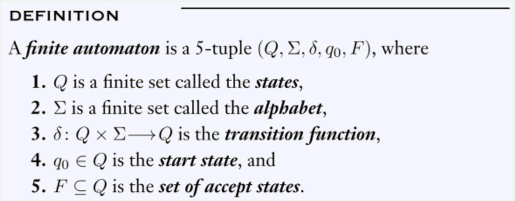
- A **DFA** (~ a "Program")
- and Input = **string of chars**, e.g. "`1101`"

To **run** the automata / "program":

- Start in **"start state"**

- **Repeat:**
  - Read **1 char** from **input;**
  - Change **state** according to the transition **table**

- **Result of computation =**
  - **Accept** if last state is **Accept state**
  - **Reject** otherwise


DEFINITION

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta\colon Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.

DFA Computation Rules

Informally

Given
- A **DFA** (~ a "Program") →
- and **Input** = **string of chars**, e.g. `“1101”` →

Formally (i.e., mathematically)

- $M =$
- $w =$

To **run** the automata / "program":
- Start in **"start state"**
- Repeat:
  - Read **1 char** from **input**;
  - Change **state** according to the **transition table**
- **Result of computation =**
  - **Accept** if last state is **Accept state**
  - **Reject** otherwise

DFA Computation Rules

Informally

Given

- A **DFA** (~ a "Program")
- and <u>**Input**</u> = **string of chars**, e.g. `"1101"`

To **run** the automata / "program":

- <u>Start</u> in "start state"
- <u>Repeat</u>:
  - <u>Read</u> **1 char** from **input**;
  - <u>Change</u> **state** according to the <u>**transition**</u> **table**
- <u>Result</u> **of computation** =
  - **Accept** if last state is **Accept state**
  - **Reject** otherwise

Formally (i.e., mathematically)

- $M = (Q, \Sigma, \delta, q_0, F)$
- $w = w_1 w_2 \cdots w_n$

A **run** is represented by **variables** $r_0, \ldots, r_n$,
the <u>**sequence of states**</u> in the **computation**, where:

- $r_0 = q_0$
- M ***accepts*** w if
  sequence of states $r_0, r_1, \ldots, r_n$ in Q exists . . .
  with $r_n \in F$

$\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***

DFA Computation Rules

Informally

Given

- A **DFA** (~ a “Program”)
- and **Input** = **string of chars**, e.g. “1101”

To **run** the automata / “program”:

- Start in **“start state”**
- Repeat:
    - **Read 1 char** from **input;**
    - **Change state** according to the **transition table**
- Result of computation =
    - **Accept** if last state is **Accept state**
    - **Reject** otherwise

Formally (i.e., mathematically)

- $M = (Q, \Sigma, \boxed{\delta}, q_0, F)$
- $w = \boxed{w_1} w_2 \cdots w_n$

A **run** is represented by **variables** $r_0, \ldots, r_n$, the **sequence of states** in the **computation,** where:

- $\boxed{r_0} = q_0$
- $r_i =$

if i=1, $r_1 = \delta(r_0, w_1)$

if i=2, $r_2 = \delta(r_1, w_2)$

if i=3, $r_3 = \delta(r_2, w_3)$

- M ***accepts*** w if sequence of states $r_0, r_1, \ldots, r_n$ in Q exists … with $r_n \in F$

$\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***

DFA Computation Rules

Informally

Given

- A **DFA** (~ a "Program")
- and <u>Input</u> = **string of chars**, e.g. "`1101`"

To **run** the automata / "program":

- <u>Start</u> in **"start state"**

- <u>**Repeat**</u>:
  - <u>Read</u> **1 char** from **input**;
  - <u>Change</u> **state** according to the <u>transition</u> **table**

- <u>Result</u> **of computation** =
  - **Accept** if last state is **Accept state**
  - **Reject** otherwise

Formally *(i.e., mathematically)*

- $M = (Q, \Sigma, \delta, q_0, F)$
- $w = w_1 w_2 \cdots w_n$

A **run** is represented by **variables** r_0, …, r_n , the <u>**sequence of states**</u> in the **computation**, where:

- $r_0 = q_0$

- $r_i = \delta(r_{i-1}, w_i)$, for $i = 1, \ldots, n$

- M ***accepts*** w if sequence of states $r_0, r_1, \ldots, r_n$ in Q exists … with $r_n \in F$

$\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***

DFA Computation Rules

Informally

Given

- A **DFA** (~ a "Program")
- and **Input** = **string of chars**, e.g. "1101"

To **run** the automata / "program":

- **Start** in **"start state"**
- **Repeat:**
  - **Read** **1 char** from **input;**
  - **Change** **state** according to the **transition** **table**
- **Result** **of computation** =
  - **Accept** if last state is **Accept state**
  - **Reject** otherwise

Formally (i.e., mathematically)

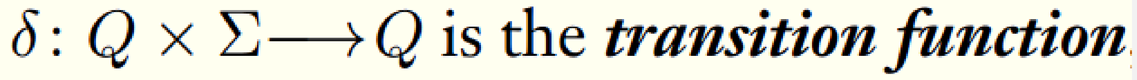
- $M = (Q, \Sigma, \delta, q_0, F)$
- $w = w_1 w_2 \cdots w_n$

This is still a little "informal"

A **run** is represented by **variables** $r_0, \ldots, r_n$, the **sequence of states** in the **computation**, where:

- $r_0 = q_0$
- $r_i = \delta(r_{i-1}, w_i)$, for $i = 1, \ldots, n$
- M ***accepts*** w if sequence of states $r_0, r_1, \ldots, r_n$ in Q exists … with $r_n \in F$

This is still a little "informal"


$\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***

An Extended Transition Function

Define **extended transition function:** $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$

set of pairs

* = “0 or more”

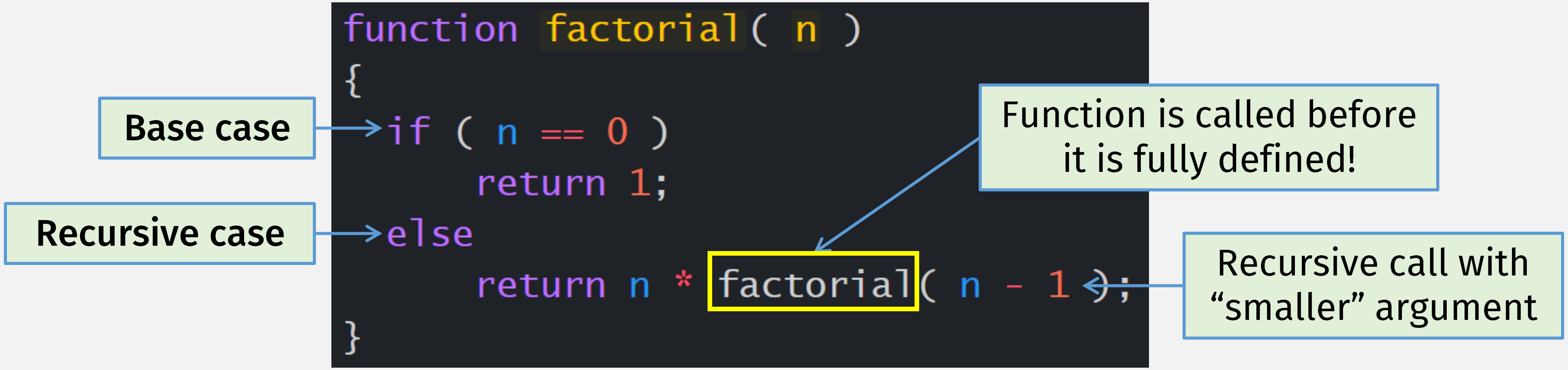
Σ^* = set of all possible strings!

- <u>Domain</u>:
  - Input state $q \in Q$ (doesn’t have to be start state)
  - Input string $w = w_1 w_2 \cdots w_n$ where $w_i \in \Sigma$
- <u>Range</u>:
  - Output state (doesn’t have to be an accept state)

(Defined recursively)

- <u>Base case</u>: …

Interlude: Recursive Definitions



```
function factorial( n )
{
  if ( n == 0 )
      return 1;
  else
      return n * factorial( n - 1 );
}
```

Base case

Recursive case

Function is called before it is fully defined!

Recursive call with “smaller” argument

- Why is this <u>allowed</u>?
  - It’s a “**feature**” (i.e., an **axiom!**) of the **programming language**
- Why does this “<u>work</u>”? (Why doesn’t it loop forever?)
  - Because the **recursive call always has** a **“smaller” argument …**
  - … and so **eventually reaches** the **base case** and **stops**

Recursive Definitions

A **Natural Number** is either:

- **Zero**, or
- the **Successor** of a **Natural Number**

Base case

Recursive case

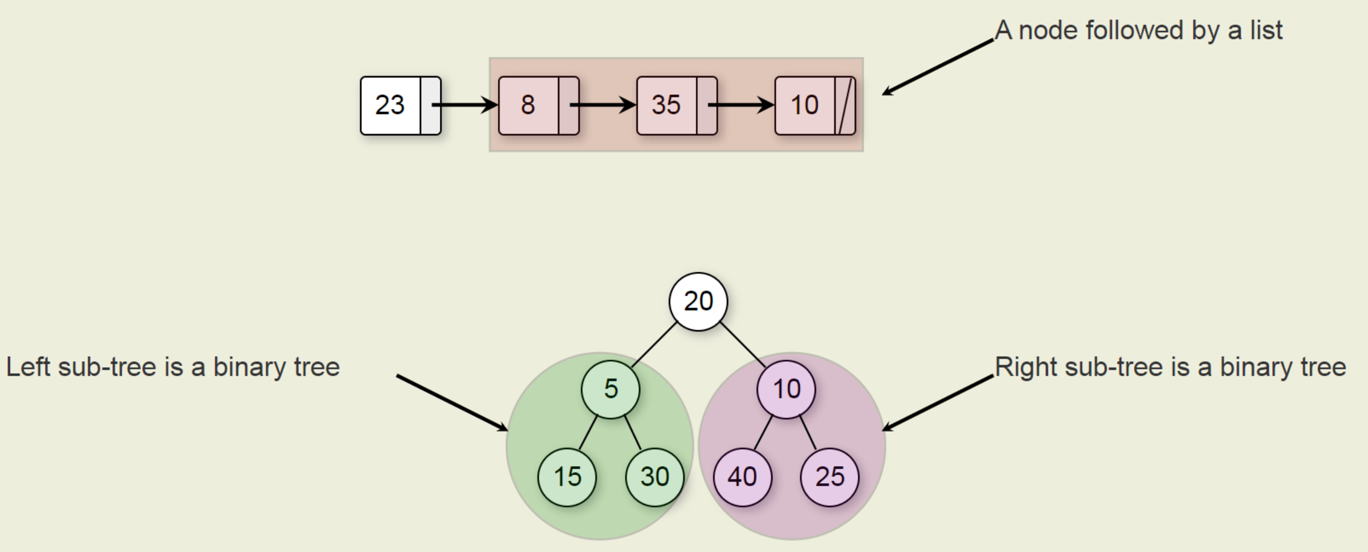
Use of definition before it is fully defined!

“smaller” argument

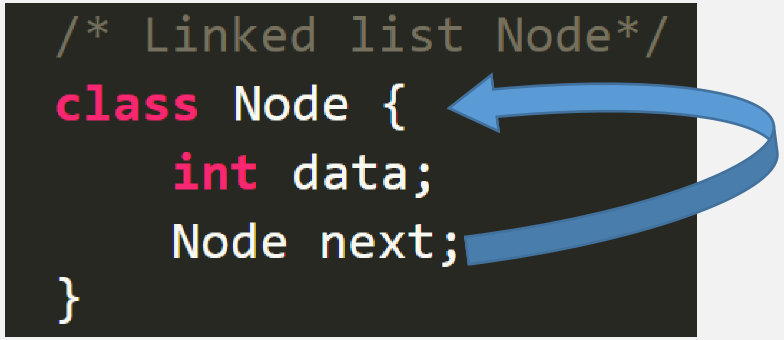
Examples

- **Zero**
- **Successor** of **Zero** (= “one”)
- **Successor** of **Successor** of **Zero** (= “two”)
- **Successor** of **Successor** of **Successor** of **Zero** (= “three”) …

Recursive Definitions



Recursive definitions have:
- <u>base case</u> and
- <u>recursive case</u>
  (with a "smaller" object)



```
/* Linked list Node*/
class Node {
    int data;
    Node next;
}
```

This is a <u>recursive definition:</u> **Node is used before it is fully defined** (but must be "smaller")

Strings Are Defined Recursively

A **String** is either:

- the **empty string** (ε), or
- xa (non-empty string) where
  - x is a **string**
  - a is a “char” in Σ

Base case

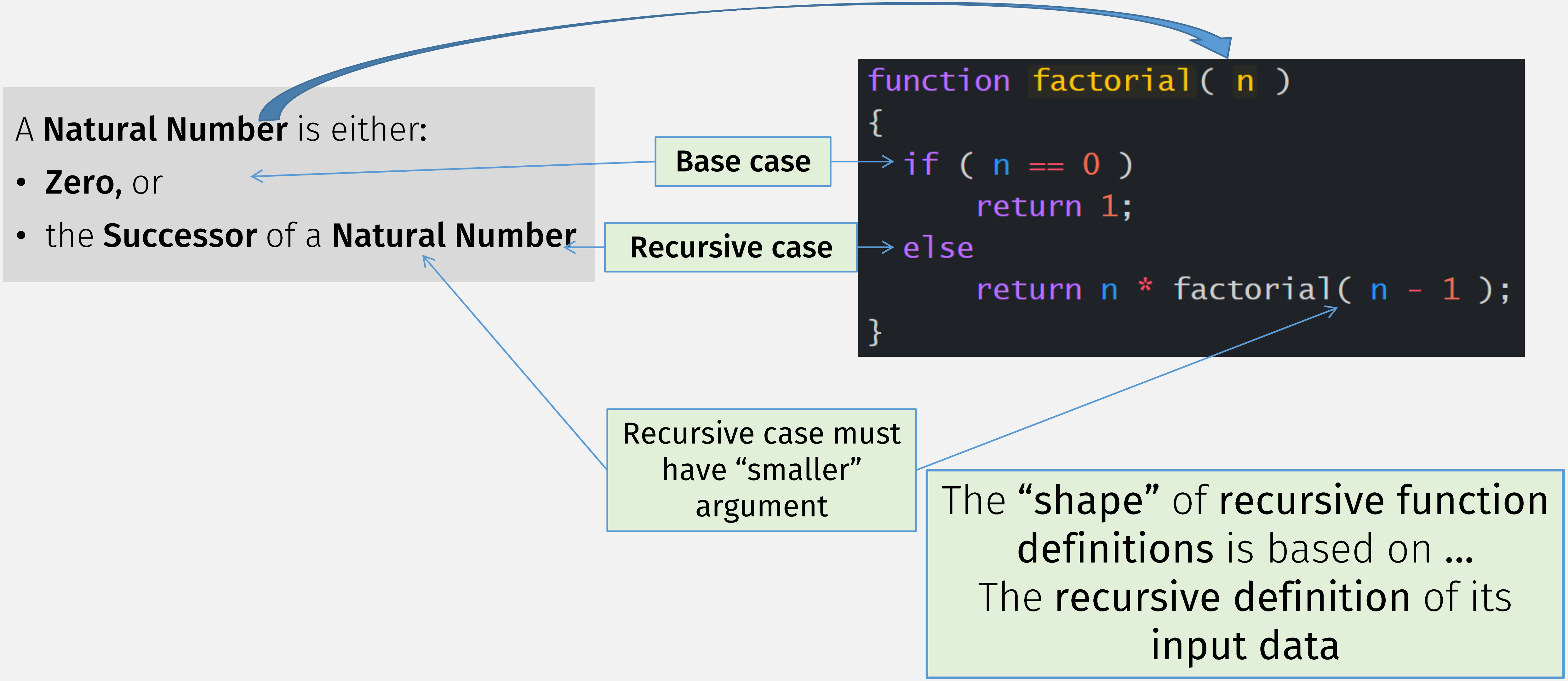
Recursive case

“smaller” argument

Remember: all strings are formed with “chars” from some **alphabet** set Σ

Σ^* = set of all possible strings!

Recursive Functions ⇔ Recursive Data



A **Natural Number** is either:

- **Zero**, or
- the **Successor** of a **Natural Number**

Base case

Recursive case

```
function factorial( n )
{
  if ( n == 0 )
      return 1;
  else
      return n * factorial( n - 1 );
}
```

Recursive case must have “smaller” argument

The **“shape”** of **recursive function definitions** is based on **…**
The **recursive definition** of its **input data**

An Extended Transition Function

Define **extended transition function:** $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$

- Domain:
  - **Input state** $q \in Q$ (doesn't have to be start state)
  - **Input string** $w = w_1 w_2 \cdots w_n$ where $w_i \in \Sigma$
- Range:
  - **Output state** (doesn't have to be an accept state)

Recursive Functions
⇔
Recursive Data

Base case

A **String** is either:

- the **empty string** (ε), or
- *xa* (non-empty string) where
  - *x* is a **string**
  - *a* is a "**char**" in Σ

(Defined recursively)

- Base case $\hat{\delta}(q, \varepsilon) =$

An Extended Transition Function

Define **extended transition function:** $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$

- <u>Domain</u>:
  - **Input state** $q \in Q$ (doesn't have to be start state)
  - **Input string** $w = w_1 w_2 \cdots w_n$ where $w_i \in \Sigma$
- <u>Range</u>:
  - **Output state** (doesn't have to be an accept state)

Recursive Functions
⇔
Recursive Data

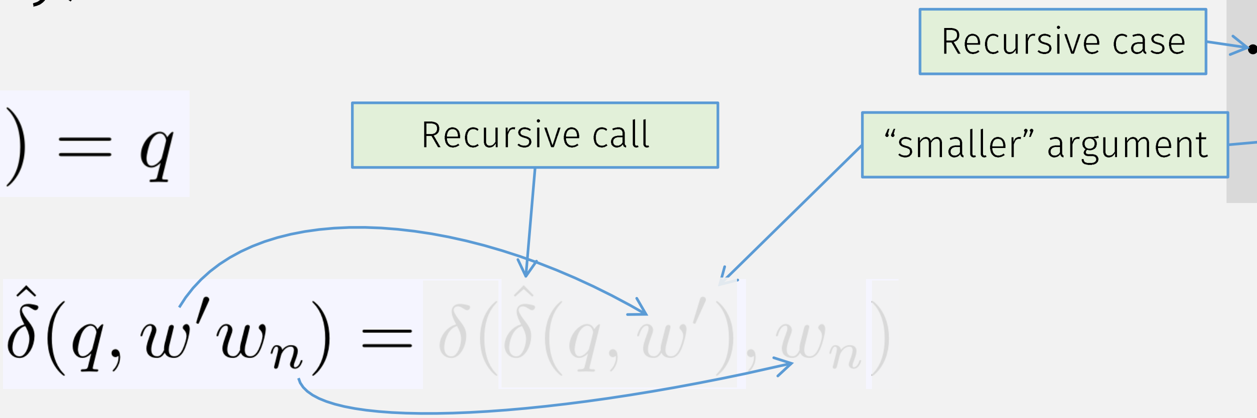
A **String** is either:

- the **empty string** (ε), or
- *xa* (non-empty string) where
  - *x* is a **string**
  - *a* is a "char" in Σ

(Defined recursively)

- Base case $\hat{\delta}(q, \varepsilon) = q$
- Recursive Case $\hat{\delta}(q, w' w_n) = \delta(\hat{\delta}(q, w'), w_n)$

where $w' = w_1 \cdots w_{n-1}$



$\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***

An Extended Transition Function

Define **extended transition function:** $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$

- <u>Domain</u>:
  - **Input state** $q \in Q$ (doesn't have to be start state)
  - **Input string** $w = w_1 w_2 \cdots w_n$ where $w_i \in \Sigma$
- <u>Range</u>:
  - **Output state** (doesn't have to be an accept state)

Recursive Functions
⇔
Recursive Data

A **String** is either:

- the **empty string** (ε), or
- *xa* **(non-empty string)** where
  - *x* is a **string**
  - *a* is a "**char**" in Σ

(Defined recursively)

- Base case $\hat{\delta}(q, \varepsilon) = q$
- Recursive Case $\hat{\delta}(q, w'w_n) = \delta(\hat{\delta}(q, w'), w_n)$

where $w' = w_1 \cdots w_{n-1}$

Previously

DFA Computation Rules

Informally

Given

- A **DFA** (~ a “Program”)
- and Input = **string of chars**, e.g. “1101”

To **run** the automata / “program”:

- Start in **“start state”**
- Repeat:
  - Read **1 char** from **input;**
  - Change **state** according to the transition **table**
- Result of computation =
  - **Accept** if last state is **Accept state**
  - **Reject** otherwise

Formally (i.e., mathematically)

- $M = (Q, \Sigma, \delta, q_0, F)$
- $w = w_1 w_2 \cdots w_n$

A **run** is represented by **variables** $r_0, \ldots, r_n$,
the sequence of states in the **computation,** where:

- $r_0 = q_0$
- $r_i = \delta(r_{i-1}, w_i)$, for $i = 1, \ldots, n$
- M ***accepts*** w if
  sequence of states $r_0, r_1, \ldots, r_n$ in Q exists …
  with $r_n \in F$

This is still a little “informal”

DFA Computation Rules

Informally

Given

- A **DFA** (~ a "Program")
- and Input = **string of chars**, e.g. "`1101`"

To **run** the automata / "program":

- Start in **"start state"**
- Repeat:
  - Read **1 char** from **input;**
  - Change **state** according to the transition **table**
- Result of computation =
  - **Accept** if last state is **Accept state**
  - **Reject** otherwise

Formally (i.e., mathematically)

- $M = (Q, \Sigma, \delta, q_0, F)$
- $w = w_1 w_2 \cdots w_n$

A **run** is represented by **variables** $r_0, \ldots, r_n$,
the sequence of states in the **computation,** where:

- $r_0 = q_0$
- $r_i = \delta(r_{i-1}, w_i)$, for $i = 1, \ldots, n$
- M ***accepts*** w if $\hat{\delta}(q_0, w) \in F$

sequence of states $r_0, r_1, \ldots, r_n$ in Q exists …

with $r_n \in F$

Definition of Accepting Computations

An **accepting computation**, for DFA $M = (Q, \Sigma, \delta, q_0, F)$ and **string** w:

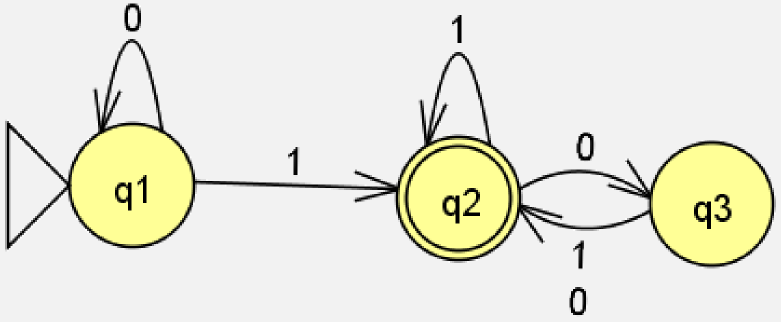
1. **starts** in the **start state** q_0
2. **goes through** a **valid sequence** of **states according to** δ
3. **ends** in an **accept state**

<u>All 3</u> **must be true** for a **computation** to be an **accepting computation!**

M ***accepts*** w if $\hat{\delta}(q_0, w) \in F$

Accepting Computation or Not?

DFA:



- $\hat{\delta}$(q1, **`1101`**)
  - Yes
- $\hat{\delta}$(q1, **`110`**)
  - **No** (doesn't end in accept state)
- $\hat{\delta}$(q2, **`101`**)
  - **No** (doesn't start in start state)

Alphabets, Strings, Languages

Alphabet specifies “all possible strings”

- An **alphabet** is a non-empty finite set of symbols

(impossible to have strings with non-alphabet chars)

$$\Sigma_1 = \{0,1\}$$

$$\Sigma_2 = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}, \mathtt{g}, \mathtt{h}, \mathtt{i}, \mathtt{j}, \mathtt{k}, \mathtt{l}, \mathtt{m}, \mathtt{n}, \mathtt{o}, \mathtt{p}, \mathtt{q}, \mathtt{r}, \mathtt{s}, \mathtt{t}, \mathtt{u}, \mathtt{v}, \mathtt{w}, \mathtt{x}, \mathtt{y}, \mathtt{z}\}$$

- A **string** is a finite sequence of **symbols** from an **alphabet**

01001 abracadabra ε ← Empty string (length 0)

- A **language** is a set of **strings**

Languages can be infinite

$$A = \{\mathtt{good}, \mathtt{bad}\}$$

$$\emptyset \qquad \{\ \}$$

Empty set is a language

$$A = \{w \mid w \text{ contains at least one 1 and an even number of 0s follow the last 1}\}$$

“the set of all …” “such that …”

Computation and Languages

- The **language** of a **machine** is the <u>set</u> **of all strings** that it **accepts**
- E.g., A **DFA** M $\boldsymbol{accepts}$ w if $\hat{\delta}(q_0, w) \in F$
- **Language** of $M = L(M) = \{w \mid M \text{ accepts } w\}$

"the set of all …"

"such that …"

Machine and Language Terminology

DFA M ***accepts*** w ← string

M ***recognizes language*** A ← Set of strings

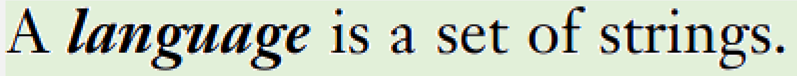
if $A = \{w \mid M \text{ accepts } w\}$

Computation and Classes of Languages

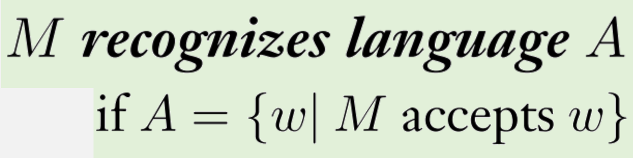
- The **language** of a **machine** = set of **all strings** that it **accepts**
  - E.g., every **DFA** is **associated with** a **language**
- A **computation model** = set of **machines** it **defines**
  - E.g., **all possible DFAs** are a **computation model**
- Thus: a **computation model** = **set** of **languages**

Regular Languages: Definition

If a **deterministic finite automata** (**DFA**) recognizes a language, then **that language** is called a **regular language.**



A ***language*** is a set of strings.



M ***recognizes language*** A

if $A = \{w |\ M \text{ accepts } w\}$

A Language, Regular or Not?

- If given: **a DFA M**
  - We know: **$L(M)$,** the **language recognized by M,** is a **regular language**

If a **DFA** <u>recognizes</u> a **language,**
then **that language** is called a **regular language.**

(modus ponens)

- If given: **a Language A**
  - Is A a regular language?
    - Not necessarily!
  - How do we determine, i.e., *prove*, that A is a regular language?

An Inference Rule: Modus Ponens

Premises

- If P then Q
- P is true

Conclusion

- Q must also be true

Example **Premises**

- If **there is an DFA recognizing language A,** then A **is a regular language**
- **There is an DFA M where $L(M) = A$**

Conclusion

- A is a regular language!

A Language, Regular or Not?

- If given: **a DFA** M
  - We know: $L(M)$, the language recognized by M, is a regular language

If a DFA <u>recognizes</u> a language,
then that language is called a **regular language**.

- If given: a Language A
  - Is A a regular language?
    - Not necessarily!
  - How do we determine, i.e., *prove*, that A is a regular language?

Create an DFA recognizing A!