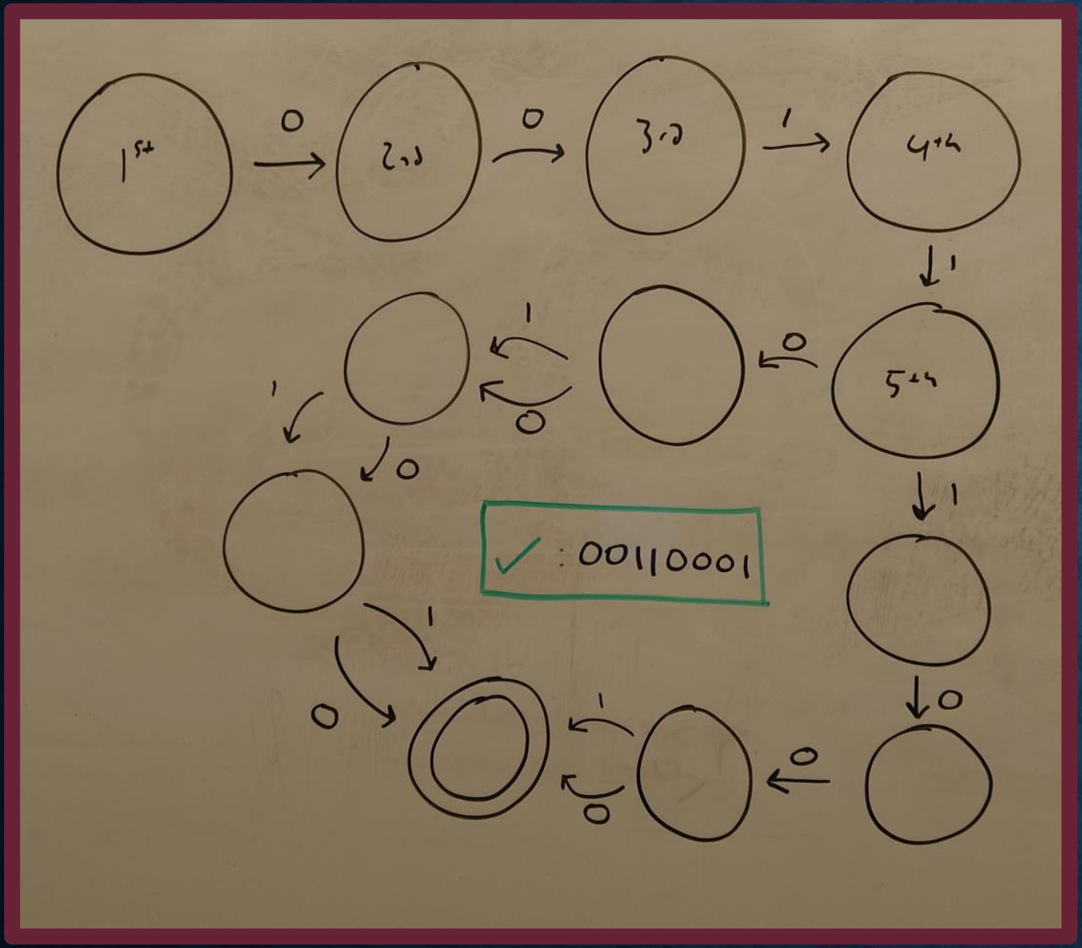
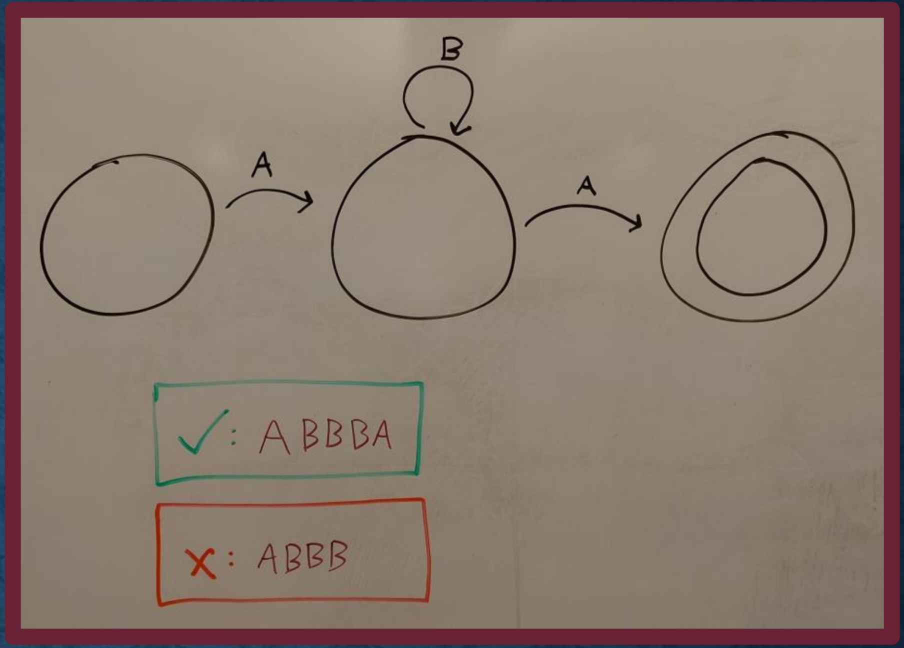
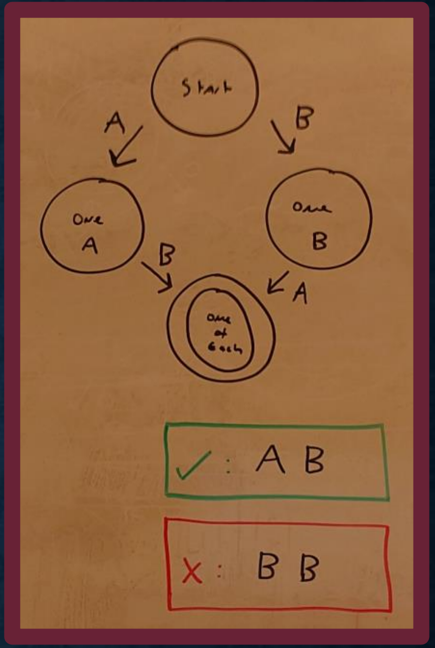
SYNTHESIZING AUTOMATA FROM IMAGES

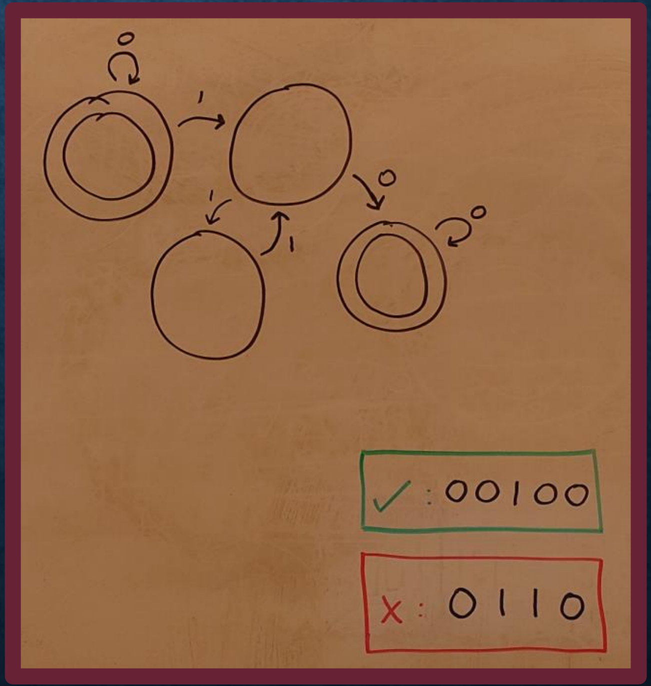
Keith Johnson

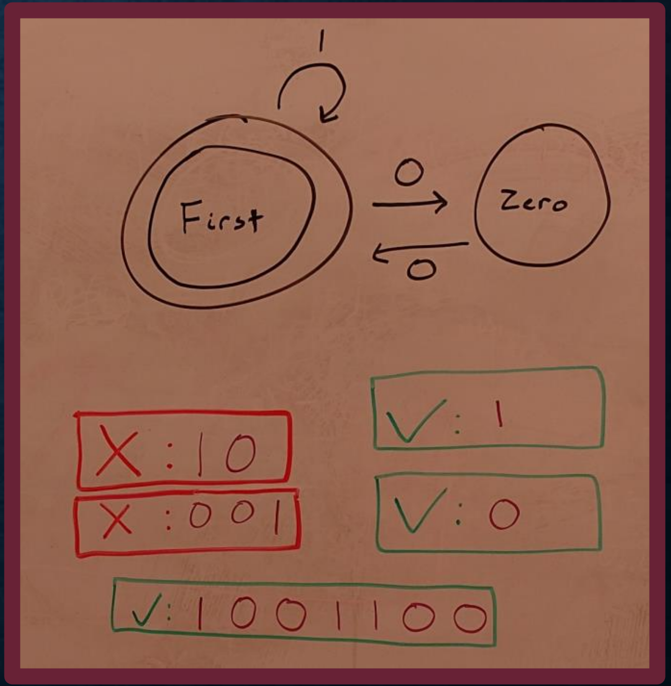
AUTOMATA


1st
0
2nd
0
3rd
1
4th
1
5th
0
1
0
1
0
1
1
0
0
1
0
0
1
0
✓: 00110001

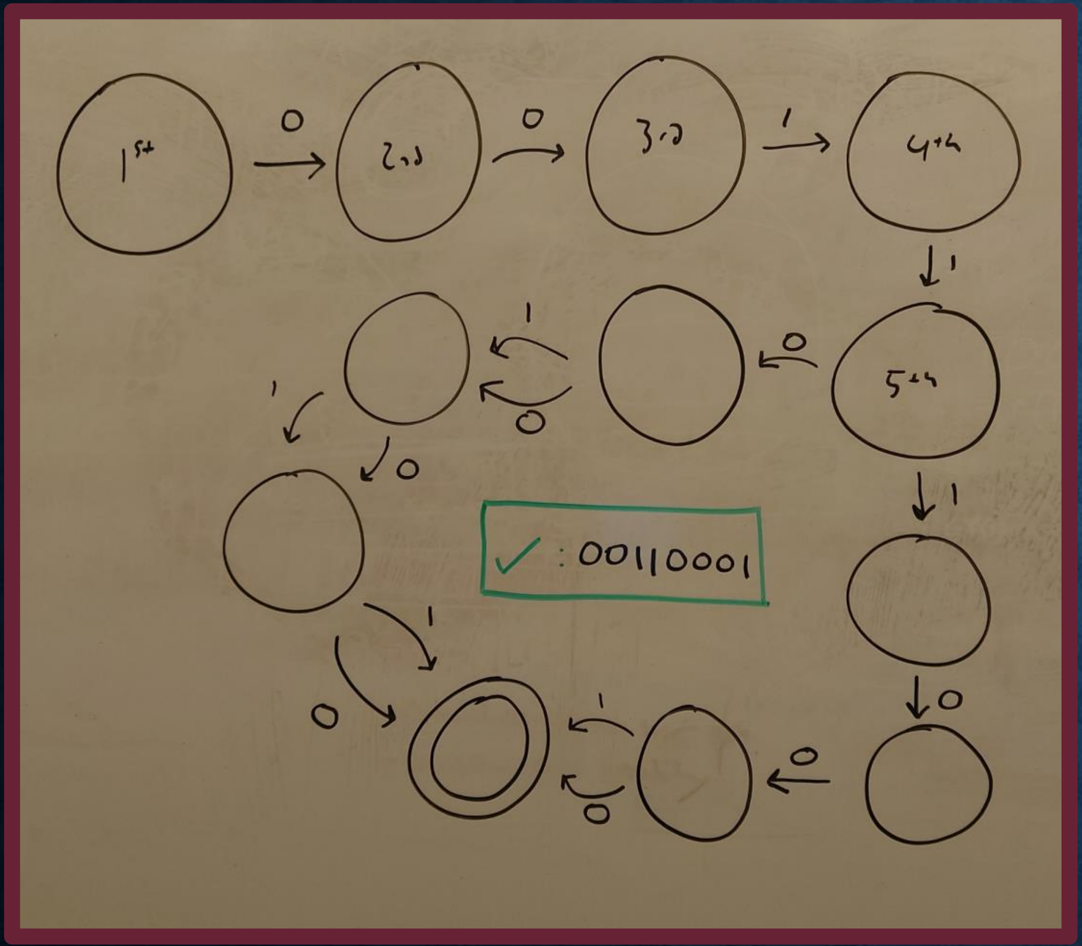

B
A
A
✓: ABBBA
X: ABBB

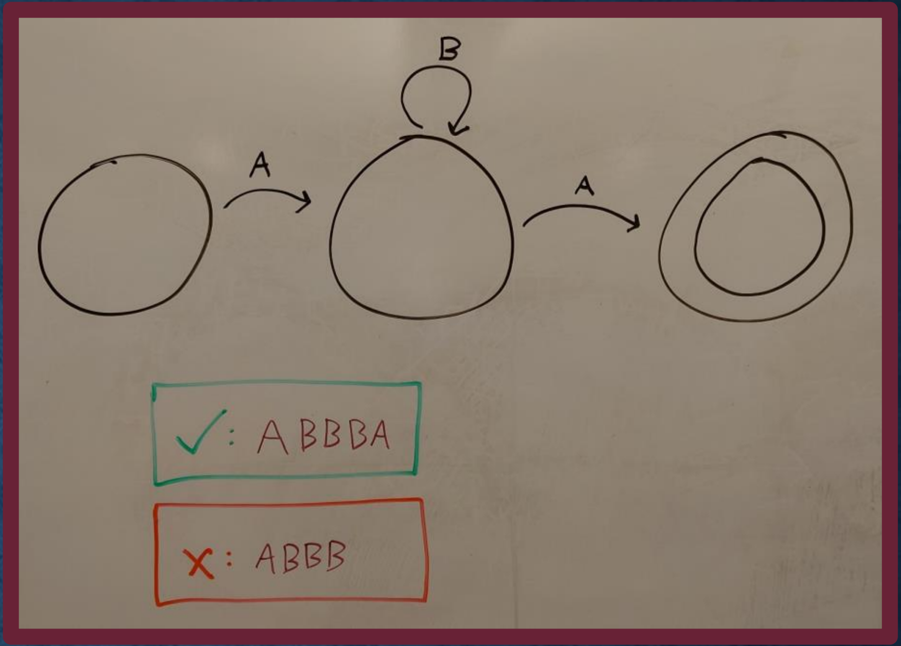

Start
A
B
One A
One B
B
A
One of Each
✓: AB
X: BB

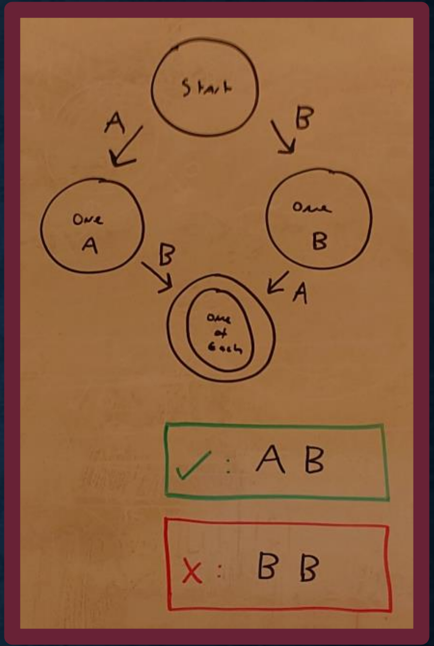

0
1
1
1
0
0
✓: 00100
X: 0110

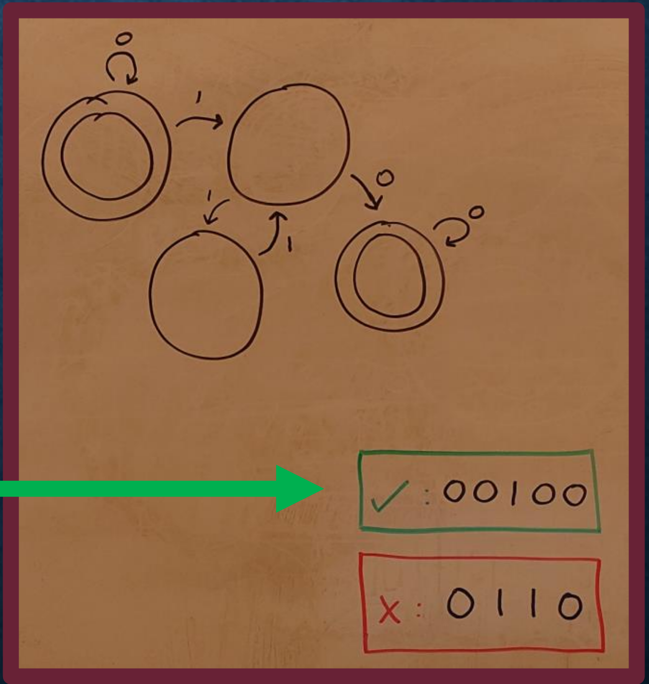

1
First
0
Zero
0
X: 10
✓: 1
X: 001
✓: 0
✓: 1001100

AUTOMATA

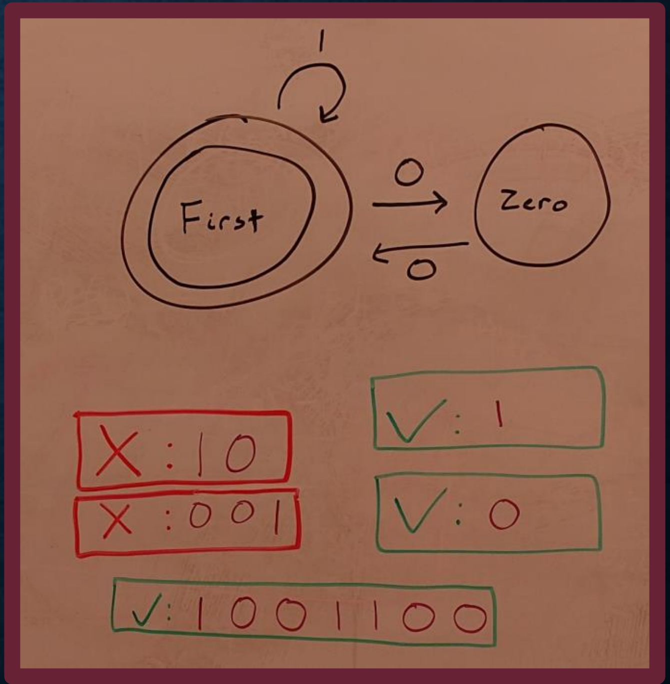




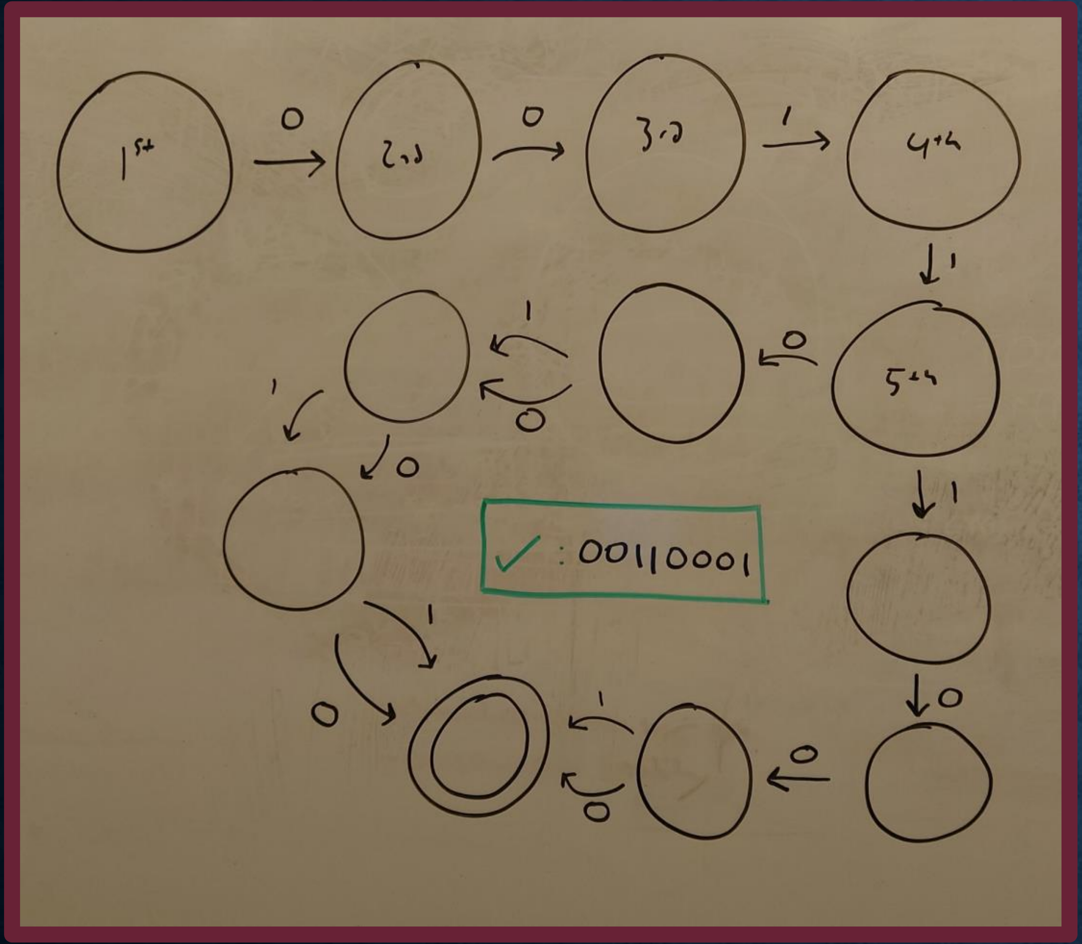


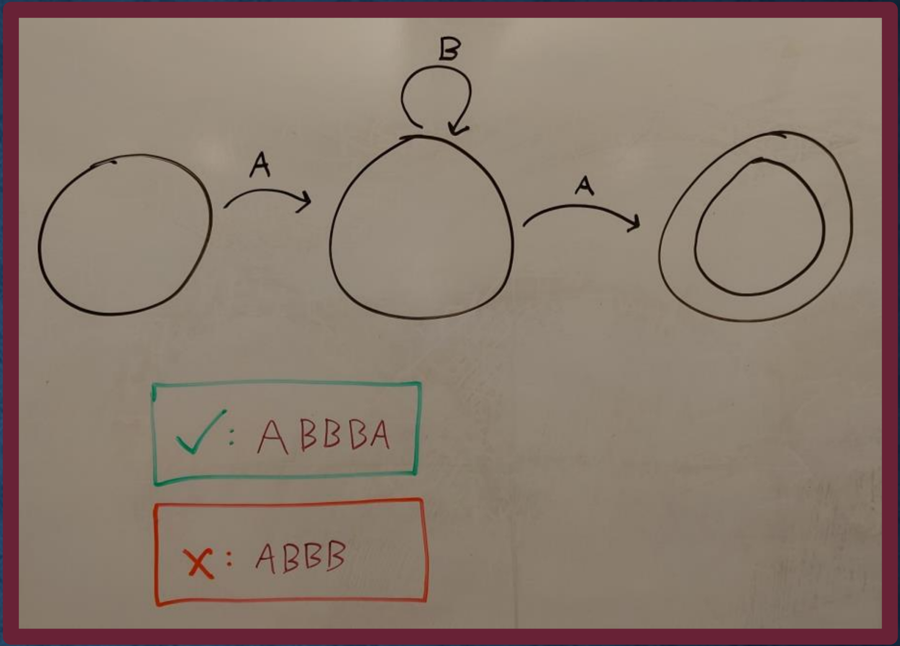


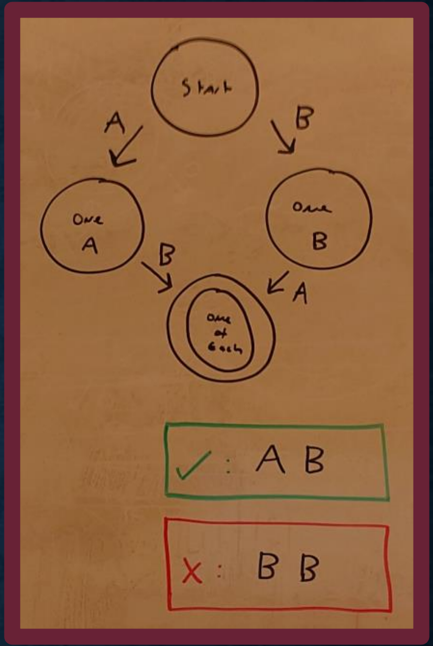
Positive Examples

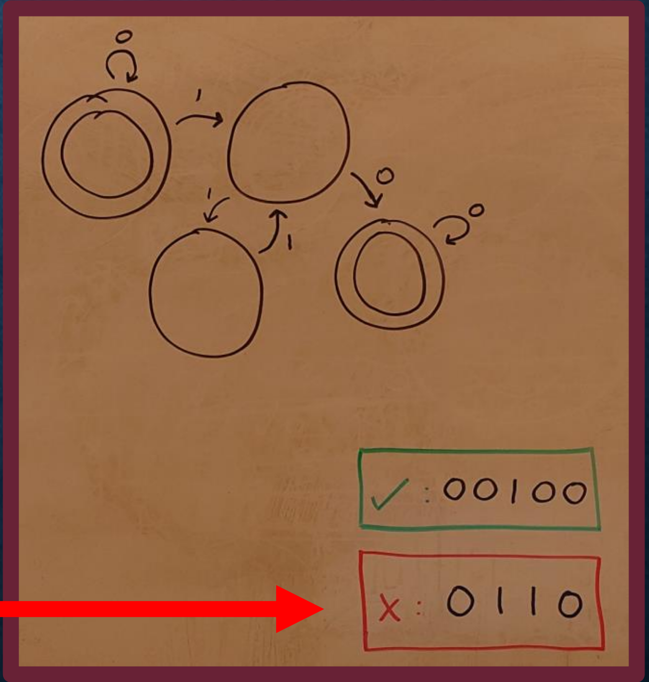


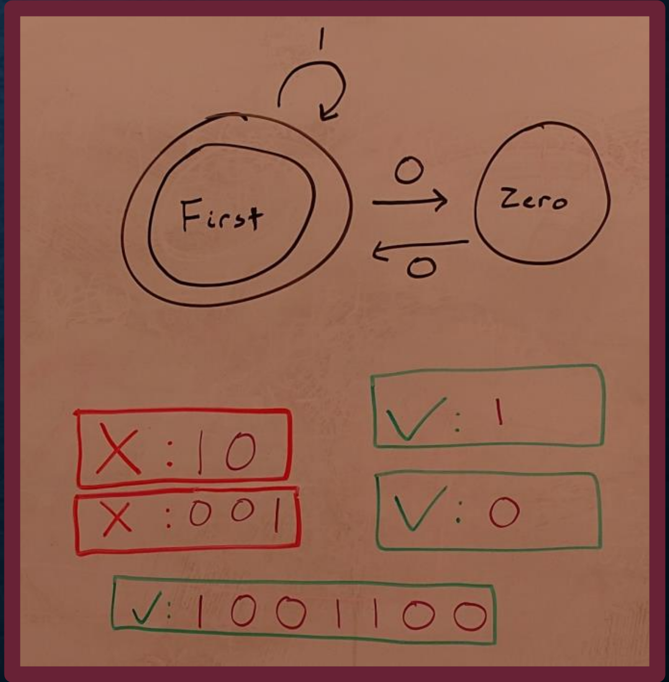
AUTOMATA







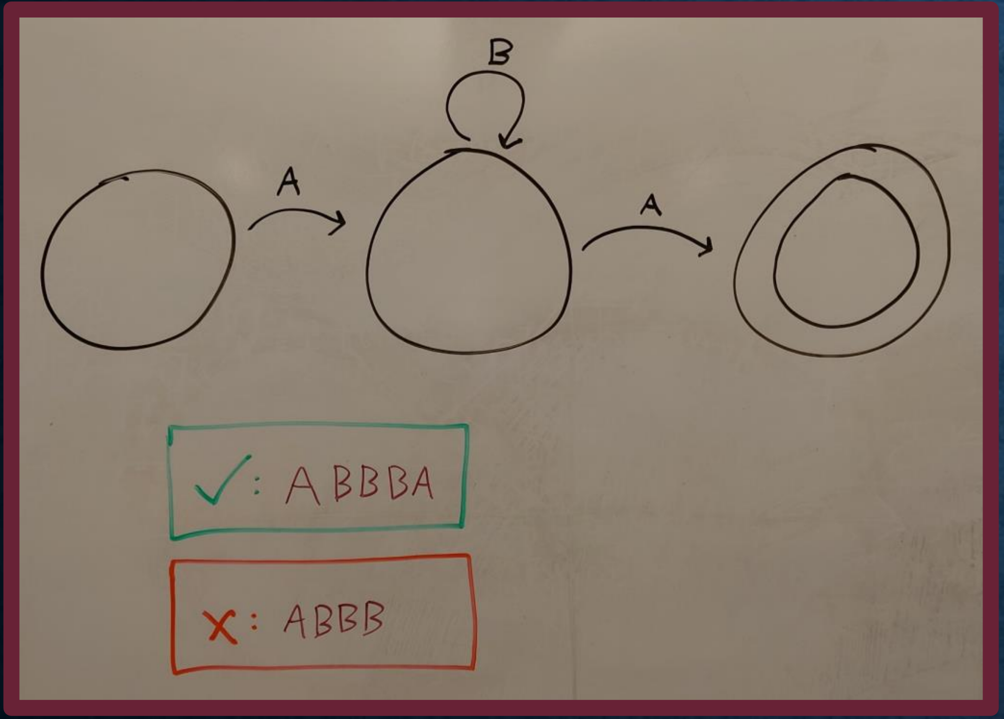




Positive Examples

Negative Examples

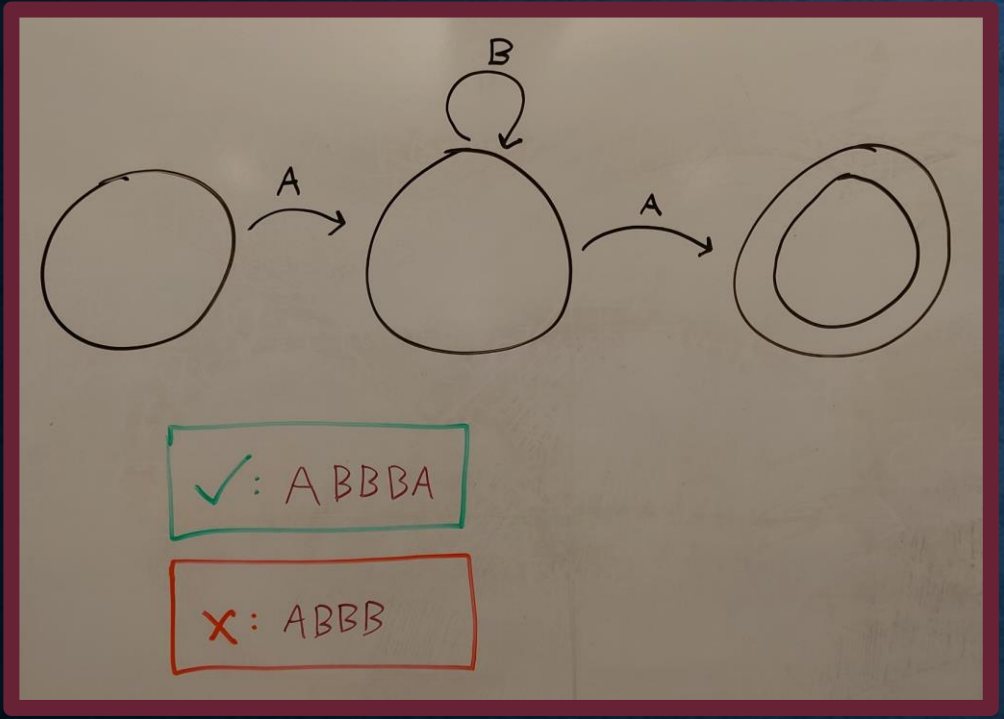
GOAL



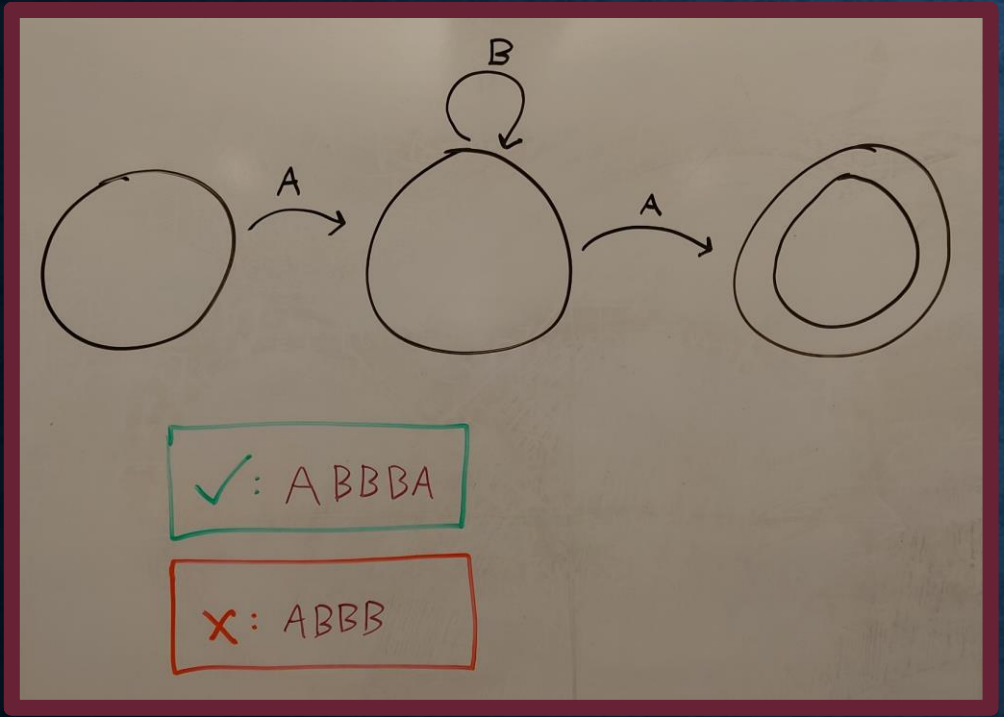
✓: ABBBA

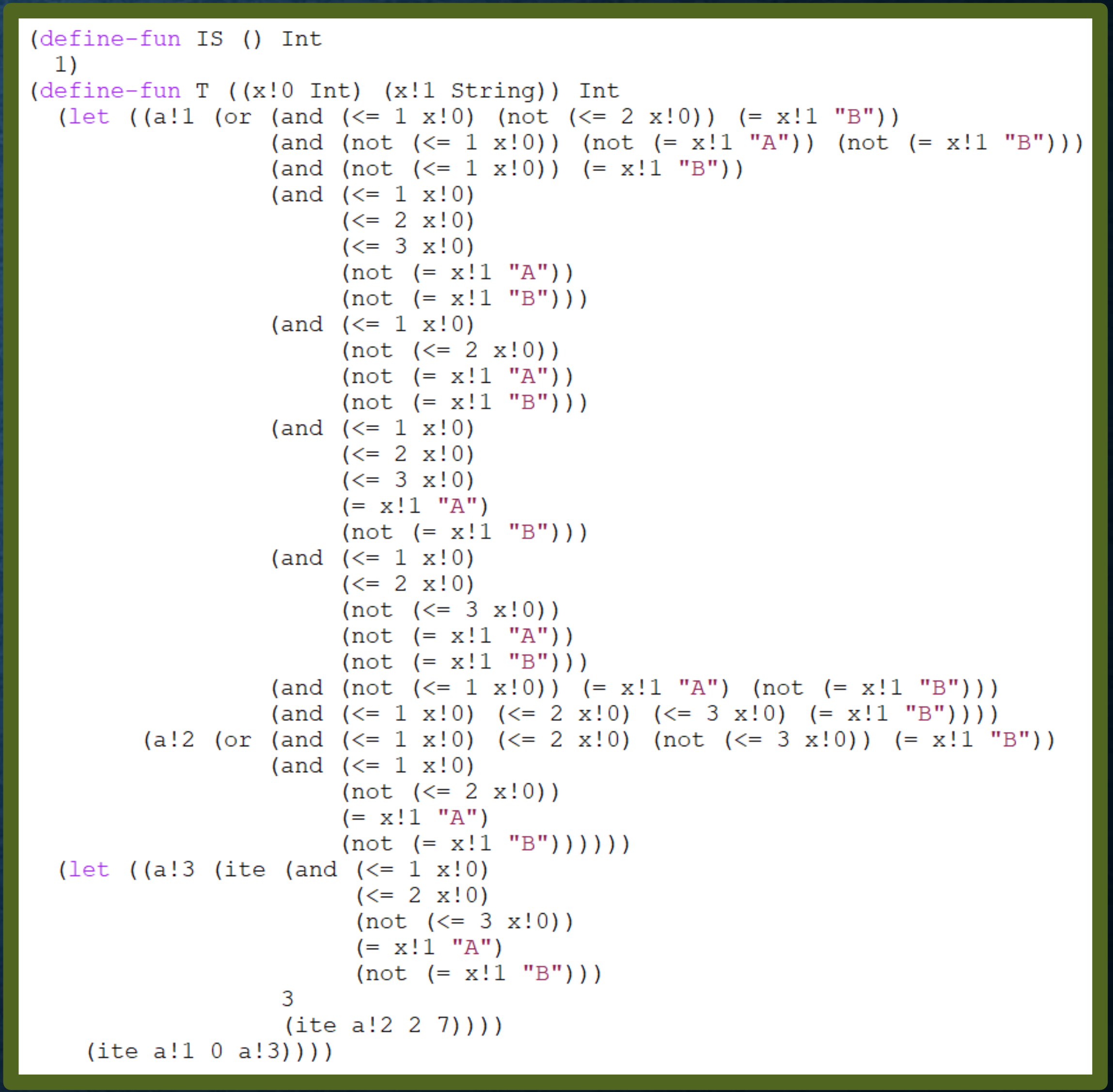
✗: ABBB

GOAL



GOAL

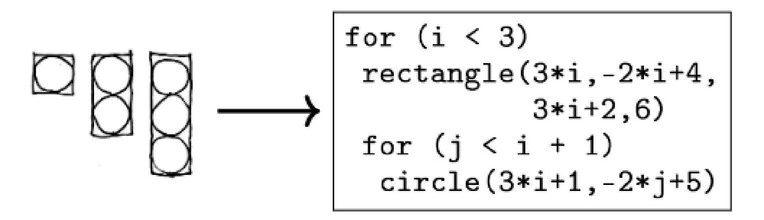




```
(define-fun IS () Int
  1)
(define-fun T ((x!0 Int) (x!1 String)) Int
  (let ((a!1 (or (and (<= 1 x!0) (not (<= 2 x!0)) (= x!1 "B"))
                 (and (not (<= 1 x!0)) (not (= x!1 "A")) (not (= x!1 "B")))
                 (and (not (<= 1 x!0)) (= x!1 "B"))
                 (and (<= 1 x!0)
                      (<= 2 x!0)
                      (<= 3 x!0)
                      (not (= x!1 "A"))
                      (not (= x!1 "B")))
                 (and (<= 1 x!0)
                      (not (<= 2 x!0))
                      (not (= x!1 "A"))
                      (not (= x!1 "B")))
                 (and (<= 1 x!0)
                      (<= 2 x!0)
                      (<= 3 x!0)
                      (= x!1 "A")
                      (not (= x!1 "B")))
                 (and (<= 1 x!0)
                      (<= 2 x!0)
                      (not (<= 3 x!0))
                      (not (= x!1 "A"))
                      (not (= x!1 "B")))
                 (and (not (<= 1 x!0)) (= x!1 "A") (not (= x!1 "B")))
                 (and (<= 1 x!0) (<= 2 x!0) (<= 3 x!0) (= x!1 "B"))))
        (a!2 (or (and (<= 1 x!0) (<= 2 x!0) (not (<= 3 x!0)) (= x!1 "B"))
                 (and (<= 1 x!0)
                      (not (<= 2 x!0))
                      (= x!1 "A")
                      (not (= x!1 "B"))))))
  (let ((a!3 (ite (and (<= 1 x!0)
                       (<= 2 x!0)
                       (not (<= 3 x!0))
                       (= x!1 "A")
                       (not (= x!1 "B")))
                  3
                  (ite a!2 2 7))))
    (ite a!1 0 a!3))))
```

PRIOR ART

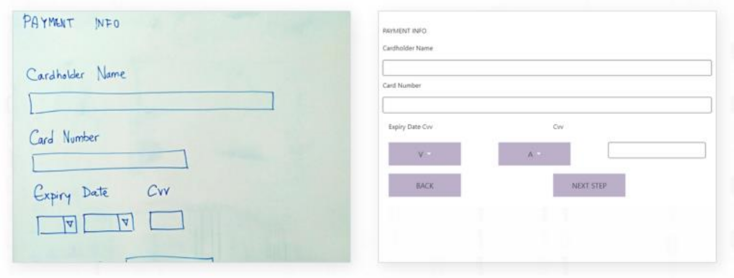
Images as a synthesis specification



```
for (i < 3)
 rectangle(3*i,-2*i+4,
          3*i+2,6)
 for (j < i + 1)
  circle(3*i+1,-2*j+5)
```

- Synthesizing programs operating on images
- Uses the image data as the program input (or output)
- E.g., *Learning to Infer Graphics Programs from Hand-Drawn Images (Ellis, Ritchie, et al., 2018)*

UI Design Tools



- E.g., Microsoft's Sketch2Code
- Focused on generating UI, not executable programs

Arrow-Connected Diagram Recognition

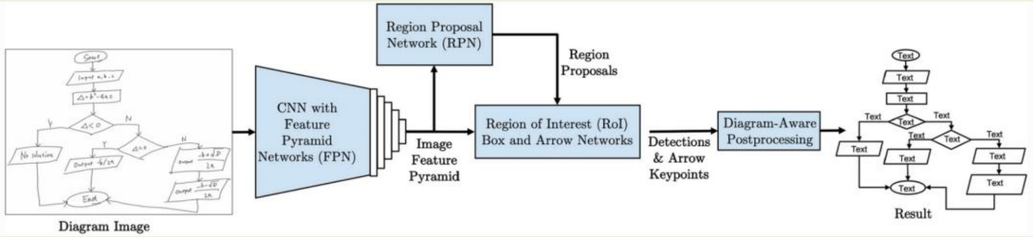
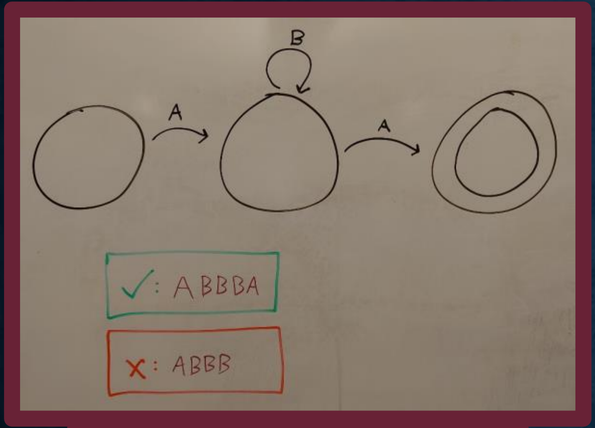


Diagram Image
CNN with Feature Pyramid Networks (FPN)
Image Feature Pyramid
Region Proposal Network (RPN)
Region Proposals
Region of Interest (RoI) Box and Arrow Networks
Detections & Arrow Keypoints
Diagram-Aware Postprocessing
Text
Result

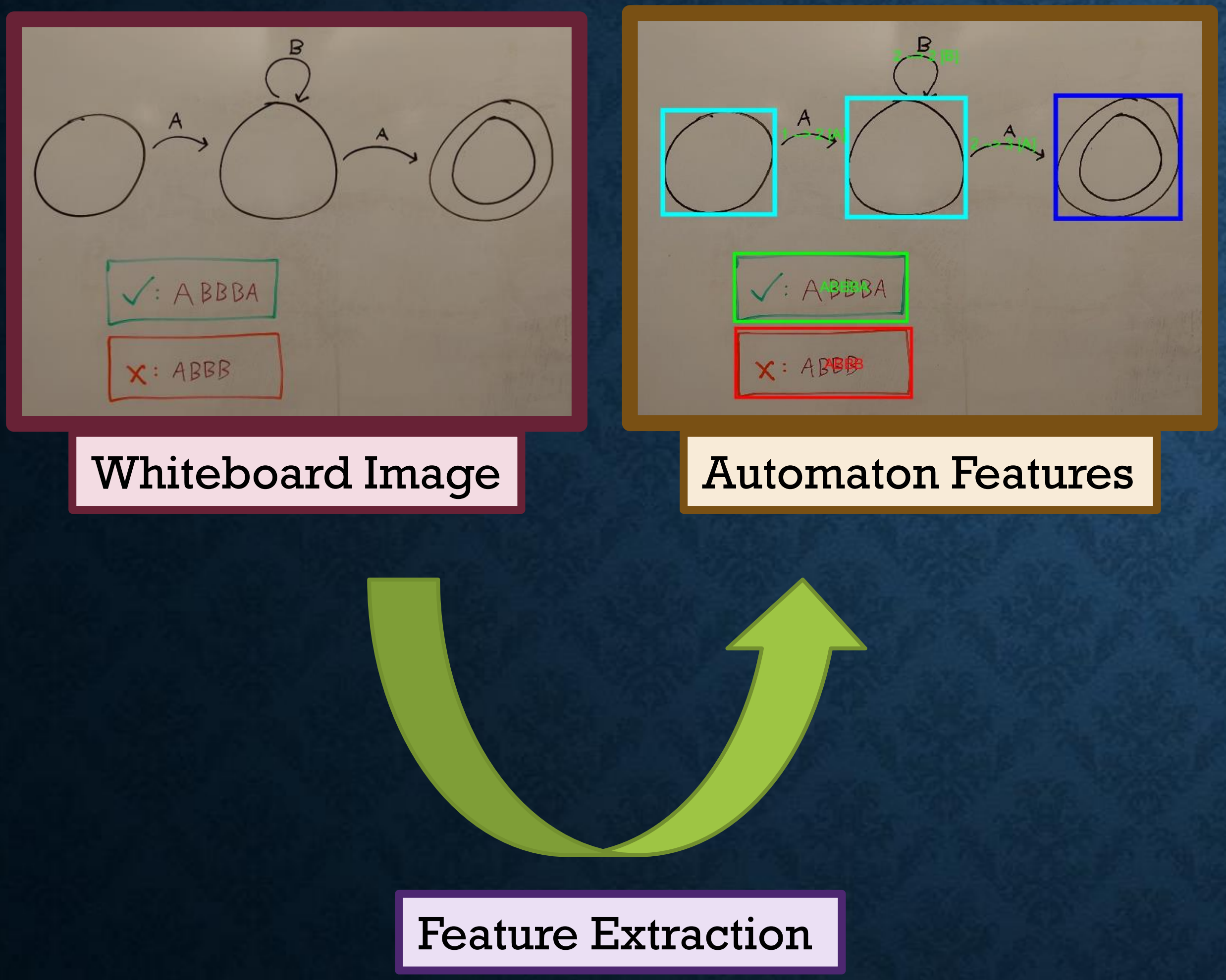
- Uses deep learning to reconstruct diagrams
- Challenge: deep learning is hard to "debug"
- This presents a challenge for program synthesis, where exact accuracy is key
- E.g., *Arrow R-CNN for handwritten diagram recognition (Schäfer, Keuper, and Stuckenschmidt, 2021)*

SYSTEM OVERVIEW



Whiteboard Image

SYSTEM OVERVIEW

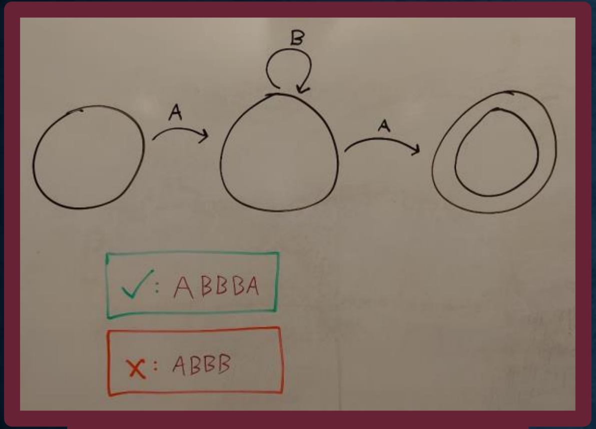


Whiteboard Image

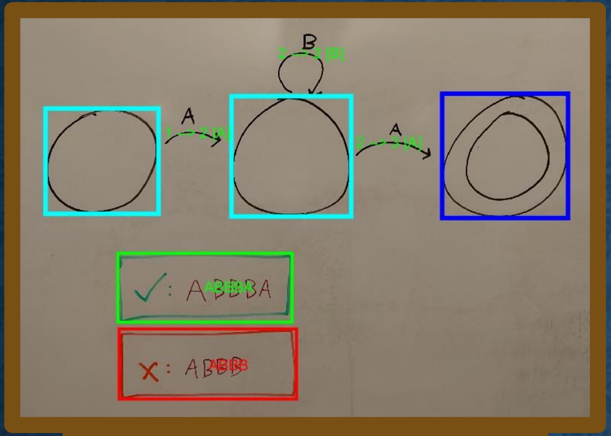
Automaton Features

Feature Extraction

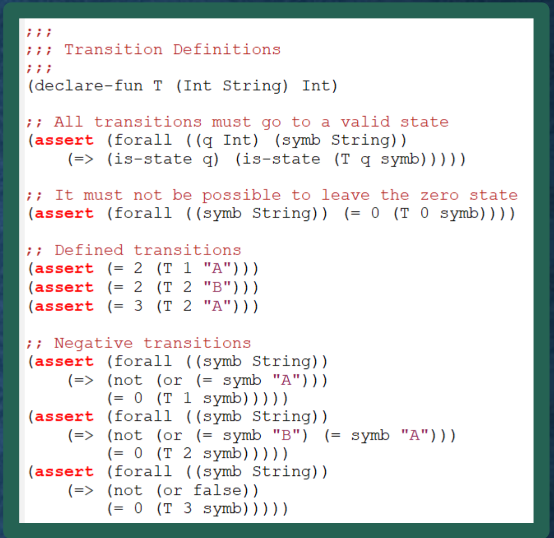
SYSTEM OVERVIEW



Whiteboard Image



Automaton Features



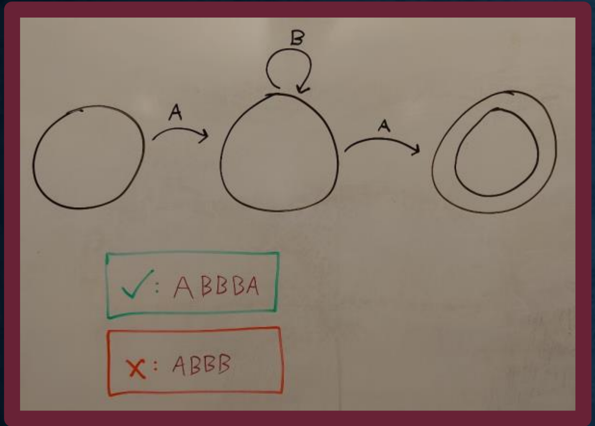
```
;;;
;;; Transition Definitions
;;;
(declare-fun T (Int String) Int)

;; All transitions must go to a valid state
(assert (forall ((q Int) (symb String))
    (=> (is-state q) (is-state (T q symb)))))

;; It must not be possible to leave the zero state
(assert (forall ((symb String)) (= 0 (T 0 symb))))

;; Defined transitions
(assert (= 2 (T 1 "A")))
(assert (= 2 (T 2 "B")))
(assert (= 3 (T 2 "A")))

;; Negative transitions
(assert (forall ((symb String))
    (=> (not (or (= symb "A")))
        (= 0 (T 1 symb)))))
(assert (forall ((symb String))
    (=> (not (or (= symb "B") (= symb "A")))
        (= 0 (T 2 symb)))))
(assert (forall ((symb String))
    (=> (not (or false))
        (= 0 (T 3 symb)))))
```

Program Specification

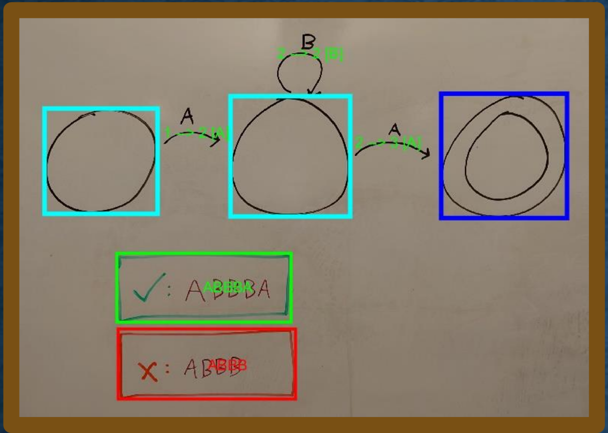
Feature Extraction

SMT-LIB2 Encoding

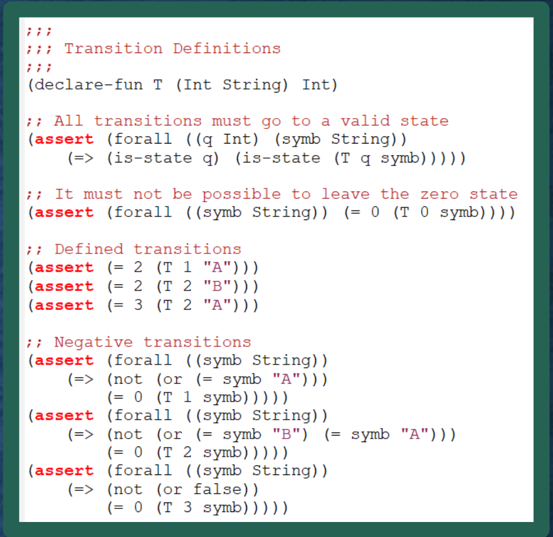
SYSTEM OVERVIEW



Whiteboard Image

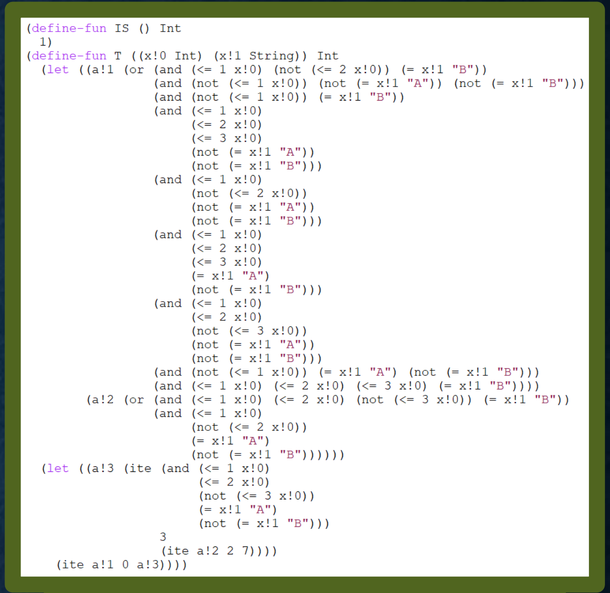


Automaton Features



```
;;;
;;; Transition Definitions
;;;
(declare-fun T (Int String) Int)

;; All transitions must go to a valid state
(assert (forall ((q Int) (symb String))
    (=> (is-state q) (is-state (T q symb)))))

;; It must not be possible to leave the zero state
(assert (forall ((symb String)) (= 0 (T 0 symb))))

;; Defined transitions
(assert (= 2 (T 1 "A")))
(assert (= 2 (T 2 "B")))
(assert (= 3 (T 2 "A")))

;; Negative transitions
(assert (forall ((symb String))
    (=> (not (or (= symb "A")))
        (= 0 (T 1 symb)))))
(assert (forall ((symb String))
    (=> (not (or (= symb "B") (= symb "A")))
        (= 0 (T 2 symb)))))
(assert (forall ((symb String))
    (=> (not (or false))
        (= 0 (T 3 symb)))))
```

Program Specification



```
(define-fun IS () Int
  1)
(define-fun T ((x!0 Int) (x!1 String)) Int
  (let ((a!1 (or (and (<= 1 x!0) (not (<= 2 x!0)) (= x!1 "B"))
                 (and (not (<= 1 x!0)) (not (= x!1 "A")) (not (= x!1 "B")))
                 (and (not (<= 1 x!0)) (= x!1 "B"))
                 (and (<= 1 x!0)
                      (<= 2 x!0)
                      (<= 3 x!0)
                      (not (= x!1 "A"))
                      (not (= x!1 "B")))
                 (and (<= 1 x!0)
                      (not (<= 2 x!0))
                      (not (= x!1 "A"))
                      (not (= x!1 "B")))
                 (and (<= 1 x!0)
                      (<= 2 x!0)
                      (<= 3 x!0)
                      (= x!1 "A")
                      (not (= x!1 "B")))
                 (and (<= 1 x!0)
                      (<= 2 x!0)
                      (not (<= 3 x!0))
                      (not (= x!1 "A"))
                      (not (= x!1 "B")))
                 (and (not (<= 1 x!0)) (= x!1 "A") (not (= x!1 "B")))
                 (and (<= 1 x!0) (<= 2 x!0) (<= 3 x!0) (= x!1 "B"))))
        (a!2 (or (and (<= 1 x!0) (<= 2 x!0) (not (<= 3 x!0)) (= x!1 "B"))
                 (and (<= 1 x!0)
                      (not (<= 2 x!0))
                      (= x!1 "A")
                      (not (= x!1 "B"))))))
  (let ((a!3 (ite (and (<= 1 x!0)
                       (<= 2 x!0)
                       (not (<= 3 x!0))
                       (= x!1 "A")
                       (not (= x!1 "B")))
                 3
                 (ite a!2 2 7))))
    (ite a!1 0 a!3))))
```

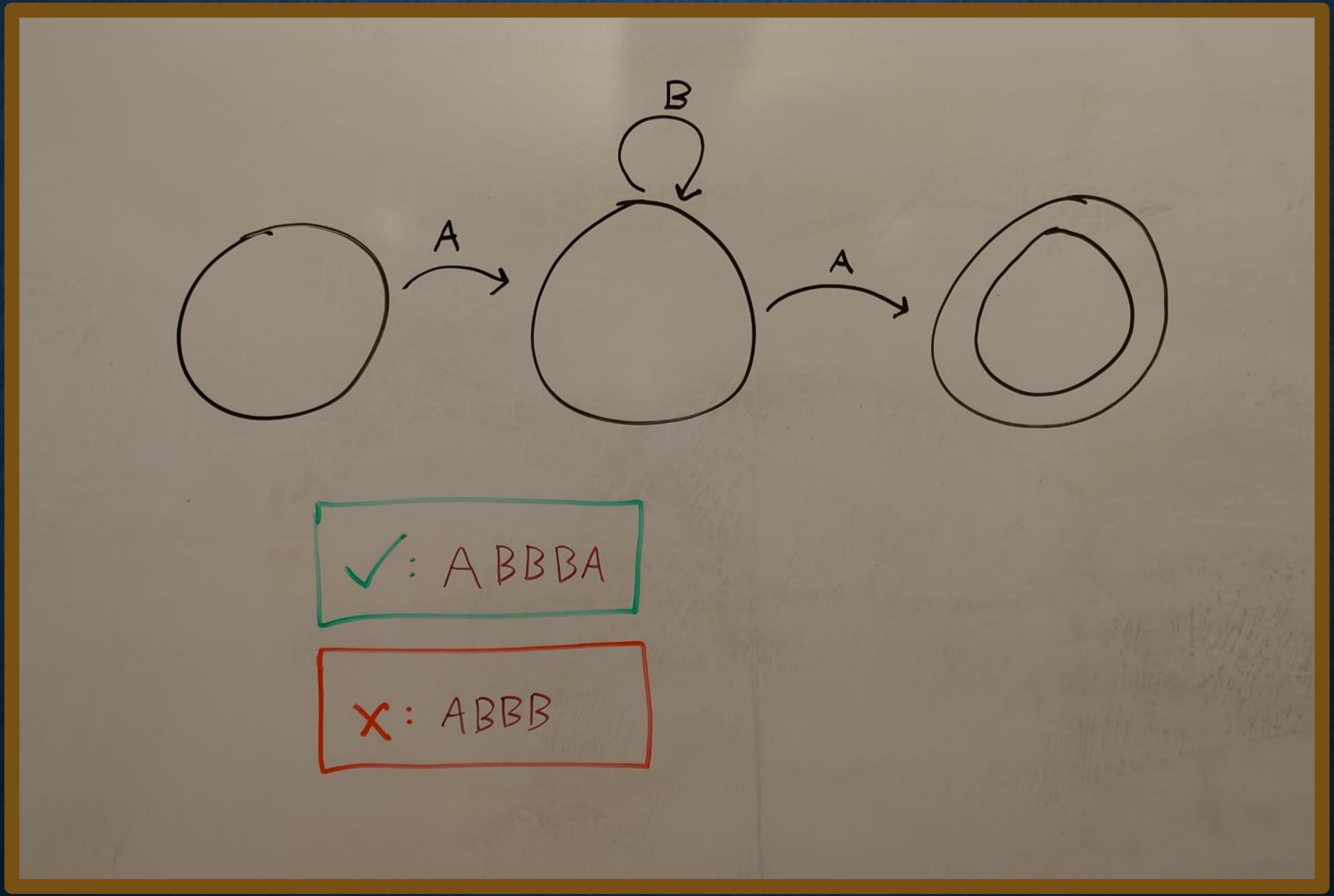
Synthesized Program

Feature Extraction

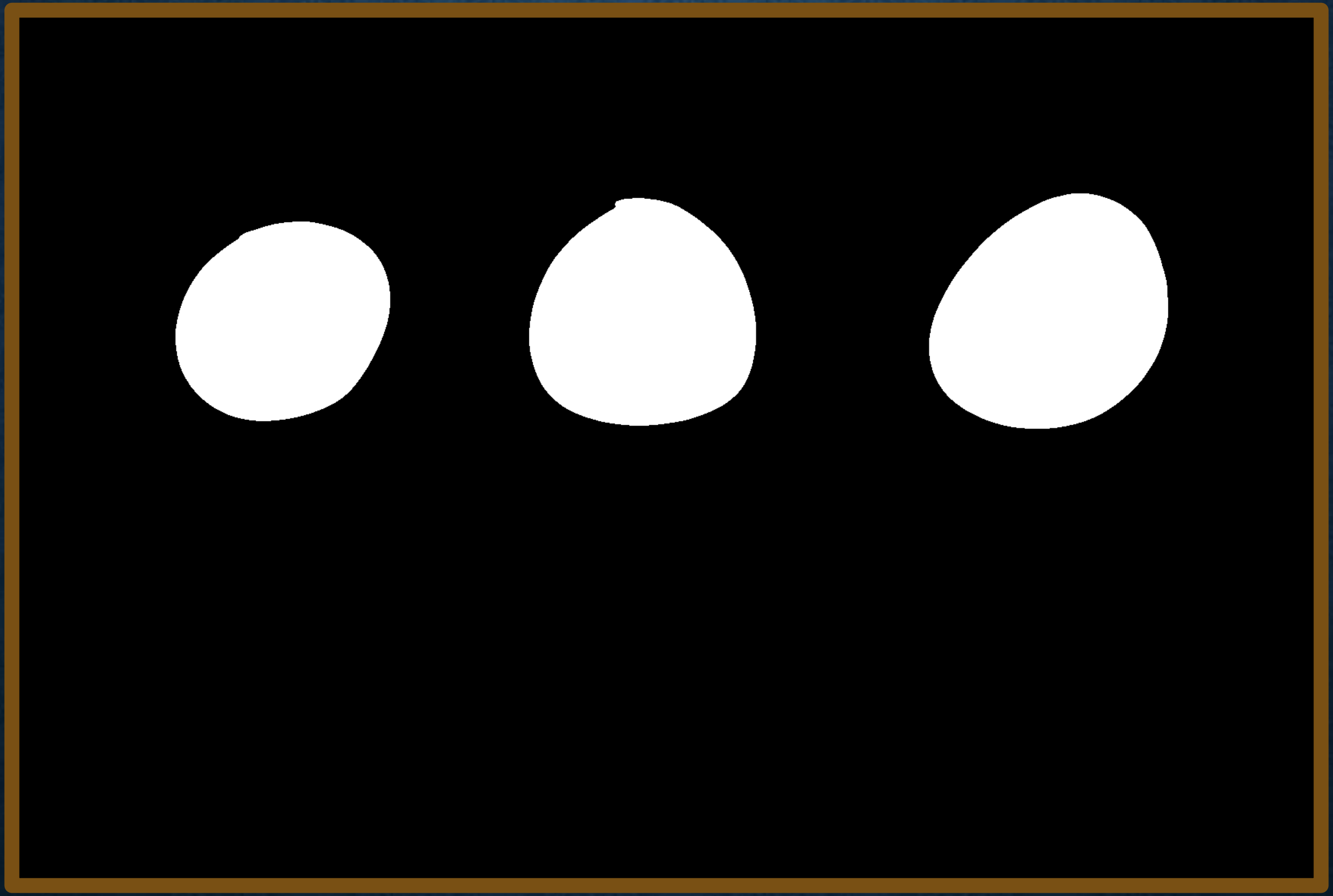
SMT-LIB2 Encoding

Satisfiability Check

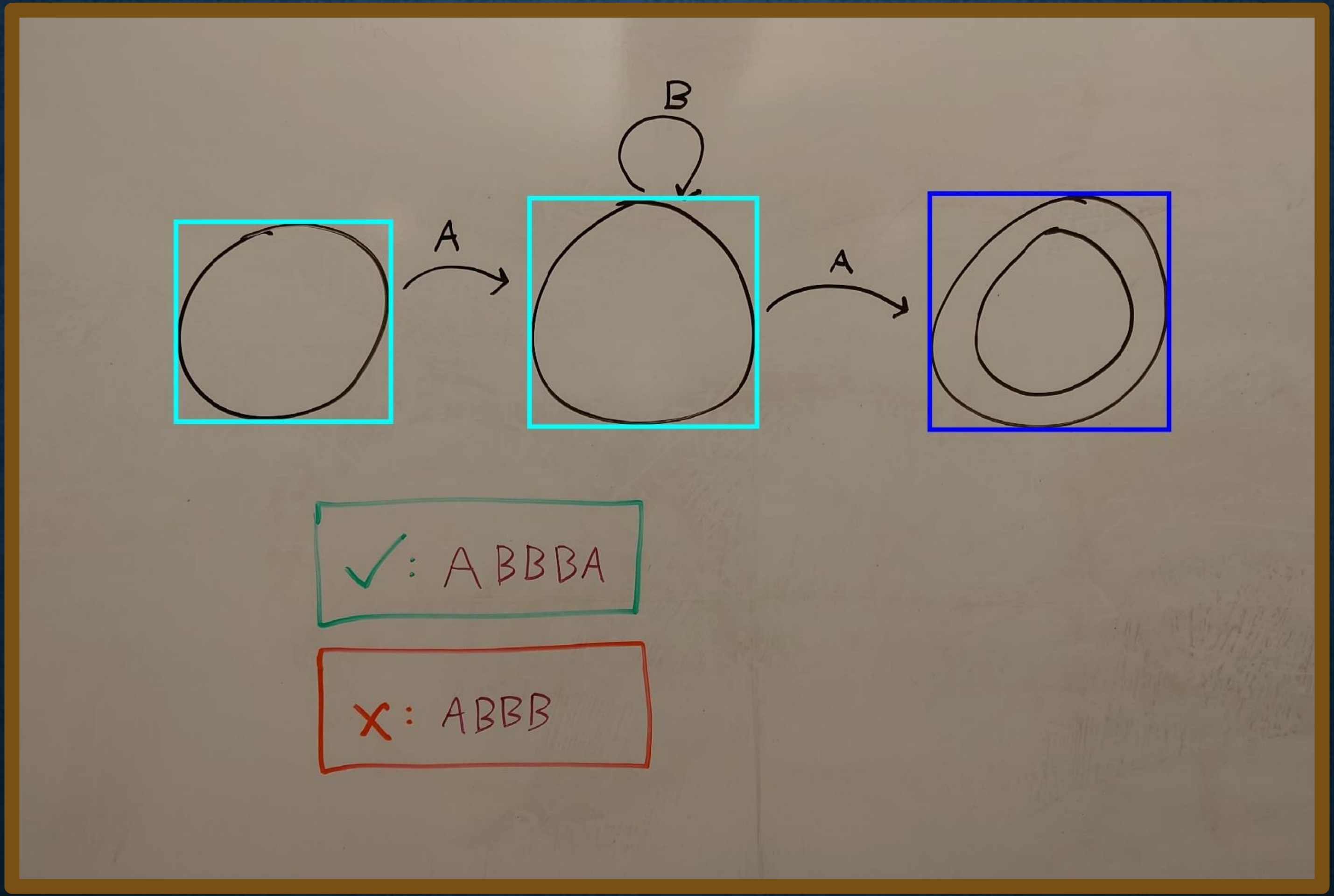
FEATURE EXTRACTION



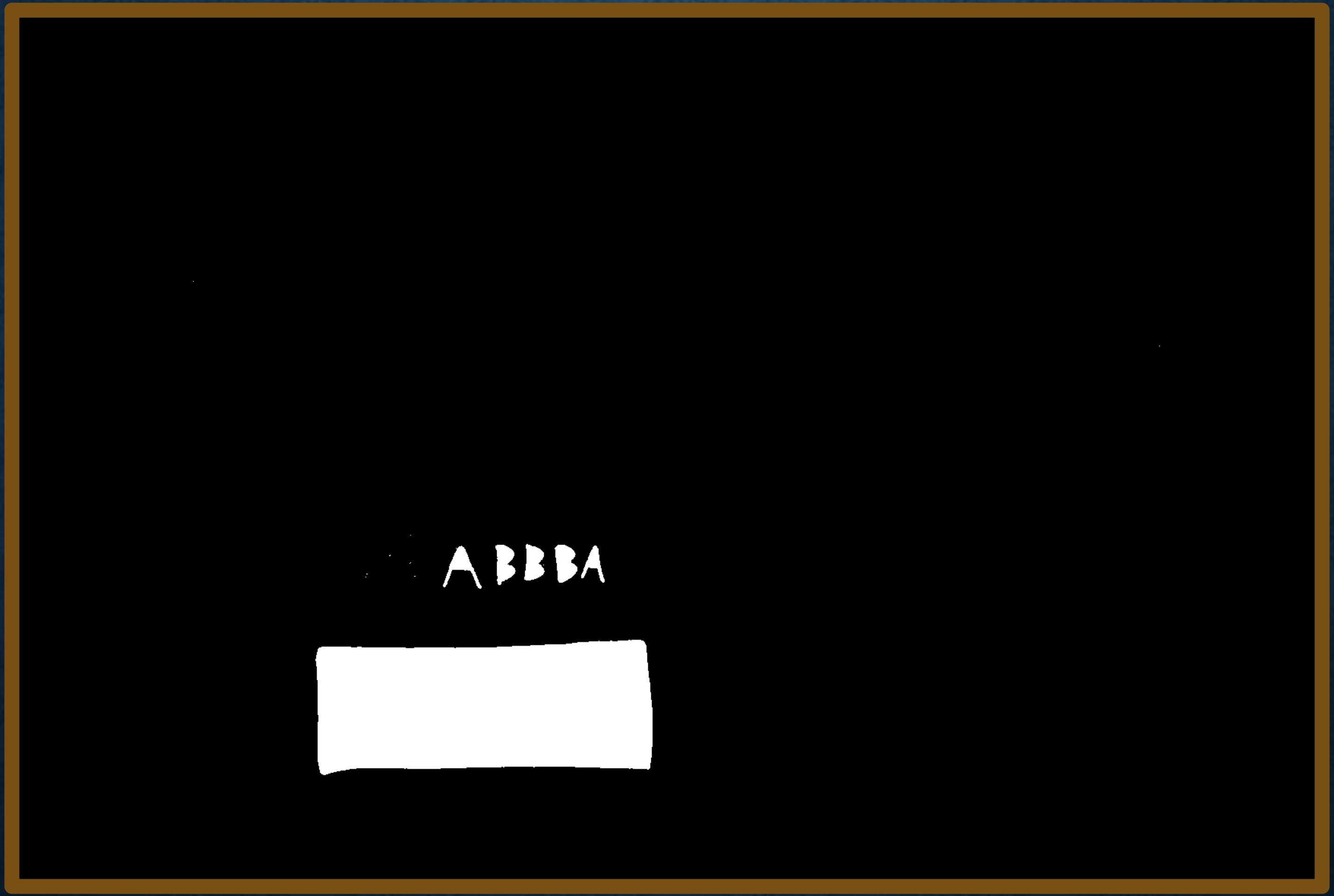
FEATURE EXTRACTION



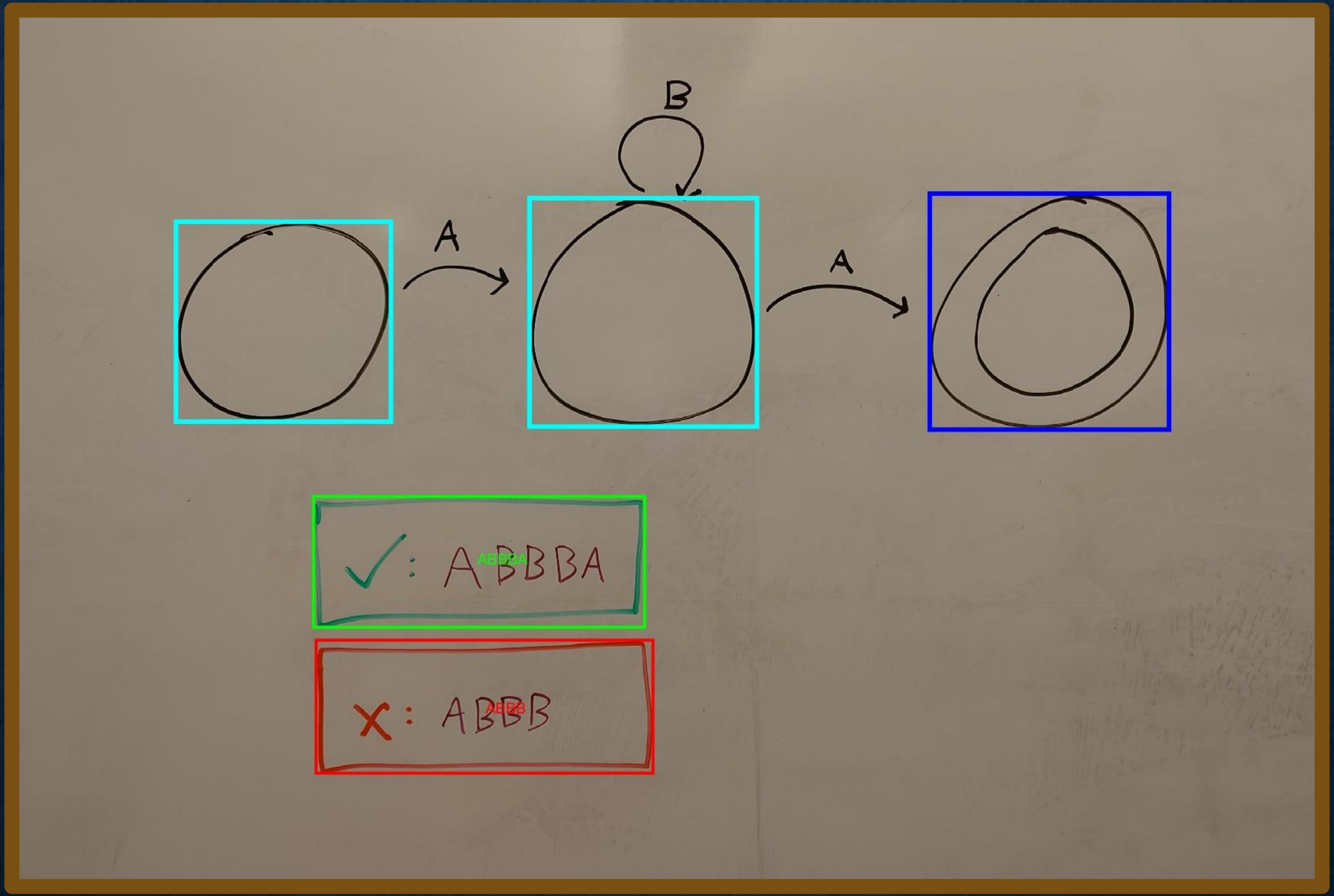
FEATURE EXTRACTION



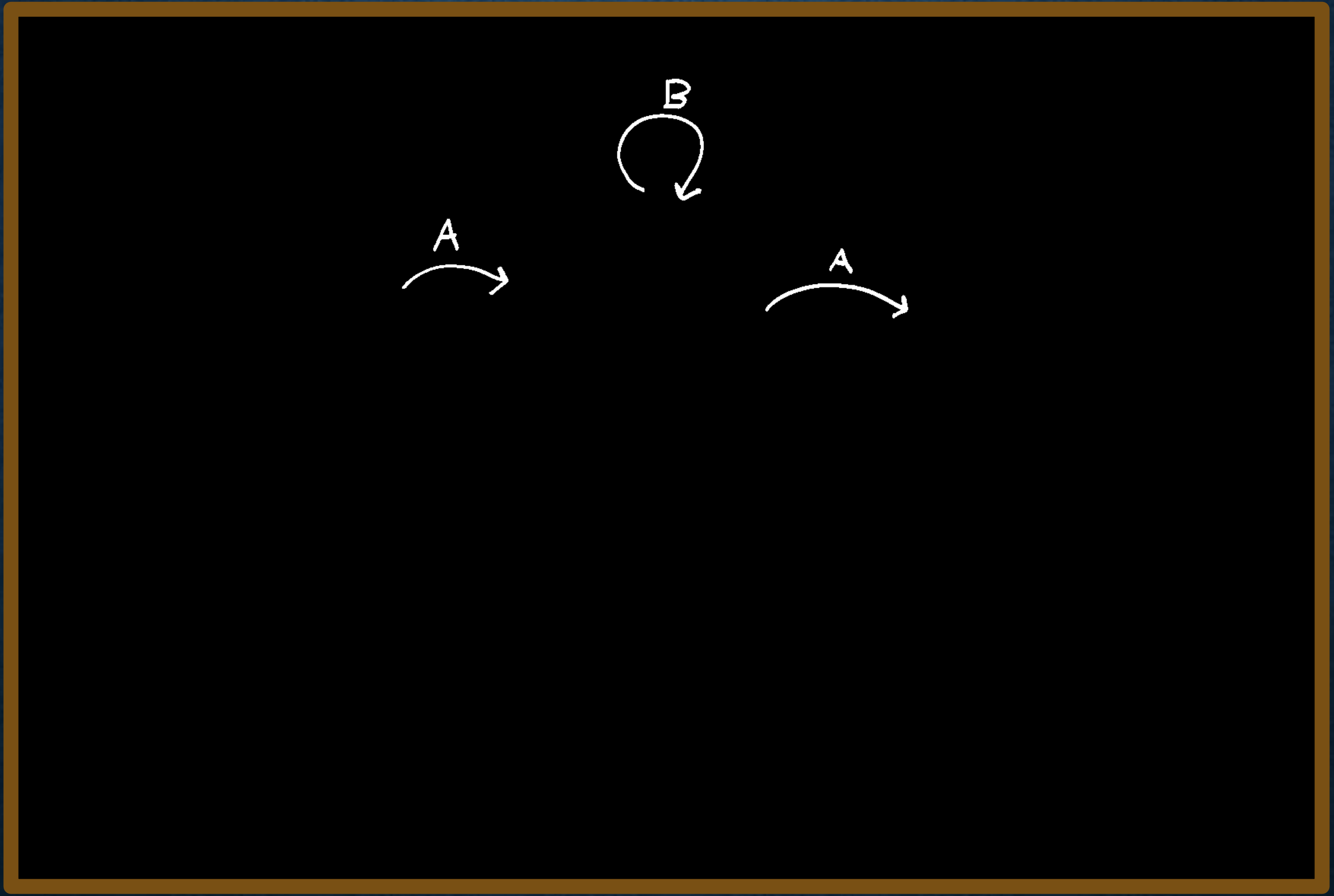
FEATURE EXTRACTION



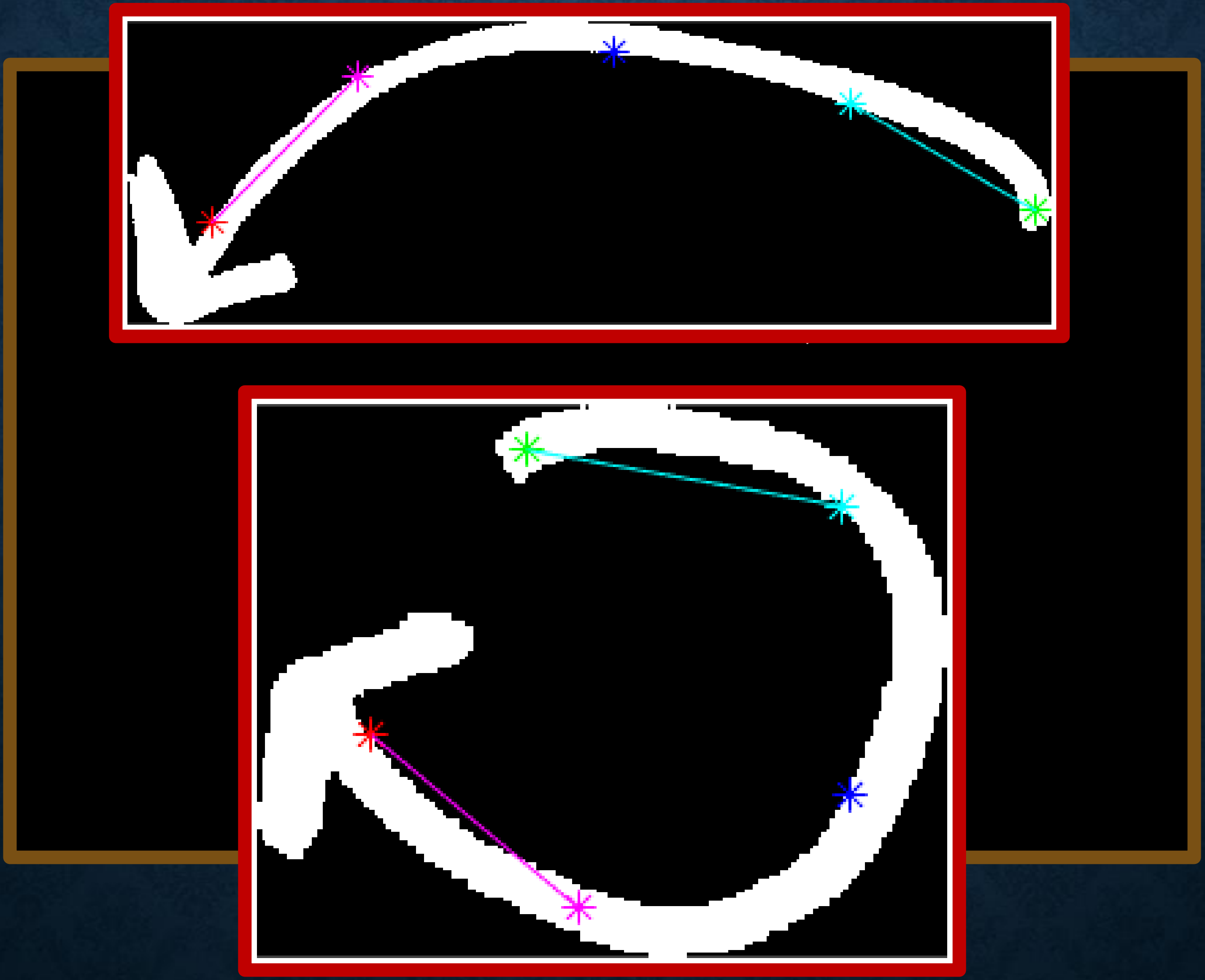
FEATURE EXTRACTION



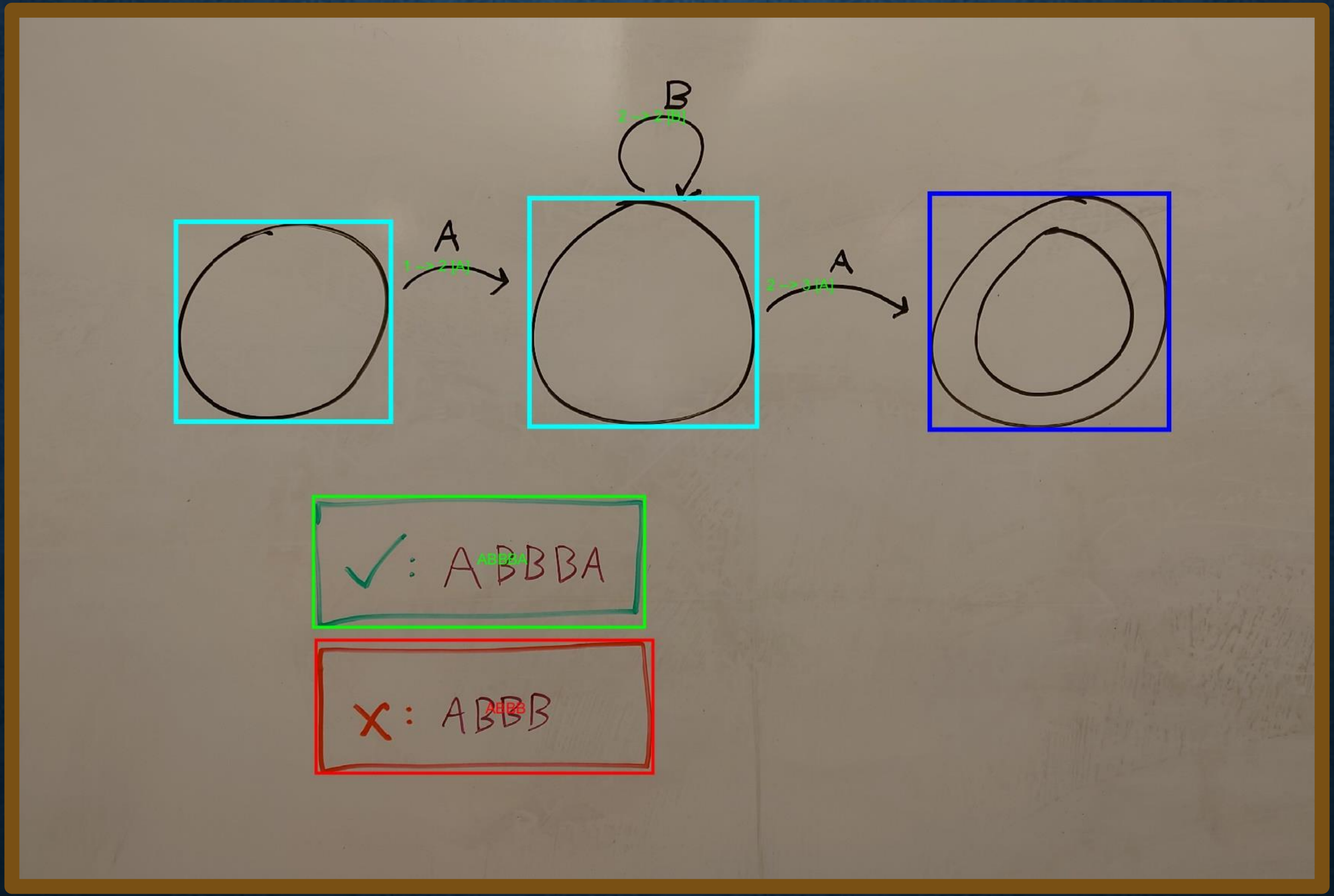
FEATURE EXTRACTION



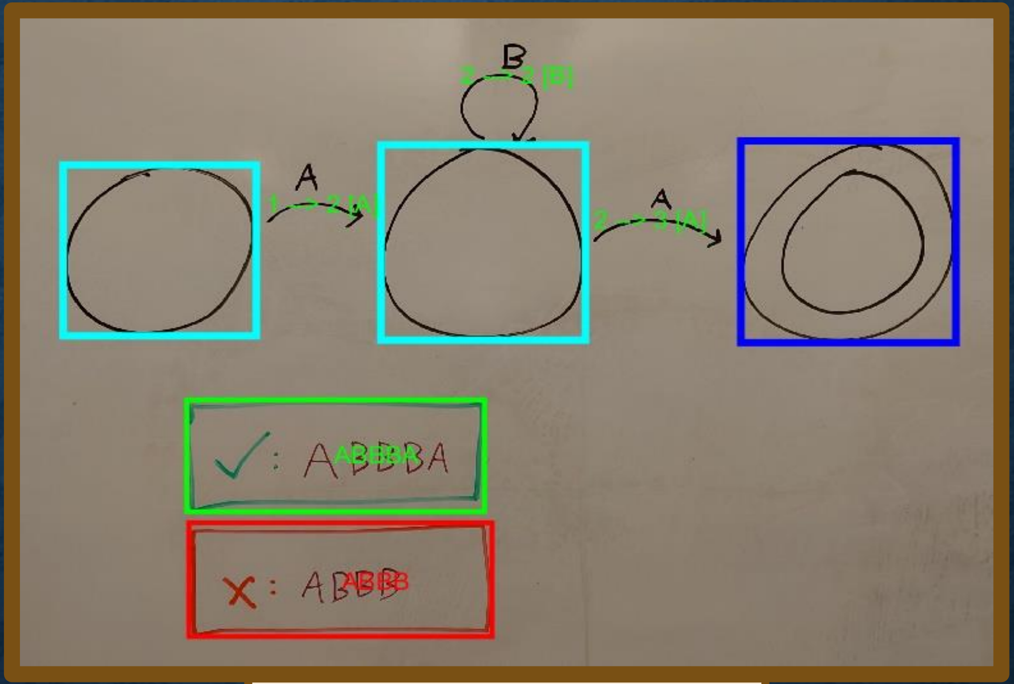
FEATURE EXTRACTION



FEATURE EXTRACTION

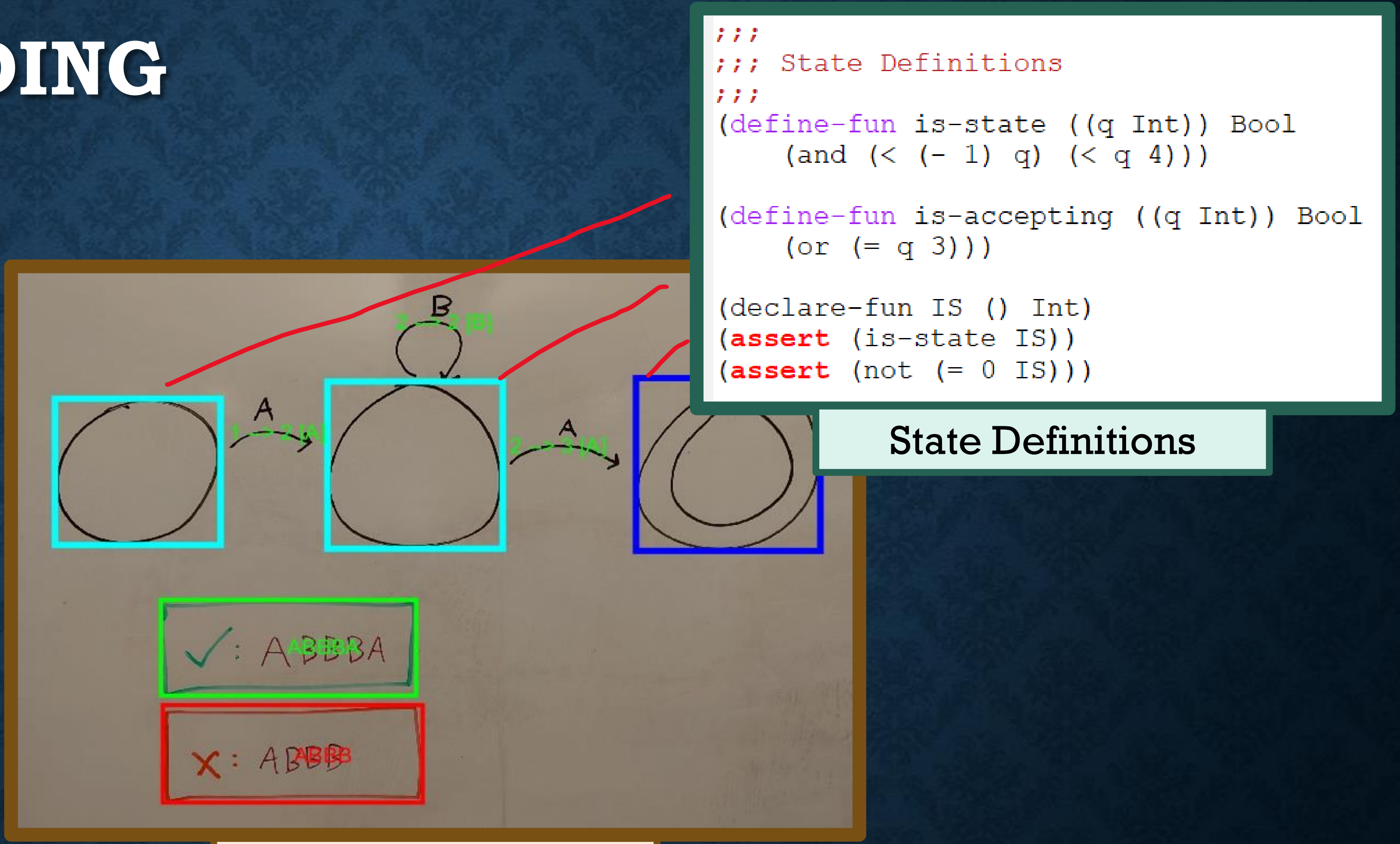


SMT-LIB2 ENCODING



Automaton Features

SMT-LIB2 ENCODING

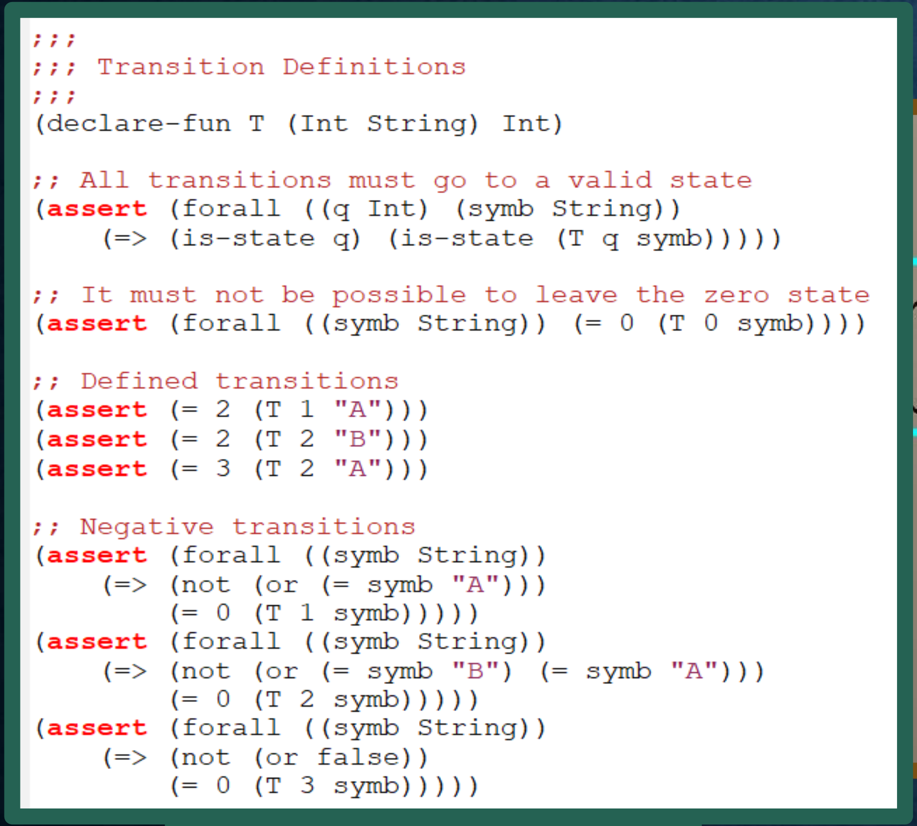


```
;;;
;;; State Definitions
;;;
(define-fun is-state ((q Int)) Bool
    (and (< (- 1) q) (< q 4)))

(define-fun is-accepting ((q Int)) Bool
    (or (= q 3)))

(declare-fun IS () Int)
(assert (is-state IS))
(assert (not (= 0 IS)))
```

State Definitions

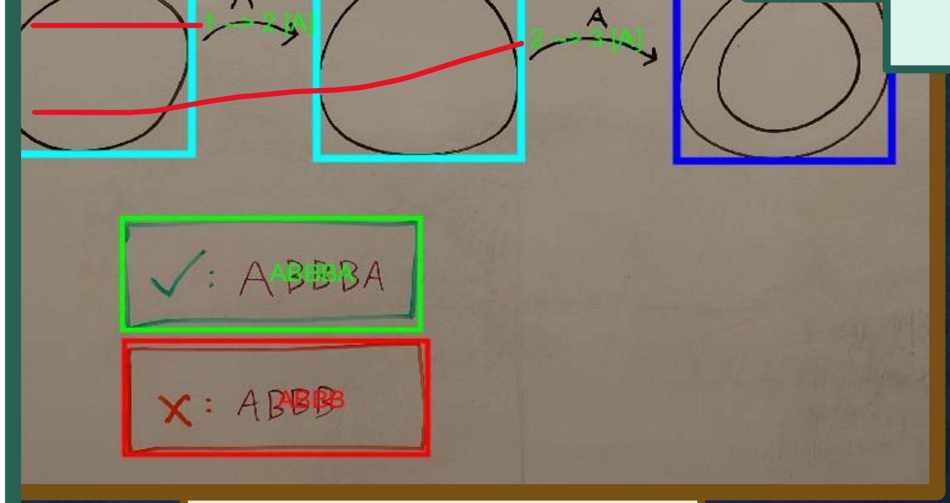
Automaton Features

SMT-LIB2 ENCODING

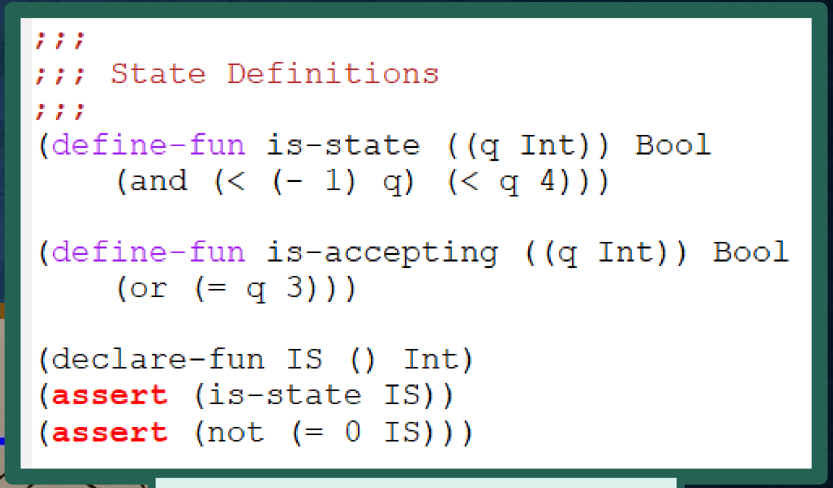


```
;;;
;;; Transition Definitions
;;;
(declare-fun T (Int String) Int)

;; All transitions must go to a valid state
(assert (forall ((q Int) (symb String))
    (=> (is-state q) (is-state (T q symb)))))

;; It must not be possible to leave the zero state
(assert (forall ((symb String)) (= 0 (T 0 symb))))

;; Defined transitions
(assert (= 2 (T 1 "A")))
(assert (= 2 (T 2 "B")))
(assert (= 3 (T 2 "A")))

;; Negative transitions
(assert (forall ((symb String))
    (=> (not (or (= symb "A")))
        (= 0 (T 1 symb)))))
(assert (forall ((symb String))
    (=> (not (or (= symb "B") (= symb "A")))
        (= 0 (T 2 symb)))))
(assert (forall ((symb String))
    (=> (not (or false))
        (= 0 (T 3 symb)))))
```

Transition Definitions



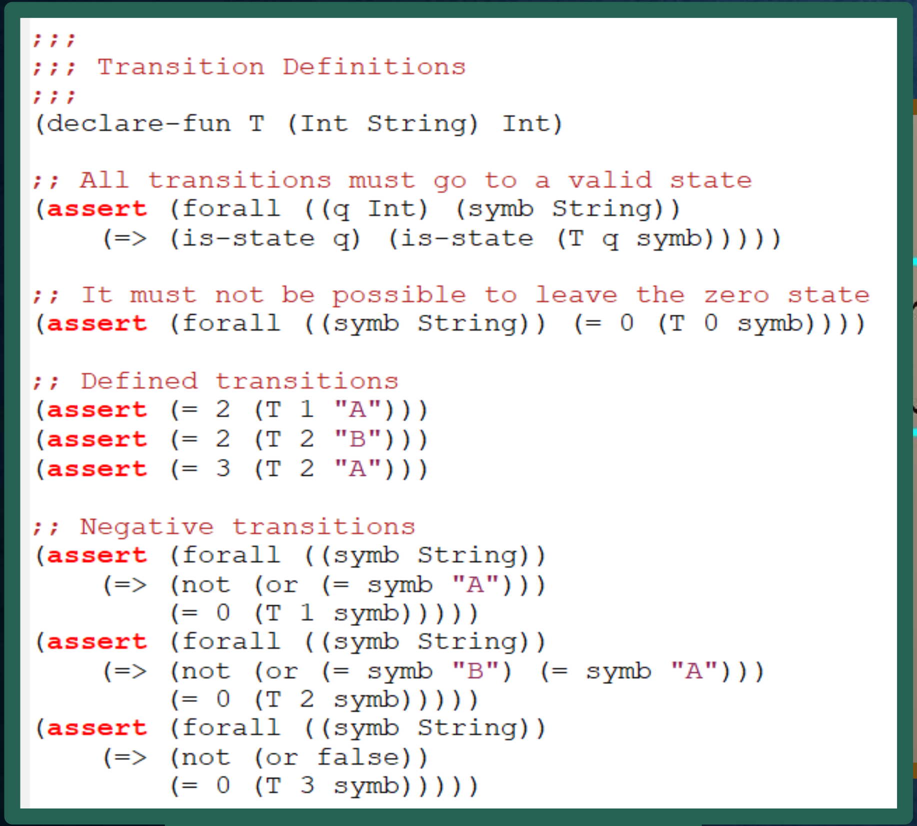
Automaton Features



```
;;;
;;; State Definitions
;;;
(define-fun is-state ((q Int)) Bool
    (and (< (- 1) q) (< q 4)))

(define-fun is-accepting ((q Int)) Bool
    (or (= q 3)))

(declare-fun IS () Int)
(assert (is-state IS))
(assert (not (= 0 IS)))
```

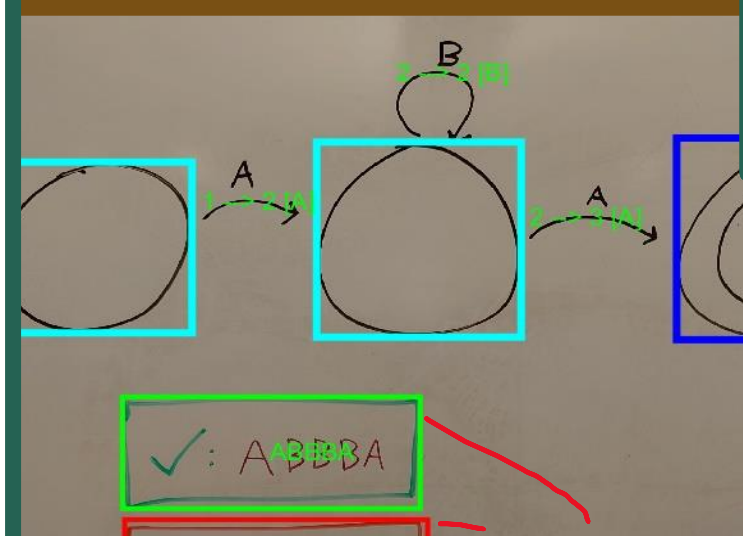
State Definitions

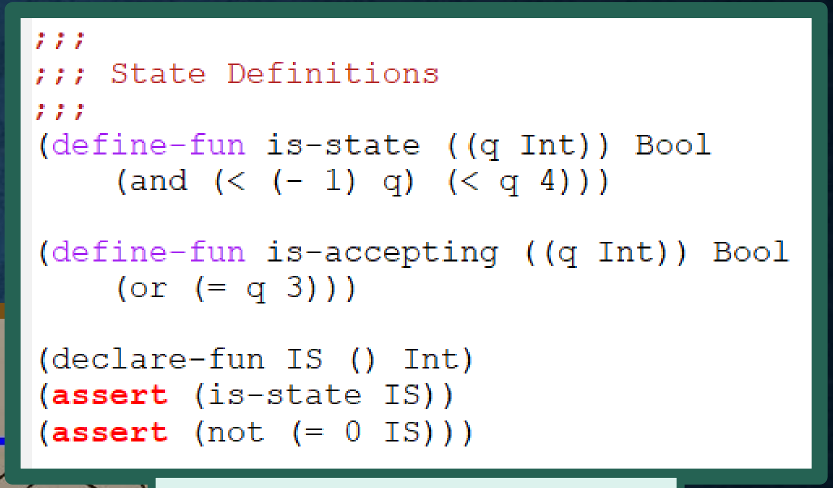
SMT-LIB2 ENCODING



```
;;;
;;; Transition Definitions
;;;
(declare-fun T (Int String) Int)

;; All transitions must go to a valid state
(assert (forall ((q Int) (symb String))
    (=> (is-state q) (is-state (T q symb)))))

;; It must not be possible to leave the zero state
(assert (forall ((symb String)) (= 0 (T 0 symb))))

;; Defined transitions
(assert (= 2 (T 1 "A")))
(assert (= 2 (T 2 "B")))
(assert (= 3 (T 2 "A")))

;; Negative transitions
(assert (forall ((symb String))
    (=> (not (or (= symb "A")))
        (= 0 (T 1 symb)))))
(assert (forall ((symb String))
    (=> (not (or (= symb "B") (= symb "A")))
        (= 0 (T 2 symb)))))
(assert (forall ((symb String))
    (=> (not (or false))
        (= 0 (T 3 symb)))))
```

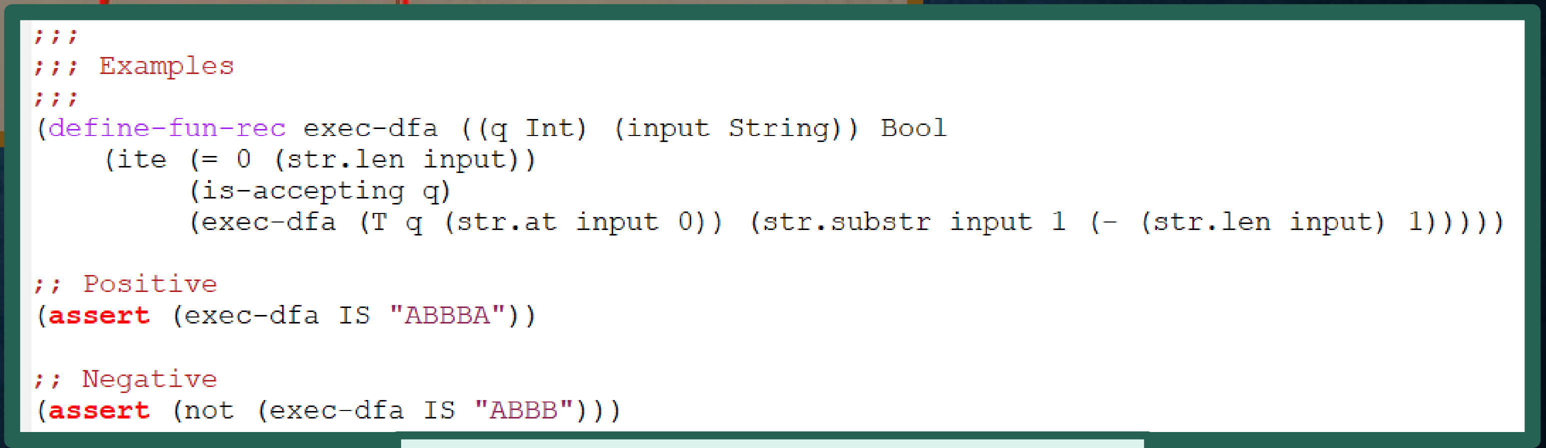
Transition Definitions





```
;;;
;;; State Definitions
;;;
(define-fun is-state ((q Int)) Bool
    (and (< (- 1) q) (< q 4)))

(define-fun is-accepting ((q Int)) Bool
    (or (= q 3)))

(declare-fun IS () Int)
(assert (is-state IS))
(assert (not (= 0 IS)))
```

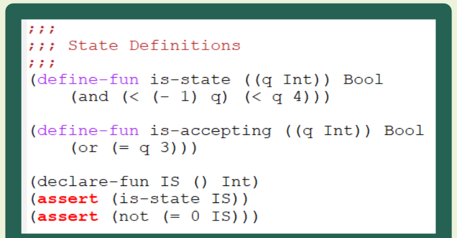
State Definitions



```
;;;
;;; Examples
;;;
(define-fun-rec exec-dfa ((q Int) (input String)) Bool
    (ite (= 0 (str.len input))
         (is-accepting q)
         (exec-dfa (T q (str.at input 0)) (str.substr input 1 (- (str.len input) 1)))))

;; Positive
(assert (exec-dfa IS "ABBBA"))

;; Negative
(assert (not (exec-dfa IS "ABBB")))
```

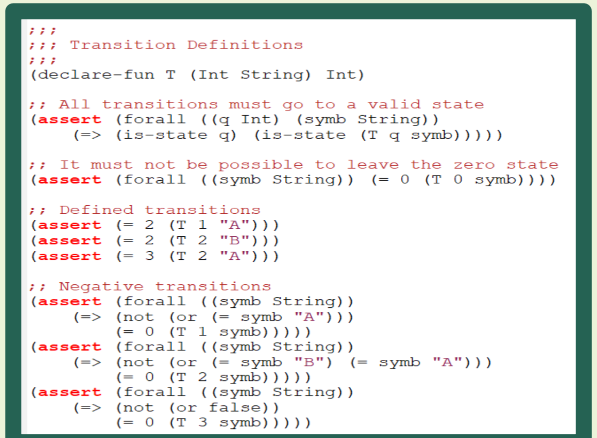
Positive and Negative Examples

SATISFIABILITY CHECK



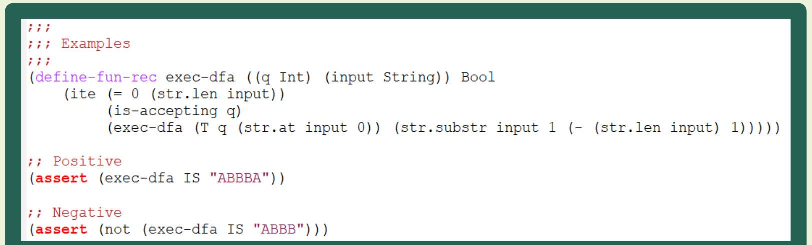
```
;;;
;;; State Definitions
;;;
(define-fun is-state ((q Int)) Bool
    (and (< (- 1) q) (< q 4)))

(define-fun is-accepting ((q Int)) Bool
    (or (= q 3)))

(declare-fun IS () Int)
(assert (is-state IS))
(assert (not (= 0 IS)))
```

State Definitions



```
;;;
;;; Transition Definitions
;;;
(declare-fun T (Int String) Int)

;; All transitions must go to a valid state
(assert (forall ((q Int) (symb String))
    (=> (is-state q) (is-state (T q symb)))))

;; It must not be possible to leave the zero state
(assert (forall ((symb String)) (= 0 (T 0 symb))))

;; Defined transitions
(assert (= 2 (T 1 "A")))
(assert (= 2 (T 2 "B")))
(assert (= 3 (T 2 "A")))

;; Negative transitions
(assert (forall ((symb String))
    (=> (not (or (= symb "A")))
        (= 0 (T 1 symb)))))
(assert (forall ((symb String))
    (=> (not (or (= symb "B") (= symb "A")))
        (= 0 (T 2 symb)))))
(assert (forall ((symb String))
    (=> (not (or false))
        (= 0 (T 3 symb)))))
```

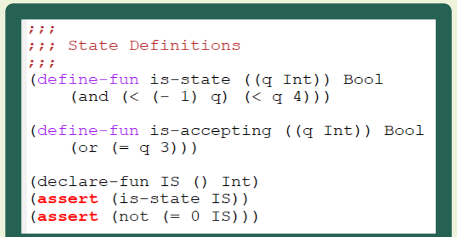
Transition Definitions



```
;;;
;;; Examples
;;;
(define-fun-rec exec-dfa ((q Int) (input String)) Bool
    (ite (= 0 (str.len input))
         (is-accepting q)
         (exec-dfa (T q (str.at input 0)) (str.substr input 1 (- (str.len input) 1)))))

;; Positive
(assert (exec-dfa IS "ABBBA"))

;; Negative
(assert (not (exec-dfa IS "ABBB")))
```

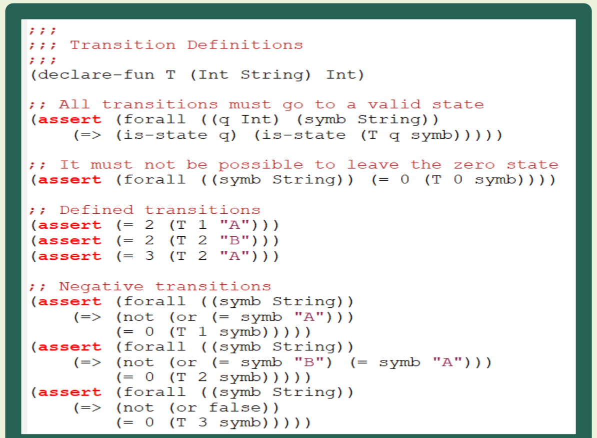
Positive and Negative Examples

SATISFIABILITY CHECK



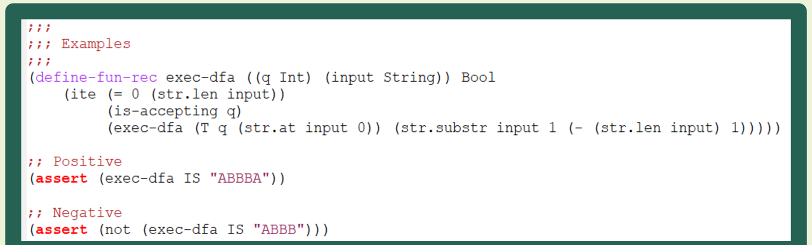
```
;;;
;;; State Definitions
;;;
(define-fun is-state ((q Int)) Bool
    (and (< (- 1) q) (< q 4)))

(define-fun is-accepting ((q Int)) Bool
    (or (= q 3)))

(declare-fun IS () Int)
(assert (is-state IS))
(assert (not (= 0 IS)))
```

State Definitions



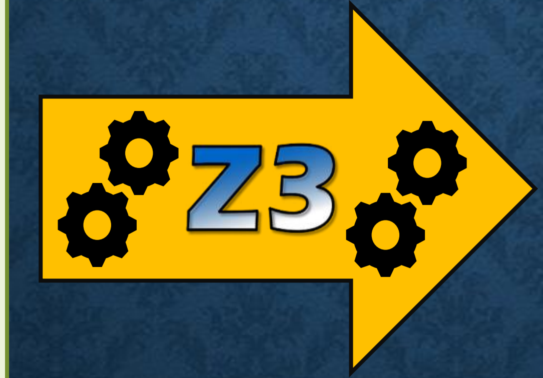
```
;;;
;;; Transition Definitions
;;;
(declare-fun T (Int String) Int)

;; All transitions must go to a valid state
(assert (forall ((q Int) (symb String))
    (=> (is-state q) (is-state (T q symb)))))

;; It must not be possible to leave the zero state
(assert (forall ((symb String)) (= 0 (T 0 symb))))

;; Defined transitions
(assert (= 2 (T 1 "A")))
(assert (= 2 (T 2 "B")))
(assert (= 3 (T 2 "A")))

;; Negative transitions
(assert (forall ((symb String))
    (=> (not (or (= symb "A")))
        (= 0 (T 1 symb)))))
(assert (forall ((symb String))
    (=> (not (or (= symb "B") (= symb "A")))
        (= 0 (T 2 symb)))))
(assert (forall ((symb String))
    (=> (not (or false))
        (= 0 (T 3 symb)))))
```

Transition Definitions

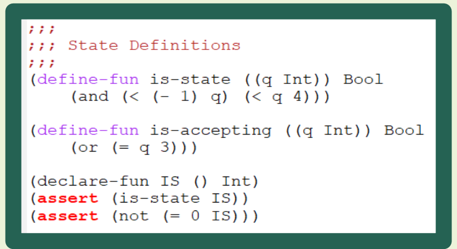


```
;;;
;;; Examples
;;;
(define-fun-rec exec-dfa ((q Int) (input String)) Bool
    (ite (= 0 (str.len input))
         (is-accepting q)
         (exec-dfa (T q (str.at input 0)) (str.substr input 1 (- (str.len input) 1)))))

;; Positive
(assert (exec-dfa IS "ABBBA"))

;; Negative
(assert (not (exec-dfa IS "ABBB")))
```

Positive and Negative Examples

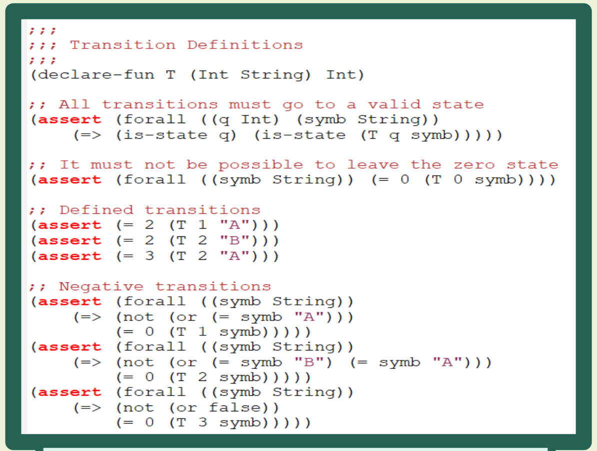


SATISFIABILITY CHECK



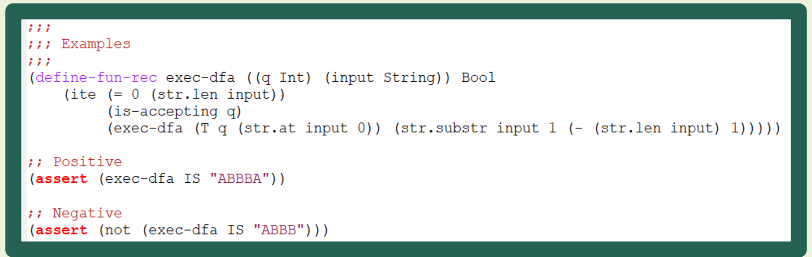
```
;;;
;;; State Definitions
;;;
(define-fun is-state ((q Int)) Bool
    (and (< (- 1) q) (< q 4)))

(define-fun is-accepting ((q Int)) Bool
    (or (= q 3)))

(declare-fun IS () Int)
(assert (is-state IS))
(assert (not (= 0 IS)))
```

State Definitions



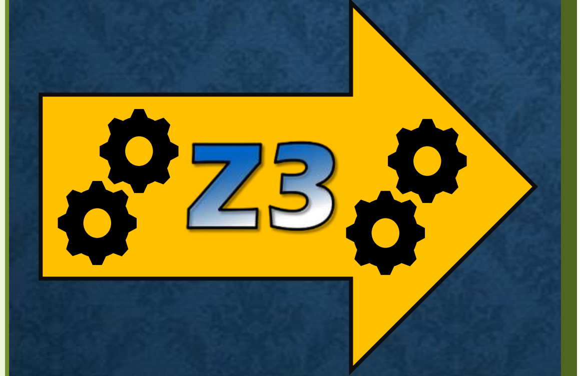
```
;;;
;;; Transition Definitions
;;;
(declare-fun T (Int String) Int)

;; All transitions must go to a valid state
(assert (forall ((q Int) (symb String))
    (=> (is-state q) (is-state (T q symb)))))

;; It must not be possible to leave the zero state
(assert (forall ((symb String)) (= 0 (T 0 symb))))

;; Defined transitions
(assert (= 2 (T 1 "A")))
(assert (= 2 (T 2 "B")))
(assert (= 3 (T 2 "A")))

;; Negative transitions
(assert (forall ((symb String))
    (=> (not (or (= symb "A")))
        (= 0 (T 1 symb)))))
(assert (forall ((symb String))
    (=> (not (or (= symb "B") (= symb "A")))
        (= 0 (T 2 symb)))))
(assert (forall ((symb String))
    (=> (not (or false))
        (= 0 (T 3 symb)))))
```

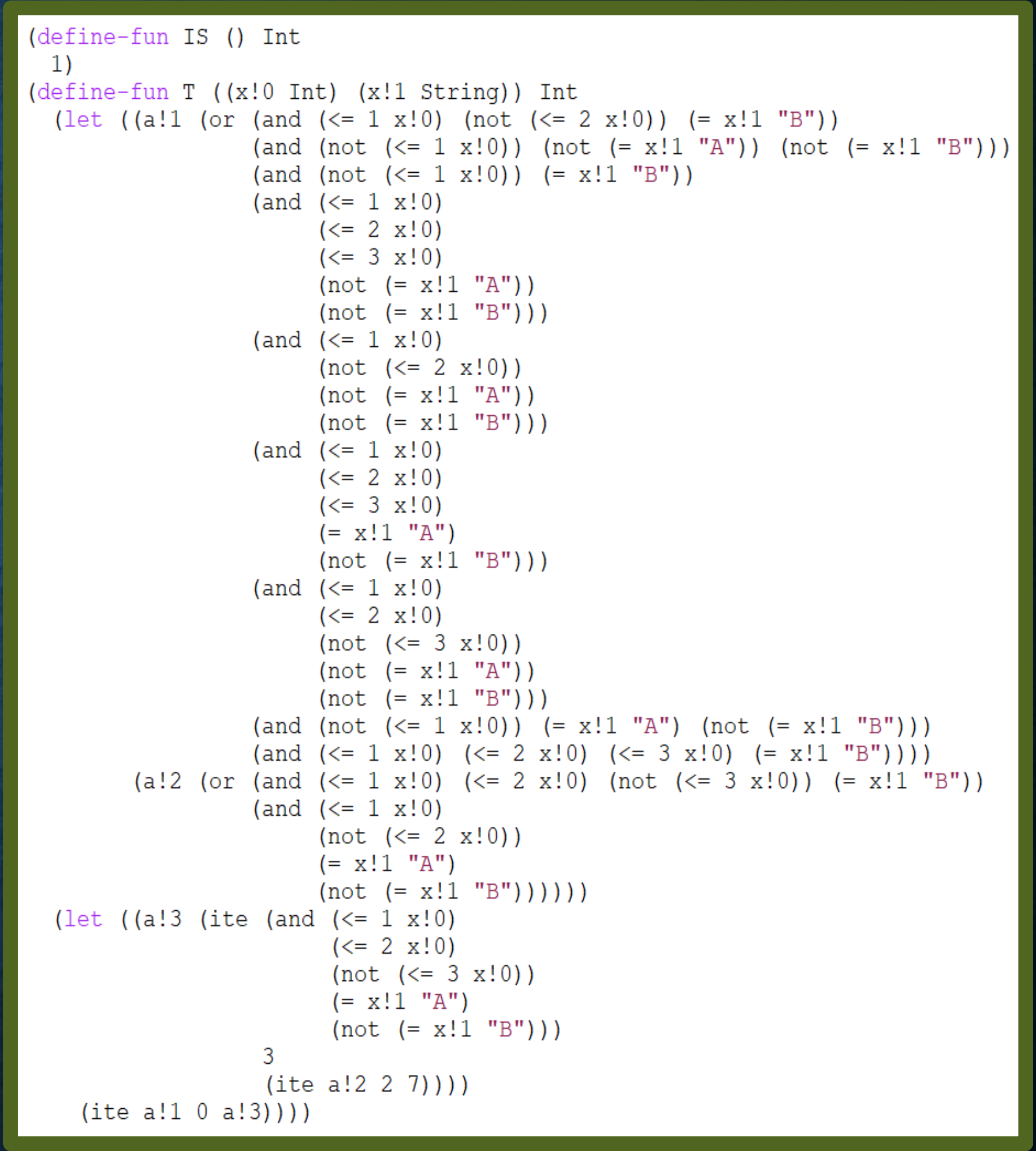
Transition Definitions



```
;;;
;;; Examples
;;;
(define-fun-rec exec-dfa ((q Int) (input String)) Bool
    (ite (= 0 (str.len input))
         (is-accepting q)
         (exec-dfa (T q (str.at input 0)) (str.substr input 1 (- (str.len input) 1)))))

;; Positive
(assert (exec-dfa IS "ABBBA"))

;; Negative
(assert (not (exec-dfa IS "ABBB")))
```

Positive and Negative Examples

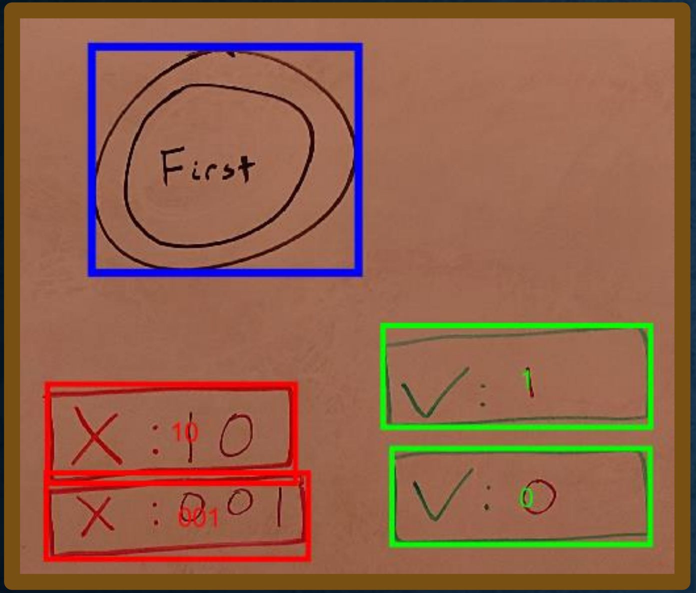




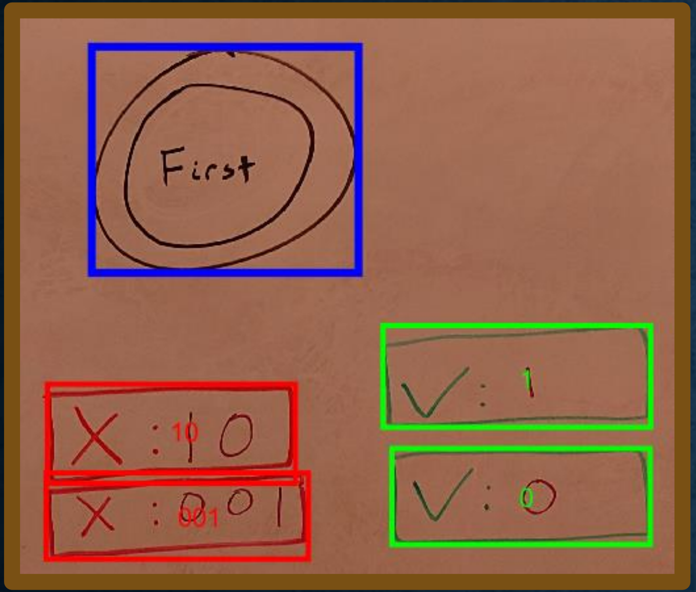
```
(define-fun IS () Int
  1)
(define-fun T ((x!0 Int) (x!1 String)) Int
  (let ((a!1 (or (and (<= 1 x!0) (not (<= 2 x!0)) (= x!1 "B"))
                 (and (not (<= 1 x!0)) (not (= x!1 "A")) (not (= x!1 "B")))
                 (and (not (<= 1 x!0)) (= x!1 "B"))
                 (and (<= 1 x!0)
                      (<= 2 x!0)
                      (<= 3 x!0)
                      (not (= x!1 "A"))
                      (not (= x!1 "B")))
                 (and (<= 1 x!0)
                      (not (<= 2 x!0))
                      (not (= x!1 "A"))
                      (not (= x!1 "B")))
                 (and (<= 1 x!0)
                      (<= 2 x!0)
                      (<= 3 x!0)
                      (= x!1 "A")
                      (not (= x!1 "B")))
                 (and (<= 1 x!0)
                      (<= 2 x!0)
                      (not (<= 3 x!0))
                      (not (= x!1 "A"))
                      (not (= x!1 "B")))
                 (and (not (<= 1 x!0)) (= x!1 "A") (not (= x!1 "B")))
                 (and (<= 1 x!0) (<= 2 x!0) (<= 3 x!0) (= x!1 "B"))))
        (a!2 (or (and (<= 1 x!0) (<= 2 x!0) (not (<= 3 x!0)) (= x!1 "B"))
                 (and (<= 1 x!0)
                      (not (<= 2 x!0))
                      (= x!1 "A")
                      (not (= x!1 "B"))))))
  (let ((a!3 (ite (and (<= 1 x!0)
                       (<= 2 x!0)
                       (not (<= 3 x!0))
                       (= x!1 "A")
                       (not (= x!1 "B")))
                  3
                  (ite a!2 2 7))))
    (ite a!1 0 a!3))))
```

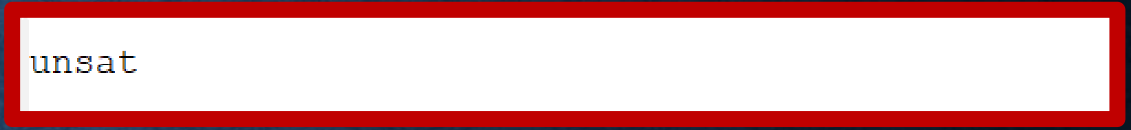
Synthesized Program

UNSATISFIABLE?



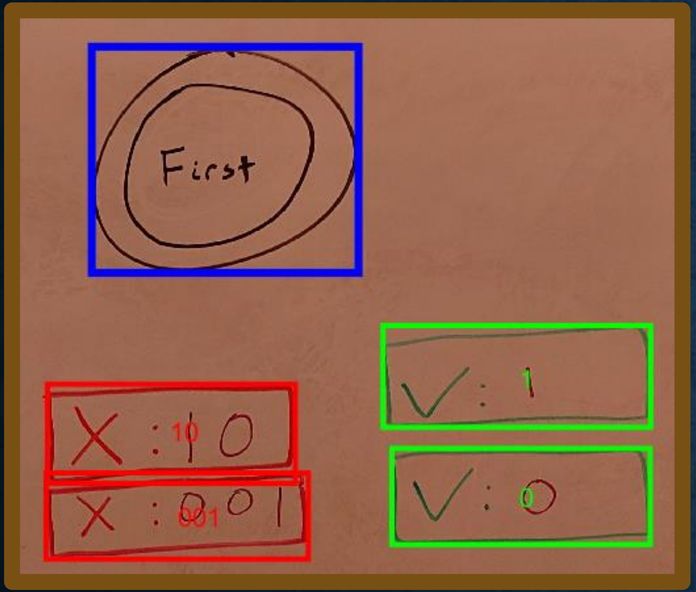
UNSATISFIABLE?

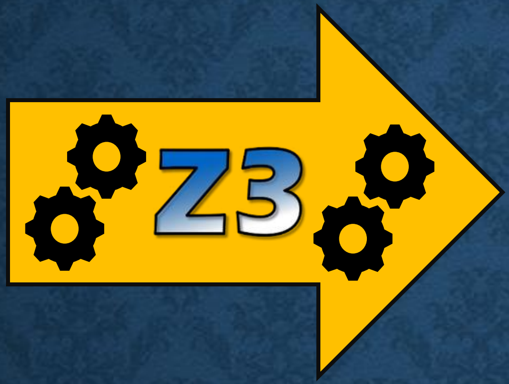




```
unsat
```

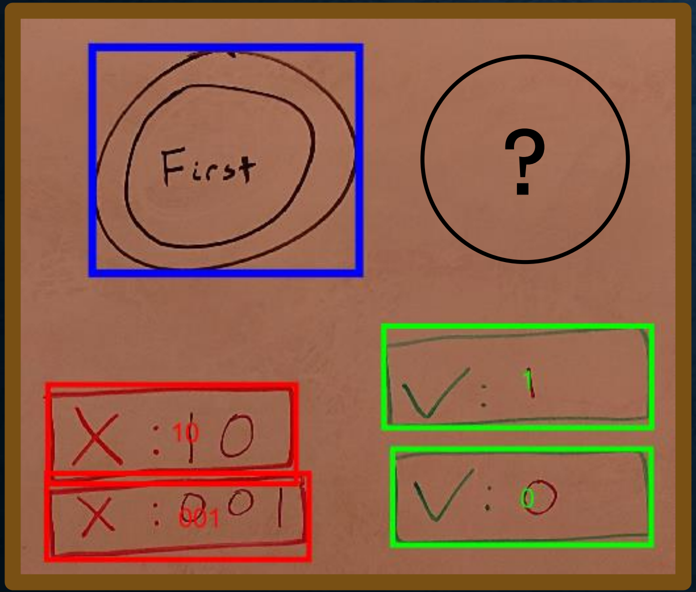
UNSATISFIABLE?



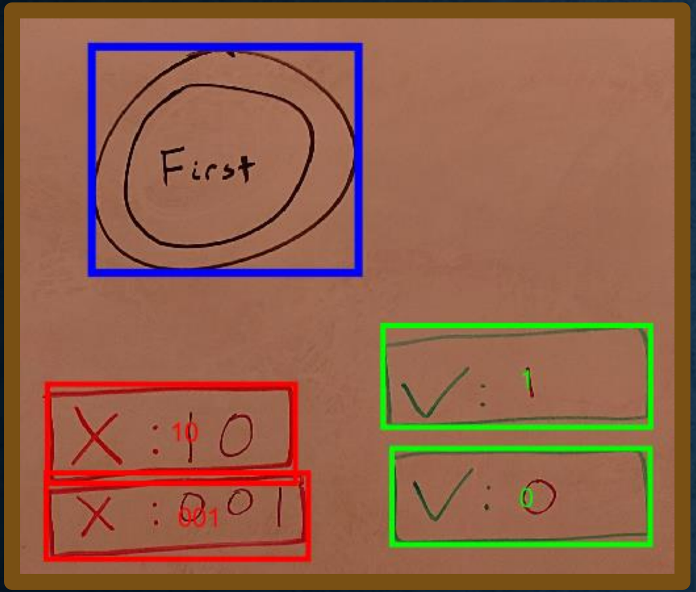


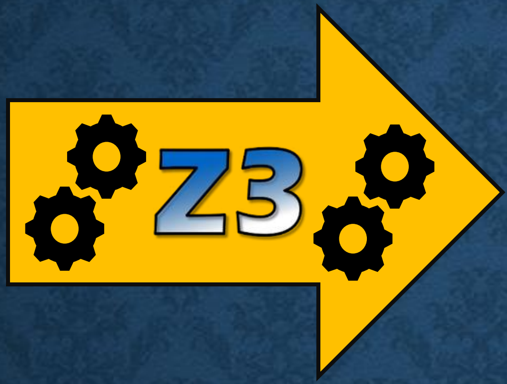
```
unsat
```

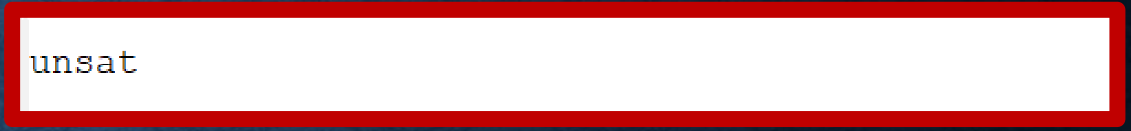
Add extra state



UNSATISFIABLE?

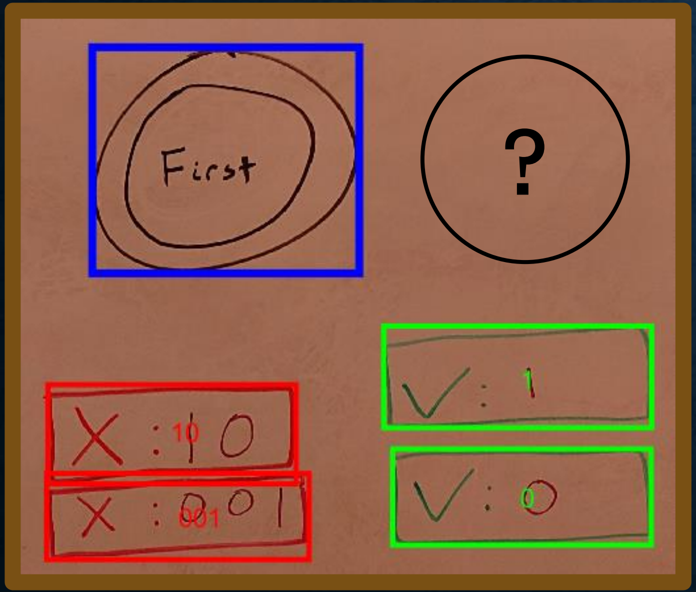


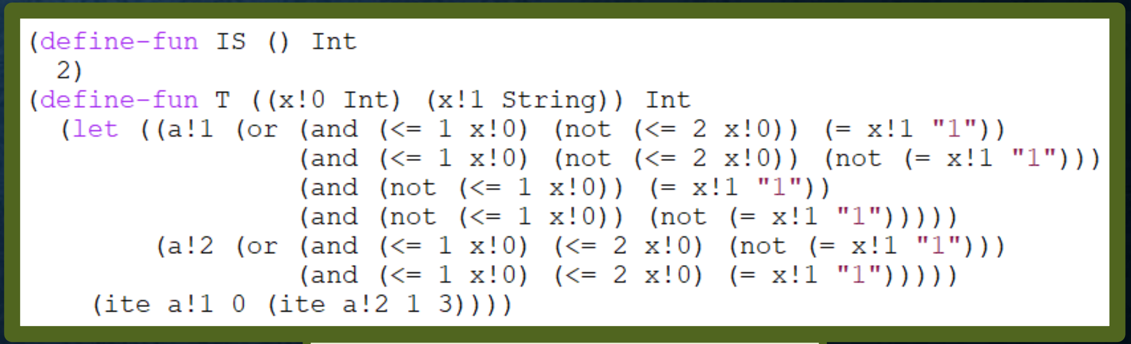




```
unsat
```

Add extra state





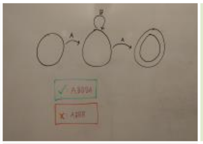
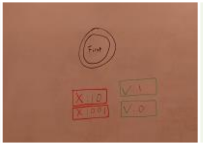
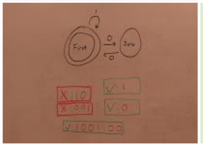
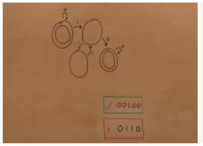
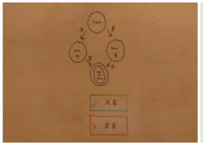
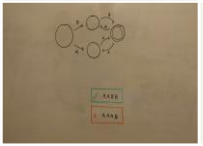
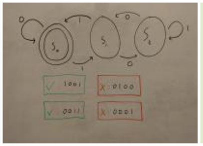
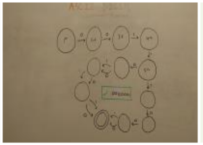
```
(define-fun IS () Int
  2)
(define-fun T ((x!0 Int) (x!1 String)) Int
  (let ((a!1 (or (and (<= 1 x!0) (not (<= 2 x!0)) (= x!1 "1"))
                 (and (<= 1 x!0) (not (<= 2 x!0)) (not (= x!1 "1")))
                 (and (not (<= 1 x!0)) (= x!1 "1"))
                 (and (not (<= 1 x!0)) (not (= x!1 "1")))))
        (a!2 (or (and (<= 1 x!0) (<= 2 x!0) (not (= x!1 "1")))
                 (and (<= 1 x!0) (<= 2 x!0) (= x!1 "1")))))
    (ite a!1 0 (ite a!2 1 3))))
```

Synthesized Program
(with an extra state)

RESULTS

Used for core algorithm design

Used for evaluation and heuristic tweaks

Image	Name	All Features?	Extra States?	Tweaks Needed?
	A1	Yes	0	Development
	A2	Yes	1	Development
	A3	Yes	1	Development
	Odd One	Yes	0	Transitions, OCR Heuristics
	One of Each	Yes	0	None
	Stutter	Yes	0	None
	Multiple of 3	Yes	0	OCR Heuristics
	ASCII Digits	Yes	0	Transitions

All synthesized successfully!

FUTURE WORK

Improve OCR

- Only 'A', 'B', '0', and '1' are supported
- Heuristics are still required to fix up cases that OCR gets wrong

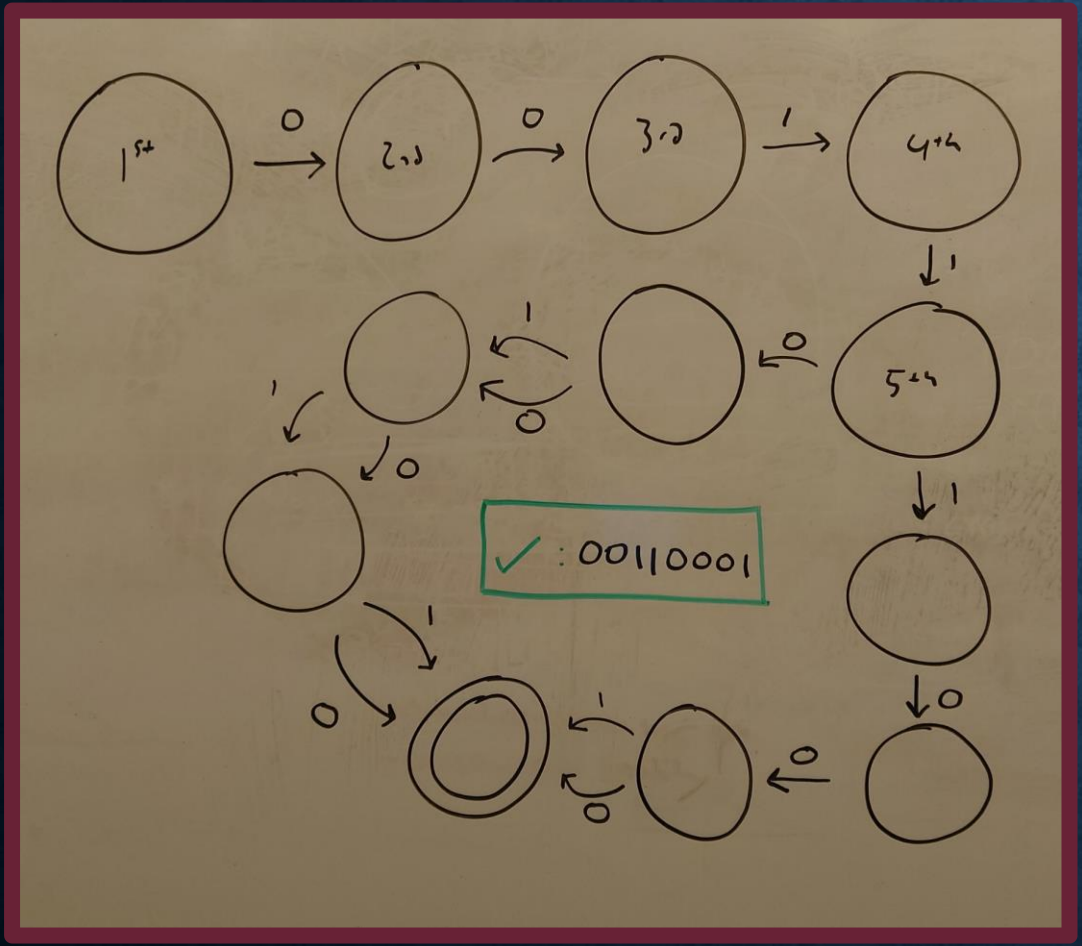
Additional automata types

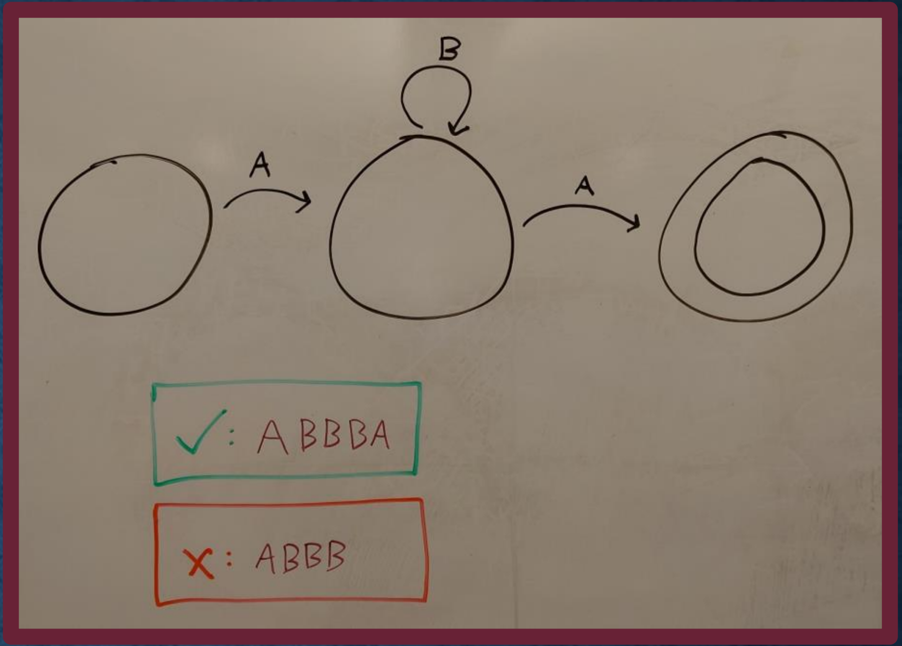
- NFAs (trivial extension)
- More exotic types: push-down automata, transducers, and Büchi automata

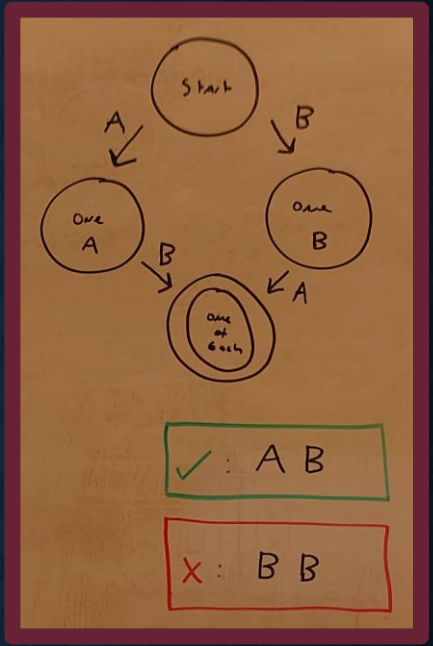
Synthesis under uncertainty

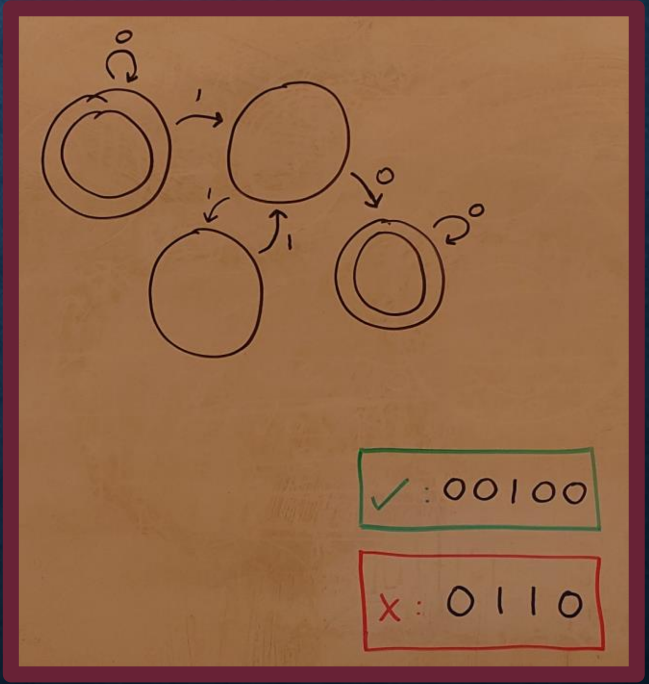
- Currently, recognized features are set in stone
- Instead of requiring "perfect" feature recognition, give confidence values to the synthesizer
- An interesting synthesis task, but is it really what a user would want?

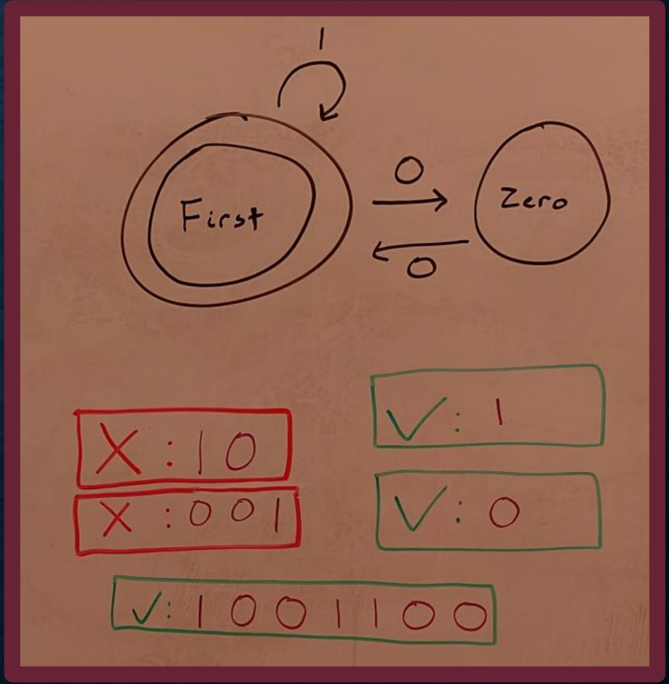
Q&A


1st
0
2nd
0
3rd
1
4th
1
5th
0
1
0
1
0
1
0
1
0
0
1
0
1
0
✓ : 00110001


A
B
A
✓: ABBBA
✗: ABBB


Start
A
B
One A
One B
B
A
One of each
✓ : A B
✗ : B B


0
1
0
1
1
0
✓ : 00100
✗ : 0110


1
First
0
Zero
0
✗ : 10
✗ : 001
✓ : 1
✓ : 0
✓ : 1001100

Thanks!