Turing Machines

Turing Machine Composition

Turing Machine Extensions

Context-Sensitive Grammars

Church-Turing Thesis and Undecidability

Complexity

Farewell and where to go from here?

Part IV: Context-Sensitive Languages

Dr. Marco T. Morazán

Seton Hall University

Outline

1. Turing Machines
2. Turing Machine Composition
3. Turing Machine Extensions
4. Context-Sensitive Grammars
5. Church-Turing Thesis and Undecidability
6. Complexity
7. Farewell and where to go from here?

Turing Machines

- pdas replace ndfas
- No pda can decide $L = \{a^n b^n c^n \mid n \geq 0\}$
- We need a more powerful model

Turing Machines

- pdas replace ndfas
- No pda can decide $L = \{a^n b^n c^n \mid n \geq 0\}$
- We need a more powerful model
- The new computation model will not be replaced by a more powerful model
- The new model is called a *Turing machine* named after its inventor Alan Turing.
- Differences:
  - head may move right or left on the input
  - may write mutate the tape

Turing Machines

- pdas replace ndfas
- No pda can decide $L = \{a^n b^n c^n \mid n \geq 0\}$
- We need a more powerful model
- The new computation model will not be replaced by a more powerful model
- The new model is called a *Turing machine* named after its inventor Alan Turing.
- Differences:
  - head may move right or left on the input
  - may write mutate the tape
- Shall do much more than simply decide a language
- Shall also compute the value of functions

Turing Machines

- pdas replace ndfas
- No pda can decide $L = \{a^n b^n c^n \mid n \geq 0\}$
- We need a more powerful model
- The new computation model will not be replaced by a more powerful model
- The new model is called a *Turing machine* named after its inventor Alan Turing.
- Differences:
  - head may move right or left on the input
  - may write mutate the tape
- Shall do much more than simply decide a language
- Shall also compute the value of functions
- It is theorized that a Turing machine can compute anything that is computable
- In this sense, Turing machines are more powerful than any computer in existence today

Turing Machines

- A Turing machine language recognizer is an instance of:

  `(make-tm K Σ R S F Y)`

Turing Machines

- A Turing machine language recognizer is an instance of:

  `(make-tm K Σ R S F Y)`
- `R` is a transition function
- We shall relax this condition and allow `R` to be a transition relation

Turing Machines

- A Turing machine language recognizer is an instance of:

  `(make-tm K Σ R S F Y)`
- `R` is a transition function
- We shall relax this condition and allow `R` to be a transition relation
- A deterministic Turing machine language recognizer requires two final states usually named `Y` and `N`
- When a Turing machine reaches a final state it halts and performs no more transitions

Turing Machines

- A Turing machine language recognizer is an instance of:

  `(make-tm K Σ R S F Y)`
- `R` is a transition function
- We shall relax this condition and allow `R` to be a transition relation
- A deterministic Turing machine language recognizer requires two final states usually named `Y` and `N`
- When a Turing machine reaches a final state it halts and performs no more transitions
- Two special symbols that may appear of the input tape: `LM` and `BLANK` (both `FSM` constants)
- A Turing machine rule, `tm`-rule, is an element of:

  `(list (list N a) (list M A))`
- `N` is non-halting state
- `a`$\in\{\Sigma \cup \{$`LM BLANK`$\}\}$,
- `M`$\in$`K`
- `A` is an `action`$\in\{\Sigma \cup \{$`RIGHT LEFT`$\}$
- If `A`$\in\Sigma$ then the machine writes `A` in the tape position under the tape's head

Turing Machines

- A Turing machine language recognizer is an instance of:

  `(make-tm K Σ R S F Y)`
- `R` is a transition function
- We shall relax this condition and allow `R` to be a transition relation
- A deterministic Turing machine language recognizer requires two final states usually named `Y` and `N`
- When a Turing machine reaches a final state it halts and performs no more transitions
- Two special symbols that may appear of the input tape: `LM` and `BLANK` (both `FSM` constants)
- A Turing machine rule, `tm`-rule, is an element of:

  `(list (list N a) (list M A))`
- `N` is non-halting state
- a∈{Σ ∪ {LM BLANK}},
- `M`∈`K`
- `A` is an `action`∈{Σ ∪ {RIGHT LEFT}
- If `A`∈Σ then the machine writes `A` in the tape position under the tape's head
- When `LM` is read the `tm` must move the tape's head right (regardless of the state it is in)
- May not overwrite `LM`
- The `tm` cannot "fall off" the left end of the input tape

Turing Machines

- A Turing machine configuration is a triple: `(state natnum tape)`
- Only the "touched" part of the tape is displayed
- The touched part of the input tape includes the left-end marker and anything specified in the initial tape value including blanks

Turing Machines

- A Turing machine configuration is a triple: `(state natnum tape)`
- Only the "touched" part of the tape is displayed
- The touched part of the input tape includes the left-end marker and anything specified in the initial tape value including blanks
- Section 1 displays the `FSM` control view visualization for `(S 2 (@ a a a))`
- Section 1 displays the `FSM` control view visualization for `(Y 4 (@ a a a _))`.

Turing Machines

- A computation, $\mathtt{C}_i \vdash^* \mathtt{C}_j$, is valid for $\mathtt{M}$ if and only if $\mathtt{M}$ can move from $\mathtt{C}_i$ to $\mathtt{C}_j$ using zero or more transitions

Turing Machines

- A computation, $\mathtt{C}_i \vdash^* \mathtt{C}_j$, is valid for `M` if and only if `M` can move from $\mathtt{C}_i$ to $\mathtt{C}_j$ using zero or more transitions
- A word, `w`, is accepted by a `tm` language recognizer if it reaches the final accepting state
- Otherwise, `w` is rejected

Turing Machines

- A computation, $\mathtt{C}_i \vdash^* \mathtt{C}_j$, is valid for `M` if and only if `M` can move from $\mathtt{C}_i$ to $\mathtt{C}_j$ using zero or more transitions
- A word, `w`, is accepted by a `tm` language recognizer if it reaches the final accepting state
- Otherwise, `w` is rejected
- A Turing machine language recognizer's execution may be observed using `sm-visualize`
- As you may already suspect, the execution may only be visualized when the given word is in the machine's language

Turing Machines

- A computation, $\mathtt{C}_i \vdash^* \mathtt{C}_j$, is valid for `M` if and only if `M` can move from $\mathtt{C}_i$ to $\mathtt{C}_j$ using zero or more transitions
- A word, `w`, is accepted by a `tm` language recognizer if it reaches the final accepting state
- Otherwise, `w` is rejected
- A Turing machine language recognizer's execution may be observed using `sm-visualize`
- As you may already suspect, the execution may only be visualized when the given word is in the machine's language
- State invariant predicates take as input the "touched" part of the input tape and the position, `i`, of the input tape's next element to read
- The predicate asserts a condition about the touched input that must hold which may or may not be in relation to the head's position
- For language recognizers, it is important to remember that when the machine's head is at position `i` the tape element at `i` has not been read.
- Finally, to add a blank to the input you type `BLANK` in the tape input box in the left column.

Turing Machines

- `L = a*`

Turing Machines

- `L = a`*
- Name: `a*` $\Sigma = \{$`a b`$\}$
- `;; PRE: tape = LMw` $\wedge$ `i = 0`

Turing Machines

- L = a*
- Name: a* $\Sigma = \{$a b$\}$
- ;; PRE: tape = LMw $\wedge$ i = 0
- Tests

```
;; Tests for a*
(check-equal? (sm-apply a* `(,LM a a a b a a)) 'reject)
(check-equal? (sm-apply a* `(,LM b a a)) 'reject)
(check-equal? (sm-apply a* `(,LM)) 'accept)
(check-equal? (sm-apply a* `(,LM a a a)) 'accept)
```

Turing Machines

- Remain in its starting state as long as it has only read as
- If a blank is read then it may move to the final accepting state
- If a b is read then the machine may move to the final rejecting state

Turing Machines

- Remain in its starting state as long as it has only read as
- If a blank is read then it may move to the final accepting state
- If a b is read then the machine may move to the final rejecting state
-
```
;; States (i = head's position)
;;   S: tape[1..i-1] only contains as, starting state
;;   Y: tape[i] = BLANK and tape[1..i-1] only contains as,
;;      final state
;;   N: tape[i] is b, final state
```

Turing Machines

- Remain in its starting state as long as it has only read as
- If a blank is read then it may move to the final accepting state
- If a b is read then the machine may move to the final rejecting state
-
```
;; States (i = head's position)
;;   S: tape[1..i-1] only contains as, starting state
;;   Y: tape[i] = BLANK and tape[1..i-1] only contains as,
;;      final state
;;   N: tape[i] is b, final state
```
- Transition function
```
((S a) (S ,RIGHT))
((S b) (N b))
((S ,BLANK) (Y ,BLANK))
```

Turing Machines

- Implementation

```
;; States (i = head's position)
;;   S: tape[1..i-1] only contains as, starting state
;;   Y: tape[i] = BLANK and tape[1..i-1] only contains as, fina
;;   N: tape[1..i-1] contains a b, final state

;; L = a*
;; PRE: tape = LMw_ AND i = 0
(define a* (make-tm '(S Y N)
                    '(a b)
                    `(((S a) (S ,RIGHT))
                      ((S b) (N b))
                      ((S ,BLANK) (Y ,BLANK)))
                    'S
                    '(Y N)
                    'Y))

;; Tests for a*
(check-equal? (sm-apply a* `(,LM a a a b a a)) 'reject)
(check-equal? (sm-apply a* `(,LM b a a)) 'reject)
(check-equal? (sm-apply a* `(,LM)) 'accept)
(check-equal? (sm-apply a* `(,LM a a a)) 'accept)
```

Turing Machines

- S-INV

```
;; tape natnum → Boolean
;; Purpose: Everything in tape[1..i-1] is an a
(define (S-INV t i)
  (or (= i 0) (andmap (λ (s) (eq? s 'a)) (take (rest t) (sub1 i)))))
;; Tests for S-INV
(check-equal? (S-INV `(,LM b ,BLANK) 2) #f)
(check-equal? (S-INV `(,LM a a b a a) 4) #f)
(check-equal? (S-INV `(,LM b) 1) #t)
(check-equal? (S-INV `(,LM a ,BLANK) 2) #t)
(check-equal? (S-INV `(,LM a a a a ,BLANK) 5) #t)
```

Turing Machines

- S-INV

```
;; tape natnum → Boolean
;; Purpose: Everything in tape[1..i-1] is an a
(define (S-INV t i)
  (or (= i 0) (andmap (λ (s) (eq? s 'a)) (take (rest t) (sub1 i)))))
;; Tests for S-INV
(check-equal? (S-INV `(,LM b ,BLANK) 2) #f)
(check-equal? (S-INV `(,LM a a b a a) 4) #f)
(check-equal? (S-INV `(,LM b) 1) #t)
(check-equal? (S-INV `(,LM a ,BLANK) 2) #t)
(check-equal? (S-INV `(,LM a a a a ,BLANK) 5) #t)
```

-

```
;; tape natnum → Boolean
;; Purpose: Everything in tape[1..i-1] is an a ∧
;;          tape[i] = BLANK
(define (Y-INV t i)
  (and (eq? (list-ref t i) BLANK)
       (andmap (λ (s) (eq? s 'a)) (take (cdr t) (sub1 i)))))
;; Tests for Y-INV
(check-equal? (Y-INV `(,LM b ,BLANK) 2) #f)
(check-equal? (Y-INV `(,LM a b a ,BLANK) 4) #f)
(check-equal? (Y-INV `(,LM ,BLANK) 1) #t)
(check-equal? (Y-INV `(,LM a a a ,BLANK) 4) #t)
```

Turing Machines

- S-INV

```
;; tape natnum → Boolean
;; Purpose: Everything in tape[1..i-1] is an a
(define (S-INV t i)
  (or (= i 0) (andmap (λ (s) (eq? s 'a)) (take (rest t) (sub1 i)))))
;; Tests for S-INV
(check-equal? (S-INV `(,LM b ,BLANK) 2) #f)
(check-equal? (S-INV `(,LM a a b a a) 4) #f)
(check-equal? (S-INV `(,LM b) 1) #t)
(check-equal? (S-INV `(,LM a ,BLANK) 2) #t)
(check-equal? (S-INV `(,LM a a a a ,BLANK) 5) #t)
```

-

```
;; tape natnum → Boolean
;; Purpose: Everything in tape[1..i-1] is an a ∧
;;          tape[i] = BLANK
(define (Y-INV t i)
  (and (eq? (list-ref t i) BLANK)
       (andmap (λ (s) (eq? s 'a)) (take (cdr t) (sub1 i)))))
;; Tests for Y-INV
(check-equal? (Y-INV `(,LM b ,BLANK) 2) #f)
(check-equal? (Y-INV `(,LM a b a ,BLANK) 4) #f)
(check-equal? (Y-INV `(,LM ,BLANK) 1) #t)
(check-equal? (Y-INV `(,LM a a a ,BLANK) 4) #t)
```

- N-INV

```
;; tape natnum → Boolean
;; Purpose: Determine that tape[i] = b
(define (N-INV t i) (eq? (list-ref t i) 'b))
;; Tests for N-INV
(check-equal? (N-INV `(,LM ,BLANK) 1) #f)
(check-equal? (N-INV `(,LM a a ,BLANK) 3) #f)
(check-equal? (N-INV `(,LM a b a b ,BLANK) 4) #t)
(check-equal? (N-INV `(,LM b b b) 2) #t)
(check-equal? (N-INV `(,LM a a a a b ,BLANK) 5) #t)
```

Turing Machines

Theorem

State invariants hold when a is applied to w.*

- The proof, as before, is done by induction on, `n`, the number of steps taken by `a*`

Turing Machines

Theorem

State invariants hold when a is applied to w.*

- The proof, as before, is done by induction on, `n`, the number of steps taken by `a*`

Proof.

Base case: $n = 0$
If no steps are taken a* must be in S and by precondition the head's tape position must be 0. Thus, S-INV holds. □

-

Turing Machines

Proof.

Inductive Step:

Assume: State invariants hold for a computation of length $n = k$ Show: State invariants hold for a computation of length $n = k + 1$ □

•

Turing Machines

Proof.

Inductive Step:

Assume: State invariants hold for a computation of length $n = k$ Show: State invariants hold for a computation of length $n = k + 1$ □

Proof.

Observe that $n = k + 1$ means that $w \neq$ EMP. Let $w = xcy$, such that $x,y \in \Sigma^*$, $|x|=k$, and $c \in \{\Sigma \cup \{\texttt{BLANK}\}\}$. The first $k + 1$ steps may be described as follows:

$$\texttt{(S 0 xcy)} \vdash^* \texttt{(S i cy)} \vdash \texttt{(B j y), where B} \in \{\texttt{S Y N}\}$$

That is, the first k elements of w are traversed in S. Traversing the $k + 1$ element is done in one step that may take the machine to any state. We must show that the state invariants hold for $k + 1$ transition. We make an argument for each rule that may be used: □

Turing Machines

Proof.

Inductive Step:
Assume: State invariants hold for a computation of length $n = k$ Show: State invariants hold for a computation of length $n = k + 1$ □

Proof.

Observe that $n = k + 1$ means that $w \neq$ EMP. Let $w = xcy$, such that $x,y \in \Sigma^*$, $|x|=k$, and $c \in \{\Sigma \cup \{\texttt{BLANK}\}\}$. The first $k + 1$ steps may be described as follows:

$$\texttt{(S 0 xcy)} \vdash^* \texttt{(S i cy)} \vdash \texttt{(B j y), where B} \in \{\texttt{S Y N}\}$$

That is, the first k elements of w are traversed in S. Traversing the $k + 1$ element is done in one step that may take the machine to any state. We must show that the state invariants hold for $k + 1$ transition. We make an argument for each rule that may be used: □

Proof.

((S @) (S R)): By inductive hypothesis, S-INV holds. Using this rule moves the head from position 0 to position 1. Observe that [1..0] is empty and, therefore, everything in that tape interval is an a. Thus, S-INV holds after using this rule. □

Turing Machines

Proof.

Inductive Step:
Assume: State invariants hold for a computation of length n = k Show: State invariants hold for a computation of length n = k + 1 □

Proof.

Observe that n = k + 1 means that w ≠ EMP. Let w = xcy, such that x,y∈Σ^*, |x|=k, and c∈{Σ ∪ {BLANK}}. The first k + 1 steps may be described as follows:

`(S 0 xcy)` $\vdash^*$ `(S i cy)` $\vdash$ `(B j y), where B∈{S Y N}`

That is, the first k elements of w are traversed in S. Traversing the k + 1 element is done in one step that may take the machine to any state. We must show that the state invariants hold for k + 1 transition. We make an argument for each rule that may be used: □

Proof.

((S @) (S R)): By inductive hypothesis, S-INV holds. Using this rule moves the head from position 0 to position 1. Observe that [1..0] is empty and, therefore, everything in that tape interval is an a. Thus, S-INV holds after using this rule. □

Proof.

((S a) (S R)): By inductive hypothesis, S-INV holds. Using this rule moves the head one position to the right from i to i+1. The inductive hypothesis informs us that all tape elements in [1..i-1] are a. The use of this rule means that the tape's i^{th} element is a. This means that all tape elements in [1..i] are a. Thus, S-INV holds after using this rule. □

Turing Machines

Proof.

Inductive Step:
Assume: State invariants hold for a computation of length n = k Show: State invariants hold for a computation of length n = k + 1 □

Proof.

Observe that n = k + 1 means that w ≠ EMP. Let w = xcy, such that x,y∈Σ^*, |x|=k, and c∈{Σ ∪ {BLANK}}. The first k + 1 steps may be described as follows:

`(S 0 xcy)` $\vdash^*$ `(S i cy)` $\vdash$ `(B j y), where B∈{S Y N}`

That is, the first k elements of w are traversed in S. Traversing the k + 1 element is done in one step that may take the machine to any state. We must show that the state invariants hold for k + 1 transition. We make an argument for each rule that may be used: □

Proof.

((S @) (S R)): By inductive hypothesis, S-INV holds. Using this rule moves the head from position 0 to position 1. Observe that [1..0] is empty and, therefore, everything in that tape interval is an a. Thus, S-INV holds after using this rule. □

Proof.

((S a) (S R)): By inductive hypothesis, S-INV holds. Using this rule moves the head one position to the right from i to i+1. The inductive hypothesis informs us that all tape elements in [1..i-1] are a. The use of this rule means that the tape's i^{th} element is a. This means that all tape elements in [1..i] are a. Thus, S-INV holds after using this rule. □

Proof.

((S b) (N b)): By inductive hypothesis, S-INV holds. Using this rule moves the machine to N without moving the head from position i and leaving the tape unchanged. This means that the tape's i^{th} element is b. Thus

Turing Machines

Proof.

Inductive Step:
Assume: State invariants hold for a computation of length n = k Show: State invariants hold for a computation of length n = k + 1 □

Proof.

Observe that n = k + 1 means that w ≠ EMP. Let w = xcy, such that x,y∈Σ^*, |x|=k, and c∈{Σ ∪ {BLANK}}. The first k + 1 steps may be described as follows:

`(S 0 xcy)` $\vdash^*$ `(S i cy)` $\vdash$ `(B j y), where B∈{S Y N}`

That is, the first k elements of w are traversed in S. Traversing the k + 1 element is done in one step that may take the machine to any state. We must show that the state invariants hold for k + 1 transition. We make an argument for each rule that may be used: □

Proof.

((S @) (S R)): By inductive hypothesis, S-INV holds. Using this rule moves the head from position 0 to position 1. Observe that [1..0] is empty and, therefore, everything in that tape interval is an a. Thus, S-INV holds after using this rule. □

Proof.

((S a) (S R)): By inductive hypothesis, S-INV holds. Using this rule moves the head one position to the right from i to i+1. The inductive hypothesis informs us that all tape elements in [1..i-1] are a. The use of this rule means that the tape's i^{th} element is a. This means that all tape elements in [1..i] are a. Thus, S-INV holds after using this rule. □

Proof.

((S b) (N b)): By inductive hypothesis, S-INV holds. Using this rule moves the machine to N without moving the head from position i and leaving the tape unchanged. This means that the tape's i^{th} element is b. Thus

Turing Machines

Theorem

$L = L(a^*)$

- As before, the proof is divided into two lemmas.

Turing Machines

Theorem

$L = L(a^*)$

- As before, the proof is divided into two lemmas.

Lemma

$w \in L \Leftrightarrow w \in L(a^*)$

Proof.

($\Rightarrow$) Assume w $\in$ L. This means w consists of 0 or more as. Given that the state invariants always hold, the only state that a$*$ may be in after consuming w is Y. Thus, w $\in$ L(a*).
($\Leftarrow$) Assume w $\in$ L(a*). Given that state invariants always hold, this means that w contains only as. Therefore, w $\in$ L. $\square$ $\square$

-

Turing Machines

Theorem

$L = L(a^*)$

- As before, the proof is divided into two lemmas.

Lemma

$w \in L \Leftrightarrow w \in L(a^*)$

Proof.

($\Rightarrow$) Assume w $\in$ L. This means w consists of 0 or more as. Given that the state invariants always hold, the only state that a* may be in after consuming w is Y. Thus, w $\in$ L(a*).
($\Leftarrow$) Assume w $\in$ L(a*). Given that state invariants always hold, this means that w contains only as. Therefore, w $\in$ L. □ □

-

Lemma

$w \notin L \Leftrightarrow w \notin L(a^*)$

Proof.

($\Rightarrow$) Assume w $\notin$ L. This means w contains a b. Given that the state invariants always hold, a* cannot be in Y after consuming w. Thus, w $\notin$ L(a*).
($\Leftarrow$) Assume w $\notin$ L(a*). Given that state invariants always hold, this means that w contains a b. Therefore, w $\notin$ L. □ □

-

Turing Machines

- HOMEWORK: 1–3

Turing Machines

- Let's now design a nondeterministic Turing machine
- The transition relation does not have to be a function

Turing Machines

- Let's now design a nondeterministic Turing machine
- The transition relation does not have to be a function
- $\mathtt{L} = \mathtt{a}^* \cup \mathtt{a}^*\mathtt{b}$

Turing Machines

- Let's now design a nondeterministic Turing machine
- The transition relation does not have to be a function
- L = $a^* \cup a^*b$
- Name: a*Ua*b Σ = {a b}

```
;; PRE: tape = LMw AND i = 1
```

Turing Machines

- Let's now design a nondeterministic Turing machine
- The transition relation does not have to be a function
- L = $a^* \cup a^*b$
- Name: a*Ua*b Σ = {a b}

```
;; PRE: tape = LMw AND i = 1
```

- Tests

```
;; Tests for a*Ua*b
(check-equal? (sm-apply a*Ua*b `(,LM b b)     1) 'reject)
(check-equal? (sm-apply a*Ua*b `(,LM a a b a) 1) 'reject)
(check-equal? (sm-apply a*Ua*b `(,LM ,BLANK)  1) 'accept)
(check-equal? (sm-apply a*Ua*b `(,LM a b)     1) 'accept)
(check-equal? (sm-apply a*Ua*b `(,LM a a a)   1) 'accept)
(check-equal? (sm-apply a*Ua*b `(,LM a a a b) 1) 'accept)
```

Turing Machines

- When the machine starts in S nothing has been read
- If the input word is empty the machine moves to accept

Turing Machines

- When the machine starts in `S` nothing has been read
- If the input word is empty the machine moves to accept
- If the first element is an `a` then the machine nondeterministically moves to a state `A` to read `a`* or to a state `B` to read `a`*`b`

Turing Machines

- When the machine starts in `S` nothing has been read
- If the input word is empty the machine moves to accept
- If the first element is an `a` then the machine nondeterministically moves to a state `A` to read `a*` or to a state `B` to read `a*b`
- Upon reading `a*` in `A` the machine may accept
- Upon reading `a*b` in `B` the machine moves to state `C` to determine if the end of the input word has been reached and then moves to either accept or reject

Turing Machines

- When the machine starts in `S` nothing has been read
- If the input word is empty the machine moves to accept
- If the first element is an `a` then the machine nondeterministically moves to a state `A` to read `a*` or to a state `B` to read `a*b`
- Upon reading `a*` in `A` the machine may accept
- Upon reading `a*b` in `B` the machine moves to state `C` to determine if the end of the input word has been reached and then moves to either accept or reject
- From `C` the machine may accept upon reading a blank and reject otherwise.

Turing Machines

- When the machine starts in S nothing has been read
- If the input word is empty the machine moves to accept
- If the first element is an a then the machine nondeterministically moves to a state A to read a^* or to a state B to read a^*b
- Upon reading a^* in A the machine may accept
- Upon reading a^*b in B the machine moves to state C to determine if the end of the input word has been reached and then moves to either accept or reject
- From C the machine may accept upon reading a blank and reject otherwise.
- The states may documented as follows:

```
;; States (i is the position of the head)
;;   S: no tape elements read, starting sate
;;   A: tape[1..i-1] has only a
;;   B: tape[1..i-1] has only a
;;   C: tape[1..i-2] has only a and tape[i-1] = b
;;   Y: tape[i] = BLANK and tape[1..i-1] in a* or a*b,
;;      final accepting state
;;   N: tape[1..i-1] != a* nor a*b, final state
```

Turing Machines

- Transition relation

```
((S ,BLANK) (Y ,BLANK))
((S a) (A ,RIGHT))
((S a) (B ,RIGHT))
((S b) (C ,RIGHT))
```

Turing Machines

- Transition relation

```
((S ,BLANK) (Y ,BLANK))
((S a) (A ,RIGHT))
((S a) (B ,RIGHT))
((S b) (C ,RIGHT))
```

- ```
  ((A a) (A ,RIGHT))
  ((A ,BLANK) (Y ,BLANK))
  ```
  ```

Turing Machines

- Transition relation

```
((S ,BLANK) (Y ,BLANK))
((S a) (A ,RIGHT))
((S a) (B ,RIGHT))
((S b) (C ,RIGHT))
```

- ```
  ((A a) (A ,RIGHT))
  ((A ,BLANK) (Y ,BLANK))
  ```
- ```
  ((B a) (B ,RIGHT))
  ((B b) (C ,RIGHT))
  ```

Turing Machines

- Transition relation

```
((S ,BLANK) (Y ,BLANK))
((S a) (A ,RIGHT))
((S a) (B ,RIGHT))
((S b) (C ,RIGHT))
```

- ```
  ((A a) (A ,RIGHT))
  ((A ,BLANK) (Y ,BLANK))
  ```
- ```
  ((B a) (B ,RIGHT))
  ((B b) (C ,RIGHT))
  ```
- ```
  ((C a) (N ,RIGHT))
  ((C b) (N ,RIGHT))
  ((C ,BLANK) (Y ,BLANK)))
  ```
  ```

Turing Machines

- Implementation

```
;; States (i is the position of the head)
;;   S: no tape elements read, starting sate
;;   A: tape[1..i-1] has only a
;;   B: tape[1..i-1] has only a
;;   C: tape[1..i-2] has only a and tape[i-1] = b
;;   Y: tape[i] = BLANK and tape[1..i-1] = a* or a*b,
;;      final accepting state
;;   N: tape[1..i-1] != a* or a*b, final state
;; L = a* U a*b     PRE: tape = LMw AND i = 1
(define a*Ua*b (make-tm '(S A B C Y N)
                        `(a b)
                        `(((S ,BLANK) (Y ,BLANK))
                          ((S a) (A ,RIGHT))
                          ((S a) (B ,RIGHT))
                          ((S b) (C ,RIGHT))
                          ((A a) (A ,RIGHT))
                          ((A ,BLANK) (Y ,BLANK))
                          ((B a) (B ,RIGHT))
                          ((B b) (C ,RIGHT))
                          ((C a) (N a))
                          ((C b) (N b))
                          ((C ,BLANK) (Y ,BLANK)))
                        'S
                        '(Y N)
                        'Y))
;; Tests for a*Ua*b
(check-equal? (sm-apply a*Ua*b `(,LM b b) 1)     'reject)
(check-equal? (sm-apply a*Ua*b `(,LM a a b a) 1) 'reject)
(check-equal? (sm-apply a*Ua*b `(,LM b) 1) 'accept)
(check-equal? (sm-apply a*Ua*b `(,LM ,BLANK) 1)  'accept)
(check-equal? (sm-apply a*Ua*b `(,LM a b) 1)     'accept)
(check-equal? (sm-apply a*Ua*b `(,LM a a a) 1)   'accept)
```

Turing Machines

- `;; tape natnum → Boolean  Purpose: Determine that no tape elements read`
  `(define (S-INV t i) (= i 1))`

Turing Machines

- ;; tape natnum → Boolean Purpose: Determine that no tape elements read
  (define (S-INV t i) (= i 1))
- ;; tape natnum → Boolean Purpose: Determine that tape[1..i-1] only has a
  (define (A-INV t i)
    (and (>= i 2) (andmap (λ (s) (eq? s 'a)) (take (rest t) (sub1 i)))))

Turing Machines

- ```
  ;; tape natnum → Boolean    Purpose: Determine that no tape elements read
  (define (S-INV t i) (= i 1))
  ```
- ```
  ;; tape natnum → Boolean   Purpose: Determine that tape[1..i-1] only has a
  (define (A-INV t i)
    (and (>= i 2)  (andmap (λ (s) (eq? s 'a)) (take (rest t) (sub1 i)))))
  ```
- ```
  ;; tape natnum → Boolean
  ;; Purpose: Determine that tape[1..i-2] has only a and tape[i-1] = b
  (define (C-INV t i)
    (and (>= i 2)
         (andmap (λ (s) (eq? s 'a)) (take (rest t) (- i 2)))
         (eq? (list-ref t (sub1 i)) 'b)))
  ```
  ```

Turing Machines

- ```
  ;; tape natnum → Boolean    Purpose: Determine that no tape elements read
  (define (S-INV t i) (= i 1))
  ```
- ```
  ;; tape natnum → Boolean   Purpose: Determine that tape[1..i-1] only has a
  (define (A-INV t i)
    (and (>= i 2)  (andmap (λ (s) (eq? s 'a)) (take (rest t) (sub1 i)))))
  ```
- ```
  ;; tape natnum → Boolean
  ;; Purpose: Determine that tape[1..i-2] has only a and tape[i-1] = b
  (define (C-INV t i)
    (and (>= i 2)
         (andmap (λ (s) (eq? s 'a)) (take (rest t) (- i 2)))
         (eq? (list-ref t (sub1 i)) 'b)))
  ```
- ```
  ;; tape natnum → Boolean
  ;; Purpose: Determine that tape[i] = BLANK and tape[1..i-1] = a* or
  ;;          tape[1..i-1] = a*b
  (define (Y-INV t i)
    (or (and (= i 2) (eq? (list-ref t (sub1 i)) BLANK))
        (andmap (λ (s) (eq? s 'a)) (take (rest t) (sub1 i)))
        (let* [(front (takef (rest t) (λ (s) (eq? s 'a))))
               (back (takef (drop t (add1 (length front)))
                            (λ (s) (not (eq? s BLANK)))))]
          (equal? back '(b))))))
  ```

Turing Machines

- ```
  ;; tape natnum → Boolean    Purpose: Determine that no tape elements read
  (define (S-INV t i) (= i 1))
  ```
- ```
  ;; tape natnum → Boolean   Purpose: Determine that tape[1..i-1] only has a
  (define (A-INV t i)
    (and (>= i 2)  (andmap (λ (s) (eq? s 'a)) (take (rest t) (sub1 i)))))
  ```
- ```
  ;; tape natnum → Boolean
  ;; Purpose: Determine that tape[1..i-2] has only a and tape[i-1] = b
  (define (C-INV t i)
    (and (>= i 2)
         (andmap (λ (s) (eq? s 'a)) (take (rest t) (- i 2)))
         (eq? (list-ref t (sub1 i)) 'b)))
  ```
- ```
  ;; tape natnum → Boolean
  ;; Purpose: Determine that tape[i] = BLANK and tape[1..i-1] = a* or
  ;;          tape[1..i-1] = a*b
  (define (Y-INV t i)
    (or (and (= i 2) (eq? (list-ref t (sub1 i)) BLANK))
        (andmap (λ (s) (eq? s 'a)) (take (rest t) (sub1 i)))
        (let* [(front (takef (rest t) (λ (s) (eq? s 'a))))
               (back (takef (drop t (add1 (length front)))
                            (λ (s) (not (eq? s BLANK)))))]
          (equal? back '(b))))))
  ```
- ```
  ;; tape natnum → Boolean   Purpose: Determine that tape[1..i-1] != a* or a*b
  (define (N-INV t i)
    (and (not (andmap (λ (s) (eq? s 'a))
                      (take (rest t) (sub1 i))))
         (let* [(front (takef (rest t) (λ (s) (eq? s 'a))))
                (back (takef (drop t (add1 (length front)))
                             (λ (s) (not (eq? s BLANK)))))]
           (not (equal? back '(b))))))
  ```
  ```

Turing Machines

Theorem

*State invariants hold when a*Ua*b is applied to w.*

- The proof, as before, is done by induction on, `n`, the number of steps taken by `a*Ua*b`. Let `a*Ua*b` = `(make-tm K Σ R S F Y)`.

Proof.

□

Turing Machines

Theorem

*State invariants hold when a*Ua*b is applied to w.*

- The proof, as before, is done by induction on, `n`, the number of steps taken by `a*Ua*b`. Let `a*Ua*b` = `(make-tm K Σ R S F Y)`.

Proof.

- Base case: $n = 0$
  If no steps are taken a*Ua*b may only be in S. By precondition, the head's position is 1. This means S-INV holds.

□

Turing Machines

Proof.

- Inductive Step:
  Assume: State invariants hold for a computation of length $n = k$
  Show: State invariants hold for a computation of length $n = k + 1$

□

Turing Machines

Proof.

- Inductive Step:
  Assume: State invariants hold for a computation of length $n = k$
  Show: State invariants hold for a computation of length $n = k + 1$
- Let $w = xcy$, such that $x,y \in \Sigma^*$, $|x|=k$, and $c \in \{\Sigma \cup \{BLANK\}\}$. The first $k + 1$ steps:

  `(S 1 xcy)` $\vdash^*$ `(U r xcy)` $\vdash$ `(V s xcy), where V∈K ∧ U∈K-{N Y}`
- That is, the first k transitions take the machine to state U and move the head to position r without changing the contents of the tape
- The $k + 1$ transition takes the machine to state V and leaves the head in position s without changing the contents of the tape
- We must show that the state invariant holds for the $k + 1$ transition
- Note that a rule of the form ((I @) (I ,RIGHT)) is never used because the machine never moves left and by precondition the head starts in position 1

□

Turing Machines

Proof.

We make an argument for each rule that may be used:

- ((S ,BLANK) (Y ,BLANK)): By inductive hypothesis, S-INV holds. This means that before using this rule nothing has been read from the input word because the head is in position 1. Reading the blank means the input word is empty. Thus, Y-INV holds.

□

Turing Machines

Proof.

We make an argument for each rule that may be used:

- ((S ,BLANK) (Y ,BLANK)): By inductive hypothesis, S-INV holds. This means that before using this rule nothing has been read from the input word because the head is in position 1. Reading the blank means the input word is empty. Thus, Y-INV holds.
- ((S a) (A ,RIGHT)): By inductive hypothesis, S-INV holds. This means that before using this rule nothing has been read from the input word because the head is in position 1. Using this rule means that the read part of the input word only contains a and that the head moves to position 2. Therefore, A-INV holds.

□

Turing Machines

Proof.

We make an argument for each rule that may be used:

- ((S ,BLANK) (Y ,BLANK)): By inductive hypothesis, S-INV holds. This means that before using this rule nothing has been read from the input word because the head is in position 1. Reading the blank means the input word is empty. Thus, Y-INV holds.
- ((S a) (A ,RIGHT)): By inductive hypothesis, S-INV holds. This means that before using this rule nothing has been read from the input word because the head is in position 1. Using this rule means that the read part of the input word only contains a and that the head moves to position 2. Therefore, A-INV holds.
- ((S a) (B ,RIGHT)): By inductive hypothesis, S-INV holds. This means that before using this rule nothing has been read from the input word because the head is in position 1. Using this rule means that the read part of the input word only contains a and that the head moves to position 2. Therefore, B-INV holds.

□

Turing Machines

Proof.

We make an argument for each rule that may be used:

- ((S ,BLANK) (Y ,BLANK)): By inductive hypothesis, S-INV holds. This means that before using this rule nothing has been read from the input word because the head is in position 1. Reading the blank means the input word is empty. Thus, Y-INV holds.
- ((S a) (A ,RIGHT)): By inductive hypothesis, S-INV holds. This means that before using this rule nothing has been read from the input word because the head is in position 1. Using this rule means that the read part of the input word only contains a and that the head moves to position 2. Therefore, A-INV holds.
- ((S a) (B ,RIGHT)): By inductive hypothesis, S-INV holds. This means that before using this rule nothing has been read from the input word because the head is in position 1. Using this rule means that the read part of the input word only contains a and that the head moves to position 2. Therefore, B-INV holds.
- ((S b) (C ,RIGHT)): By inductive hypothesis, S-INV holds. This means that before using this rule nothing has been read from the input word because the head is in position 1. Using this rule means that the read part of the input word only contains b and that the head moves to position 2. Therefore, C-INV holds.

□

Turing Machines

Proof.

We make an argument for each rule that may be used:

- ((S ,BLANK) (Y ,BLANK)): By inductive hypothesis, S-INV holds. This means that before using this rule nothing has been read from the input word because the head is in position 1. Reading the blank means the input word is empty. Thus, Y-INV holds.
- ((S a) (A ,RIGHT)): By inductive hypothesis, S-INV holds. This means that before using this rule nothing has been read from the input word because the head is in position 1. Using this rule means that the read part of the input word only contains a and that the head moves to position 2. Therefore, A-INV holds.
- ((S a) (B ,RIGHT)): By inductive hypothesis, S-INV holds. This means that before using this rule nothing has been read from the input word because the head is in position 1. Using this rule means that the read part of the input word only contains a and that the head moves to position 2. Therefore, B-INV holds.
- ((S b) (C ,RIGHT)): By inductive hypothesis, S-INV holds. This means that before using this rule nothing has been read from the input word because the head is in position 1. Using this rule means that the read part of the input word only contains b and that the head moves to position 2. Therefore, C-INV holds.
- ((A a) (A ,RIGHT)): By inductive hypothesis, A-INV holds. This means that the read part of the input word is a member of a^* and that, the head's position, $i \geq 2$. Reading the a means the read part of the input word continues to be a member of a^* and $i \geq 2$ continues to hold. Thus, A-INV holds.

□

Turing Machines

Proof.

We make an argument for each rule that may be used:

- ((S ,BLANK) (Y ,BLANK)): By inductive hypothesis, S-INV holds. This means that before using this rule nothing has been read from the input word because the head is in position 1. Reading the blank means the input word is empty. Thus, Y-INV holds.
- ((S a) (A ,RIGHT)): By inductive hypothesis, S-INV holds. This means that before using this rule nothing has been read from the input word because the head is in position 1. Using this rule means that the read part of the input word only contains a and that the head moves to position 2. Therefore, A-INV holds.
- ((S a) (B ,RIGHT)): By inductive hypothesis, S-INV holds. This means that before using this rule nothing has been read from the input word because the head is in position 1. Using this rule means that the read part of the input word only contains a and that the head moves to position 2. Therefore, B-INV holds.
- ((S b) (C ,RIGHT)): By inductive hypothesis, S-INV holds. This means that before using this rule nothing has been read from the input word because the head is in position 1. Using this rule means that the read part of the input word only contains b and that the head moves to position 2. Therefore, C-INV holds.
- ((A a) (A ,RIGHT)): By inductive hypothesis, A-INV holds. This means that the read part of the input word is a member of a^* and that, the head's position, $i \geq 2$. Reading the a means the read part of the input word continues to be a member of a^* and $i \geq 2$ continues to hold. Thus, A-INV holds.
- ((A ,BLANK) (Y ,BLANK)): By inductive hypothesis, A-INV holds. This means that the read part of the input word is a member of a^*. Reading a blank means the input word is a member of a^*. Thus, Y-INV holds.

□

Turing Machines

Proof.

- ((B a) (B ,RIGHT)): By inductive hypothesis, B-INV holds. This means that the read part of the input word is a member of a^* and that, the head's position, $i \geq 2$. Reading the a and moving the head right means the read part of the input word continues to be a member of a^* and $i \geq 2$ continues to hold. Thus, B-INV holds.

□

Turing Machines

Proof.

- ((B a) (B ,RIGHT)): By inductive hypothesis, B-INV holds. This means that the read part of the input word is a member of a^* and that, the head's position, $i \geq 2$. Reading the a and moving the head right means the read part of the input word continues to be a member of a^* and $i \geq 2$ continues to hold. Thus, B-INV holds.
- ((B b) (C ,RIGHT)): By inductive hypothesis, B-INV holds. This means that the read part of the input word is a member of a^* and, the head's position, $i \geq 2$. Reading a b means the read part of the input word is a member of a^*b and that $i \geq 2$ continues to hold. Thus, C-INV holds.

□

Turing Machines

Proof.

- ((B a) (B ,RIGHT)): By inductive hypothesis, B-INV holds. This means that the read part of the input word is a member of a^* and that, the head's position, $i \geq 2$. Reading the a and moving the head right means the read part of the input word continues to be a member of a^* and $i \geq 2$ continues to hold. Thus, B-INV holds.
- ((B b) (C ,RIGHT)): By inductive hypothesis, B-INV holds. This means that the read part of the input word is a member of a^* and, the head's position, $i \geq 2$. Reading a b means the read part of the input word is a member of a^*b and that $i \geq 2$ continues to hold. Thus, C-INV holds.
- ((C a) (N ,RIGHT)): By inductive hypothesis, C-INV holds. This means that the read part of the input word is a member of a^*b. Reading an a means the input word is not a member of a^* nor a^*b. Thus, N-INV holds.

□

Turing Machines

Proof.

- ((B a) (B ,RIGHT)): By inductive hypothesis, B-INV holds. This means that the read part of the input word is a member of a^* and that, the head's position, $i \geq 2$. Reading the a and moving the head right means the read part of the input word continues to be a member of a^* and $i \geq 2$ continues to hold. Thus, B-INV holds.
- ((B b) (C ,RIGHT)): By inductive hypothesis, B-INV holds. This means that the read part of the input word is a member of a^* and, the head's position, $i \geq 2$. Reading a b means the read part of the input word is a member of a^*b and that $i \geq 2$ continues to hold. Thus, C-INV holds.
- ((C a) (N ,RIGHT)): By inductive hypothesis, C-INV holds. This means that the read part of the input word is a member of a^*b. Reading an a means the input word is not a member of a^* nor a^*b. Thus, N-INV holds.
- ((C b) (N ,RIGHT)): By inductive hypothesis, C-INV holds. This means that the read part of the input word is a member of a^*b. Reading an b means the input word is not a member of a^* nor a^*b. Thus, N-INV holds.

□

Turing Machines

Proof.

- ((B a) (B ,RIGHT)): By inductive hypothesis, B-INV holds. This means that the read part of the input word is a member of a^* and that, the head's position, $i \geq 2$. Reading the a and moving the head right means the read part of the input word continues to be a member of a^* and $i \geq 2$ continues to hold. Thus, B-INV holds.
- ((B b) (C ,RIGHT)): By inductive hypothesis, B-INV holds. This means that the read part of the input word is a member of a^* and, the head's position, $i \geq 2$. Reading a b means the read part of the input word is a member of a^*b and that $i \geq 2$ continues to hold. Thus, C-INV holds.
- ((C a) (N ,RIGHT)): By inductive hypothesis, C-INV holds. This means that the read part of the input word is a member of a^*b. Reading an a means the input word is not a member of a^* nor a^*b. Thus, N-INV holds.
- ((C b) (N ,RIGHT)): By inductive hypothesis, C-INV holds. This means that the read part of the input word is a member of a^*b. Reading an b means the input word is not a member of a^* nor a^*b. Thus, N-INV holds.
- ((C ,BLANK) (Y ,BLANK)): By inductive hypothesis, C-INV holds. This means that the read part of the input word is a member of a^*b. Reading a blank means the input word is a member of a^*b. Thus, Y-INV holds.

□

Turing Machines

Theorem

$L = L(a^*Ua^*b)$

Turing Machines

Theorem

$L = L(a^*Ua^*b)$

Lemma

$w \in L \Leftrightarrow w \in L(a^*Ua^*b)$

-

Lemma

$w \notin L \Leftrightarrow w \notin L(a^*Ua^*b)$

-

Turing Machines

Theorem

$L = L(a^*Ua^*b)$

Lemma

$w \in L \Leftrightarrow w \in L(a^*Ua^*b)$

•

Proof.

($\Rightarrow$) Assume w $\in$ L. This means that w $\in$ a^* or w $\in$ a^*b. Given that state invariants always hold, a*Ua*b must halt in Y after reading w. Thus, w $\in$ L(a*Ua*b).
($\Leftarrow$) Assume w $\in$ L(a*Ua*b). This means that a*Ua*b halts in Y after consuming w. Given that the invariants always hold, w $\in$ a^* or w $\in$ a^*b. Thus, w $\in$ L. □

•

Lemma

$w \notin L \Leftrightarrow w \notin L(a^*Ua^*b)$

•

Turing Machines

Theorem

$L = L(a^*Ua^*b)$

Lemma

$w \in L \Leftrightarrow w \in L(a^*Ua^*b)$

-

Proof.

($\Rightarrow$) Assume w $\in$ L. This means that w $\in$ a^* or w $\in$ a^*b. Given that state invariants always hold, a*Ua*b must halt in Y after reading w. Thus, w $\in$ L(a*Ua*b).
($\Leftarrow$) Assume w $\in$ L(a*Ua*b). This means that a*Ua*b halts in Y after consuming w. Given that the invariants always hold, w $\in$ a^* or w $\in$ a^*b. Thus, w $\in$ L. □

-

Lemma

$w \notin L \Leftrightarrow w \notin L(a^*Ua^*b)$

-

Proof.

($\Rightarrow$) Assume w $\notin$ L. This means that w $\notin$ a^* and w $\notin$ a^*b. Given that state invariants always hold, a*Ua*b does not halt in Y after reading w. Thus, w $\notin$ L(a*Ua*b).
($\Leftarrow$) Assume w $\notin$ L(a*Ua*b). This means that a*Ua*b does not halt in Y after reading w. Given that state invariants always hold, this means that w $\notin$ a^* and w $\notin$ a^*b. Thus, w $\notin$ L. □

-

Turing Machines

HOMEWORK

- HOMEWORK: 4–6

Turing Machines

- You may suspect that `tms` can decide any regular language
- We ought to design and implement a constructor for `tms` that takes as input a `dfa` and decides the given `dfa`'s language

Turing Machines

- You may suspect that `tms` can decide any regular language
- We ought to design and implement a constructor for `tms` that takes as input a `dfa` and decides the given `dfa`'s language
- Think of a `dfa` as a `tm` that never moves left and that never writes to the tape.
- A Turing machine may simulate a `dfa` rule:

  ((state$_1$ symbol) (state$_2$ RIGHT))

Turing Machines

- You may suspect that `tms` can decide any regular language
- We ought to design and implement a constructor for `tms` that takes as input a `dfa` and decides the given `dfa`'s language
- Think of a `dfa` as a `tm` that never moves left and that never writes to the tape.
- A Turing machine may simulate a `dfa` rule:

  $((\texttt{state}_1\ \texttt{symbol})\ (\texttt{state}_2\ \texttt{RIGHT}))$
- Handling a `dfa`'s final states requires a little care
- We cannot simply make every final state in the `dfa` a final state in the constructed `tm`
- Need a mechanism for the constructed `tm` to move to its accepting final state when the `dfa` would be in a final state with no more input to consume

Turing Machines

- You may suspect that `tms` can decide any regular language
- We ought to design and implement a constructor for `tms` that takes as input a `dfa` and decides the given `dfa`'s language
- Think of a `dfa` as a `tm` that never moves left and that never writes to the tape.
- A Turing machine may simulate a `dfa` rule:

  $((\texttt{state}_1\ \texttt{symbol})\ (\texttt{state}_2\ \texttt{RIGHT}))$
- Handling a `dfa`'s final states requires a little care
- We cannot simply make every final state in the `dfa` a final state in the constructed `tm`
- Need a mechanism for the constructed `tm` to move to its accepting final state when the `dfa` would be in a final state with no more input to consume
- The `tm` needs transition rules from every `dfa` final state on `BLANK` to the `tm`'s final accepting state

Turing Machines

- ```
  ;; DFA for testing
  ;; L(M) = ab*
  (define M (make-dfa '(S F) '(a b) 'S '(F) '((S a F) (F b F))))
  ```
  ```

Turing Machines

- ```
  ;; DFA for testing
  ;; L(M) = ab*
  (define M (make-dfa '(S F) '(a b) 'S '(F) '((S a F) (F b F))))
  ```

- ```
  ;; Tests for dfa2tm
  (define M-tm (dfa2tm M))
  (check-equal? (sm-apply M '()) (sm-apply M-tm `(,LM)))
  (check-equal? (sm-apply M '(b b)) (sm-apply M-tm `(,LM b b)))
  (check-equal? (sm-apply M '(a b b a a)) (sm-apply M-tm `(,LM a b b a a)))
  (check-equal? (sm-apply M '(a)) (sm-apply M-tm `(,LM a)))
  (check-equal? (sm-apply M '(a b)) (sm-apply M-tm `(,LM a b)))
  (check-equal? (sm-apply M '(a b b)) (sm-apply M-tm `(,LM a b b)))
  (check-equal? (sm-apply M '(a b b b b b)) (sm-apply M-tm `(,LM a b b b b b)))
  ```

Turing Machines

- ```
  ;; DFA for testing
  ;; L(M) = ab*
  (define M (make-dfa '(S F) '(a b) 'S '(F) '((S a F) (F b F))))
  ```
- ```
  ;; dfa → tm
  ;; Purpose: Build a tm for the language of the given dfa
  (define (dfa2tm m)
    (let [(accept-state (gen-state (sm-states m)))]new state-generating function
  ```
- ```
  ;; Tests for dfa2tm
  (define M-tm (dfa2tm M))
  (check-equal? (sm-apply M '()) (sm-apply M-tm `(,LM)))
  (check-equal? (sm-apply M '(b b)) (sm-apply M-tm `(,LM b b)))
  (check-equal? (sm-apply M '(a b b a a)) (sm-apply M-tm `(,LM a b b a a)))
  (check-equal? (sm-apply M '(a)) (sm-apply M-tm `(,LM a)))
  (check-equal? (sm-apply M '(a b)) (sm-apply M-tm `(,LM a b)))
  (check-equal? (sm-apply M '(a b b)) (sm-apply M-tm `(,LM a b b)))
  (check-equal? (sm-apply M '(a b b b b b)) (sm-apply M-tm `(,LM a b b b b b)))
  ```
  ```

Turing Machines

- ```
  ;; DFA for testing
  ;; L(M) = ab*
  (define M (make-dfa '(S F) '(a b) 'S '(F) '((S a F) (F b F))))
  ```
- ```
  ;; dfa → tm
  ;; Purpose: Build a tm for the language of the given dfa
  (define (dfa2tm m)
    (let [(accept-state (gen-state (sm-states m)))]new state-generating function
  ```
- ```
      (make-tm (cons accept-state (sm-states m))
               (sm-sigma m)
  ```
- ```
  ;; Tests for dfa2tm
  (define M-tm (dfa2tm M))
  (check-equal? (sm-apply M '()) (sm-apply M-tm `(,LM)))
  (check-equal? (sm-apply M '(b b)) (sm-apply M-tm `(,LM b b)))
  (check-equal? (sm-apply M '(a b b a a)) (sm-apply M-tm `(,LM a b b a a)))
  (check-equal? (sm-apply M '(a)) (sm-apply M-tm `(,LM a)))
  (check-equal? (sm-apply M '(a b)) (sm-apply M-tm `(,LM a b)))
  (check-equal? (sm-apply M '(a b b)) (sm-apply M-tm `(,LM a b b)))
  (check-equal? (sm-apply M '(a b b b b b)) (sm-apply M-tm `(,LM a b b b b b)))
  ```

Turing Machines

```
;; DFA for testing
;; L(M) = ab*
(define M (make-dfa '(S F) '(a b) 'S '(F) '((S a F) (F b F))))
```

```
;; dfa → tm
;; Purpose: Build a tm for the language of the given dfa
(define (dfa2tm m)
  (let [(accept-state (gen-state (sm-states m)))]new state-generating function
    (make-tm (cons accept-state (sm-states m))
             (sm-sigma m)
             (append (map (λ (f) (list (list f BLANK)
                                       (list accept-state BLANK)))
                          (sm-finals m))
                     (map (λ (r) (list (list (first r) (second r))
                                       (list (third r) RIGHT)))
                          (sm-rules m)))
```

```
;; Tests for dfa2tm
(define M-tm (dfa2tm M))
(check-equal? (sm-apply M '()) (sm-apply M-tm `(,LM)))
(check-equal? (sm-apply M '(b b)) (sm-apply M-tm `(,LM b b)))
(check-equal? (sm-apply M '(a b b a a)) (sm-apply M-tm `(,LM a b b a a)))
(check-equal? (sm-apply M '(a)) (sm-apply M-tm `(,LM a)))
(check-equal? (sm-apply M '(a b)) (sm-apply M-tm `(,LM a b)))
(check-equal? (sm-apply M '(a b b)) (sm-apply M-tm `(,LM a b b)))
(check-equal? (sm-apply M '(a b b b b b)) (sm-apply M-tm `(,LM a b b b b b)))
```

Turing Machines

```
;; DFA for testing
;; L(M) = ab*
(define M (make-dfa '(S F) '(a b) 'S '(F) '((S a F) (F b F))))
```

```
;; dfa → tm
;; Purpose: Build a tm for the language of the given dfa
(define (dfa2tm m)
  (let [(accept-state (gen-state (sm-states m)))]new state-generating function
    (make-tm (cons accept-state (sm-states m))
             (sm-sigma m)
             (append (map (λ (f) (list (list f BLANK)
                                       (list accept-state BLANK)))
                          (sm-finals m))
                     (map (λ (r) (list (list (first r) (second r))
                                       (list (third r) RIGHT)))
                          (sm-rules m)))
             (sm-start m)
             (list accept-state)
             accept-state)))
```

```
;; Tests for dfa2tm
(define M-tm (dfa2tm M))
(check-equal? (sm-apply M '()) (sm-apply M-tm `(,LM)))
(check-equal? (sm-apply M '(b b)) (sm-apply M-tm `(,LM b b)))
(check-equal? (sm-apply M '(a b b a a)) (sm-apply M-tm `(,LM a b b a a)))
(check-equal? (sm-apply M '(a)) (sm-apply M-tm `(,LM a)))
(check-equal? (sm-apply M '(a b)) (sm-apply M-tm `(,LM a b)))
(check-equal? (sm-apply M '(a b b)) (sm-apply M-tm `(,LM a b b)))
(check-equal? (sm-apply M '(a b b b b b)) (sm-apply M-tm `(,LM a b b b b b)))
```

Turing Machines

- HOMEWORK: 8–9

Turing Machines

- tms can perform computations that neither dfas nor pdas can perform
- $L = a^n b^n c^n$

Turing Machines

- tms can perform computations that neither dfas nor pdas can perform
- $L = a^n b^n c^n$
- The input word is preceded by a blank and the head starts on this blank

Turing Machines

- tms can perform computations that neither dfas nor pdas can perform
- $L = a^n b^n c^n$
- The input word is preceded by a blank and the head starts on this blank
- Three phases:
  1. Determine if word is empty. If so, move to accept
  2. Determine if the word is in $a^* b^* c^*$. If not, halt and reject
  3. Check that there are an equal number of as, bs, and cs

Turing Machines

- Algorithm
  1. The machine starts with the head in position 1 that must be a blank.

Turing Machines

- Algorithm
  1. The machine starts with the head in position 1 that must be a blank.
  2. The machine moves the head to the right to determine if the input word is empty. If the input word is empty then the machine moves to accept. Otherwise, it determines if the input word is in $a^+b^+c^+$.

Turing Machines

- Algorithm
  1. The machine starts with the head in position 1 that must be a blank.
  2. The machine moves the head to the right to determine if the input word is empty. If the input word is empty then the machine moves to accept. Otherwise, it determines if the input word is in $a^+b^+c^+$.
  3. If the input word is not empty, the machine
     1. Tries to read as before a b. If a c or a blank are read then the machine halts and rejects.
     2. tries to read bs before a c. If a blank or an a is read then the machine halts and rejects.
     3. tries to read cs before a blank. If an a or a b is read then the machine halts and rejects.

Turing Machines

- Algorithm
  1. The machine starts with the head in position 1 that must be a blank.
  2. The machine moves the head to the right to determine if the input word is empty. If the input word is empty then the machine moves to accept. Otherwise, it determines if the input word is in $a^+b^+c^+$.
  3. If the input word is not empty, the machine
     1. Tries to read as before a b. If a c or a blank are read then the machine halts and rejects.
     2. tries to read bs before a c. If a blank or an a is read then the machine halts and rejects.
     3. tries to read cs before a blank. If an a or a b is read then the machine halts and rejects.
     4. The machine moves the head to position 1.

Turing Machines

- Algorithm
  1. The machine starts with the head in position 1 that must be a blank.
  2. The machine moves the head to the right to determine if the input word is empty. If the input word is empty then the machine moves to accept. Otherwise, it determines if the input word is in $a^+b^+c^+$.
  3. If the input word is not empty, the machine
     1. Tries to read as before a b. If a c or a blank are read then the machine halts and rejects.
     2. tries to read bs before a c. If a blank or an a is read then the machine halts and rejects.
     3. tries to read cs before a blank. If an a or a b is read then the machine halts and rejects.
     4. The machine moves the head to position 1.
     5. The machine skips xs to the right until it reaches an a.

Turing Machines

- Algorithm
  1. The machine starts with the head in position 1 that must be a blank.
  2. The machine moves the head to the right to determine if the input word is empty. If the input word is empty then the machine moves to accept. Otherwise, it determines if the input word is in $a^+b^+c^+$.
  3. If the input word is not empty, the machine
     1. Tries to read as before a b. If a c or a blank are read then the machine halts and rejects.
     2. tries to read bs before a c. If a blank or an a is read then the machine halts and rejects.
     3. tries to read cs before a blank. If an a or a b is read then the machine halts and rejects.
     4. The machine moves the head to position 1.
     5. The machine skips xs to the right until it reaches an a.
     6. The machine substitutes an a and nondeterministically decides to either:

```
  1a. Substitute a b.
  1b. Substitute a c and loop.
or
  2a. Substitute the last b.
  2b. Substitute the last c.
  2c. Move the head to the right.
  2d. Move to accept if a blank is read.
```

- Suggests that 14 states are needed: one for each step outlined and a final accepting state

Turing Machines

- Start

```
;; S: i = 1 ∧ tape[i] = BLANK, starting state
```

Turing Machines

- Start

```
;; S: i = 1 ∧ tape[i] = BLANK, starting state
```

- Input is empty

```
J: i = 2 AND tape[i-1] = BLANK
```

Turing Machines

- Start

  ;; S: i = 1 $\wedge$ tape[i] = BLANK, starting state

- Input is empty

  J: i = 2 AND tape[i-1] = BLANK

- Determine if in $a^*b^*c^*$

  A: tape[2..i-1] = a^+ $\wedge$ i > 2
  B: tape[2..i-1] = a^+b^+ $\wedge$ i > 3
  C: tape[2..i-1] = $a^+b^+c^+$ $\wedge$ i > 4

Turing Machines

- Start

  ;; S: i = 1 $\wedge$ tape[i] = BLANK, starting state

- Input is empty

  J: i = 2 AND tape[i-1] = BLANK

- Determine if in $a^*b^*c^*$

  A: tape[2..i-1] = a^+ $\wedge$ i > 2
  B: tape[2..i-1] = a^+b^+ $\wedge$ i > 3
  C: tape[2..i-1] = $a^+b^+c^+$ $\wedge$ i > 4

- Equal number of a, b, and c substituted

  D: w = $x^n a^+ x^n b^+ x^n c^+$ $\wedge$ i $\geq$ 1

Turing Machines

- Start

  ;; S: i = 1 $\wedge$ tape[i] = BLANK, starting state

- Input is empty

  J: i = 2 AND tape[i-1] = BLANK

- Determine if in $a^*b^*c^*$

  A: tape[2..i-1] = a^+ $\wedge$ i > 2
  B: tape[2..i-1] = a^+b^+ $\wedge$ i > 3
  C: tape[2..i-1] = $a^+b^+c^+$ $\wedge$ i > 4

- Equal number of a, b, and c substituted

  D: w = $x^n a^+ x^n b^+ x^n c^+$ $\wedge$ i $\geq$ 1

- Substituting non-last a, b, and c

  E: i > 1 $\wedge$ w = $x^n a^+ x^n b^+ x^n c^+$
  F: i > 1 $\wedge$ w = $x^{n+1} a^+ x^n bb^+ x^n cc^+$
  G: i > 3 $\wedge$ w = $x^{n+1} a^+ x^{n+1} b^+ x^n cc^+$

Turing Machines

- Start

  ;; S: i = 1 $\wedge$ tape[i] = BLANK, starting state

- Input is empty

  J: i = 2 AND tape[i-1] = BLANK

- Determine if in $a^*b^*c^*$

  A: tape[2..i-1] = a^+ $\wedge$ i > 2
  B: tape[2..i-1] = a^+b^+ $\wedge$ i > 3
  C: tape[2..i-1] = $a^+b^+c^+$ $\wedge$ i > 4

- Equal number of a, b, and c substituted

  D: w = $x^n a^+ x^n b^+ x^n c^+$ $\wedge$ i $\geq$ 1

- Substituting non-last a, b, and c

  E: i > 1 $\wedge$ w = $x^n a^+ x^n b^+ x^n c^+$
  F: i > 1 $\wedge$ w = $x^{n+1} a^+ x^n bb^+ x^n cc^+$
  G: i > 3 $\wedge$ w = $x^{n+1} a^+ x^{n+1} b^+ x^n cc^+$

- Substituting last a, b, and c

  H: i > 1 $\wedge$ w=x^+bx^+c $\wedge$ |xs| remainder 3 = 1 $\wedge$ $|x^*b|=2|x^*c|$
  I: i > 2 $\wedge$ w=x^+c $\wedge$ |xs| remainder 3 = 2

Turing Machines

- Start

  ;; S: i = 1 $\wedge$ tape[i] = BLANK, starting state

- Input is empty

  J: i = 2 AND tape[i-1] = BLANK

- Determine if in $a^*b^*c^*$

  A: tape[2..i-1] = a^+ $\wedge$ i > 2
  B: tape[2..i-1] = a^+b^+ $\wedge$ i > 3
  C: tape[2..i-1] = $a^+b^+c^+$ $\wedge$ i > 4

- Equal number of a, b, and c substituted

  D: w = $x^na^+x^nb^+x^nc^+$ $\wedge$ i $\geq$ 1

- Substituting non-last a, b, and c

  E: i > 1 $\wedge$ w = $x^na^+x^nb^+x^nc^+$
  F: i > 1 $\wedge$ w = $x^{n+1}a^+x^nbb^+x^ncc^+$
  G: i > 3 $\wedge$ w = $x^{n+1}a^+x^{n+1}b^+x^ncc^+$

- Substituting last a, b, and c

  H: i > 1 $\wedge$ w=x^+bx^+c $\wedge$ |xs| remainder 3 = 1 $\wedge$ $|x^*b|=2|x^*c|$
  I: i > 2 $\wedge$ w=x^+c $\wedge$ |xs| remainder 3 = 2

- Skip last c substituted and check if blank read

  K: i > 3 $\wedge$ w = xxxx^* $\wedge$ |xs| remainder 3 = 0 $\wedge$ tape[i]=x
  $\wedge$ i=|w|+1
  L: i > 4 $\wedge$ w = xxxx^* $\wedge$ |xs| remainder 3 = 0 $\wedge$ i=|w|+2

Turing Machines

- Start

  ;; S: i = 1 $\wedge$ tape[i] = BLANK, starting state

- Input is empty

  J: i = 2 AND tape[i-1] = BLANK

- Determine if in $a^*b^*c^*$

  A: tape[2..i-1] = a^+ $\wedge$ i > 2
  B: tape[2..i-1] = a^+b^+ $\wedge$ i > 3
  C: tape[2..i-1] = $a^+b^+c^+$ $\wedge$ i > 4

- Equal number of a, b, and c substituted

  D: w = $x^n a^+ x^n b^+ x^n c^+$ $\wedge$ i $\geq$ 1

- Substituting non-last a, b, and c

  E: i > 1 $\wedge$ w = $x^n a^+ x^n b^+ x^n c^+$
  F: i > 1 $\wedge$ w = $x^{n+1} a^+ x^n bb^+ x^n cc^+$
  G: i > 3 $\wedge$ w = $x^{n+1} a^+ x^{n+1} b^+ x^n cc^+$

- Substituting last a, b, and c

  H: i > 1 $\wedge$ w=x^+bx^+c $\wedge$ |xs| remainder 3 = 1 $\wedge$ $|x^*b|=2|x^*c|$
  I: i > 2 $\wedge$ w=x^+c $\wedge$ |xs| remainder 3 = 2

- Skip last c substituted and check if blank read

  K: i > 3 $\wedge$ w = xxxx^* $\wedge$ |xs| remainder 3 = 0 $\wedge$ tape[i]=x
  $\wedge$ i=|w|+1
  L: i > 4 $\wedge$ w = xxxx^* $\wedge$ |xs| remainder 3 = 0 $\wedge$ i=|w|+2

- Accepting

  Y: w = x^* $\wedge$ |xs| remainder 3 = 0, final accepting state

Turing Machines

- Transition relation

```
((S ,BLANK) (J ,RIGHT))
```

Turing Machines

- Transition relation

```
((S ,BLANK) (J ,RIGHT))
```

- ```
  ((J ,BLANK) (Y ,BLANK))     ((J ,BLANK) (A ,RIGHT))
  ```
  ```

Turing Machines

- Transition relation

```
((S ,BLANK) (J ,RIGHT))
```

- ((J ,BLANK) (Y ,BLANK)) ((J ,BLANK) (A ,RIGHT))
- ((A a) (A ,RIGHT)) ((A b) (B ,RIGHT))
  ((B b) (B ,RIGHT)) ((B c) (C ,RIGHT))
  ((C c) (C ,RIGHT)) ((C ,BLANK) (D ,LEFT))

Turing Machines

- Transition relation

```
((S ,BLANK) (J ,RIGHT))
```

- ```
  ((J ,BLANK) (Y ,BLANK))     ((J ,BLANK) (A ,RIGHT))
  ```
- ```
  ((A a) (A ,RIGHT))     ((A b) (B ,RIGHT))
  ((B b) (B ,RIGHT))     ((B c) (C ,RIGHT))
  ((C c) (C ,RIGHT))     ((C ,BLANK) (D ,LEFT))
  ```
- Move left to start a new substitution cycle

```
((D a) (D ,LEFT))
((D b) (D ,LEFT))
((D c) (D ,LEFT))
((D x) (D ,LEFT))
((D ,BLANK) (E ,RIGHT))
```

Turing Machines

- Transition relation

```
((S ,BLANK) (J ,RIGHT))
```

- ((J ,BLANK) (Y ,BLANK)) ((J ,BLANK) (A ,RIGHT))
- ((A a) (A ,RIGHT)) ((A b) (B ,RIGHT))
  ((B b) (B ,RIGHT)) ((B c) (C ,RIGHT))
  ((C c) (C ,RIGHT)) ((C ,BLANK) (D ,LEFT))
- Move left to start a new substitution cycle

```
((D a) (D ,LEFT))
((D b) (D ,LEFT))
((D c) (D ,LEFT))
((D x) (D ,LEFT))
((D ,BLANK) (E ,RIGHT))
```

- Substitution cycle

```
((E x) (E ,RIGHT))    ((E a) (F x))    ((E a) (H x)) Last or not last cycle
```

Turing Machines

- Transition relation

```
((S ,BLANK) (J ,RIGHT))
```

- ((J ,BLANK) (Y ,BLANK)) ((J ,BLANK) (A ,RIGHT))
- ((A a) (A ,RIGHT)) ((A b) (B ,RIGHT))
  ((B b) (B ,RIGHT)) ((B c) (C ,RIGHT))
  ((C c) (C ,RIGHT)) ((C ,BLANK) (D ,LEFT))
- Move left to start a new substitution cycle

```
((D a) (D ,LEFT))
((D b) (D ,LEFT))
((D c) (D ,LEFT))
((D x) (D ,LEFT))
((D ,BLANK) (E ,RIGHT))
```

- Substitution cycle

```
((E x) (E ,RIGHT))    ((E a) (F x))    ((E a) (H x)) Last or not last cycle
```

- not last substitution

```
((F a) (F ,RIGHT))   ((F b) (G x))     ((F x) (F ,RIGHT))
((G b) (G ,RIGHT))   ((G x) (G ,RIGHT)) ((G c) (D x))
```

Turing Machines

- Transition relation

```
((S ,BLANK) (J ,RIGHT))
```

- ((J ,BLANK) (Y ,BLANK)) ((J ,BLANK) (A ,RIGHT))
- ((A a) (A ,RIGHT)) ((A b) (B ,RIGHT))
  ((B b) (B ,RIGHT)) ((B c) (C ,RIGHT))
  ((C c) (C ,RIGHT)) ((C ,BLANK) (D ,LEFT))
- Move left to start a new substitution cycle

```
((D a) (D ,LEFT))
((D b) (D ,LEFT))
((D c) (D ,LEFT))
((D x) (D ,LEFT))
((D ,BLANK) (E ,RIGHT))
```

- Substitution cycle

```
((E x) (E ,RIGHT))     ((E a) (F x))     ((E a) (H x)) Last or not last cycle
```

- not last substitution

```
((F a) (F ,RIGHT))    ((F b) (G x))      ((F x) (F ,RIGHT))
((G b) (G ,RIGHT))    ((G x) (G ,RIGHT)) ((G c) (D x))
```

- last substitution

```
((H x) (H ,RIGHT))     ((H b) (I x))
((I x) (I ,RIGHT))     ((I c) (K x))
((K x) (L ,RIGHT))
((L ,BLANK) (Y ,BLANK)))
```

Turing Machines

- Invariant predicates defined in the textbook

Turing Machines

- Invariant predicates defined in the textbook
- Full proof in the textbook

Theorem

The state invariants hold when M is applied to w.

The proof is by induction on, `n`, the number of transitions performed by `M`.

Proof.

- Base case: $n = 0$
  When M starts it is in state S and, by precondition, M's head position is 1 and position 1 is the blank before w. Therefore, S-INV holds.

□

Turing Machines

- Invariant predicates defined in the textbook
- Full proof in the textbook

Theorem

The state invariants hold when M is applied to w.

The proof is by induction on, `n`, the number of transitions performed by `M`.

Proof.

- Base case: $n = 0$
  When M starts it is in state S and, by precondition, M's head position is 1 and position 1 is the blank before w. Therefore, S-INV holds.
- Inductive Step:
  Assume: State invariants hold for a computation of length $n = k$
  Show: State invariants hold for a computation of length $n = k + 1$

□

Turing Machines

- Invariant predicates defined in the textbook
- Full proof in the textbook

Theorem

The state invariants hold when M is applied to w.

The proof is by induction on, `n`, the number of transitions performed by `M`.

Proof.

- Base case: $n = 0$
  When M starts it is in state S and, by precondition, M's head position is 1 and position 1 is the blank before w. Therefore, S-INV holds.
- Inductive Step:
  Assume: State invariants hold for a computation of length $n = k$
  Show: State invariants hold for a computation of length $n = k + 1$
- `((S ,BLANK) (J ,RIGHT))`: By inductive hypothesis, S-INV holds. This means M's head position is 1 and a blank is read. Using this transition means the head's position becomes 2 and the tape at position 1 has a blank. Thus, J-INV holds.

□

Turing Machines

- Invariant predicates defined in the textbook
- Full proof in the textbook

Theorem

The state invariants hold when M is applied to w.

The proof is by induction on, `n`, the number of transitions performed by `M`.

Proof.

- Base case: $n = 0$
  When M starts it is in state S and, by precondition, M's head position is 1 and position 1 is the blank before w. Therefore, S-INV holds.
- Inductive Step:
  Assume: State invariants hold for a computation of length $n = k$
  Show: State invariants hold for a computation of length $n = k + 1$
- `((S ,BLANK) (J ,RIGHT))`: By inductive hypothesis, S-INV holds. This means M's head position is 1 and a blank is read. Using this transition means the head's position becomes 2 and the tape at position 1 has a blank. Thus, J-INV holds.
- `((J ,BLANK) (Y ,BLANK))`: By inductive hypothesis, J-INV holds. This means the head's position is 2 and the tape at position 1 has a blank. Using this rule means that w is empty and the number of `x`s (i.e., 0) is divisible by 3. Furthermore, the tape is reading a blank. Thus, Y-INV holds.

□

Turing Machines

- Invariant predicates defined in the textbook
- Full proof in the textbook

Theorem

The state invariants hold when M is applied to w.

The proof is by induction on, `n`, the number of transitions performed by `M`.

Proof.

- Base case: $n = 0$
  When M starts it is in state S and, by precondition, M's head position is 1 and position 1 is the blank before w. Therefore, S-INV holds.
- Inductive Step:
  Assume: State invariants hold for a computation of length $n = k$
  Show: State invariants hold for a computation of length $n = k + 1$
- `((S ,BLANK) (J ,RIGHT))`: By inductive hypothesis, S-INV holds. This means M's head position is 1 and a blank is read. Using this transition means the head's position becomes 2 and the tape at position 1 has a blank. Thus, J-INV holds.
- `((J ,BLANK) (Y ,BLANK))`: By inductive hypothesis, J-INV holds. This means the head's position is 2 and the tape at position 1 has a blank. Using this rule means that w is empty and the number of `x`s (i.e., 0) is divisible by 3. Furthermore, the tape is reading a blank. Thus, Y-INV holds.
- `((J a) (A ,RIGHT))`: By inductive hypothesis, J-INV holds. This means the head's position is 2 and the tape at position 1 has a blank. Using this rule means that the head is moved to the right to position 3 and that a single a and nothing else has been read from w. Thus, A-INV holds.

  ⋮

□

Turing Machines

Lemma

$w \in L \Leftrightarrow w \in L(M)$

Proof.

($\Rightarrow$) Assume w∈L. This means that $w = a^n b^n c^n$. Given that state invariants always hold, there is a computation that has M repeatedly mutating matching as, bs, and cs to x and ending in Y with the input word only having a multiple of 3 number of xs. Therefore, w∈L(M).
($\Leftarrow$) Assume w∈L(M). This means that M halts in Y, the only accepting state, with the input word only having a multiple of 3 number xs. Given that the state invariants always hold, the only way for M to reach Y is by the original input being in $a^n b^n c^n$ and substituting equal number of as, bs, and cs with xs. Thus, w∈L. □ □

•

Turing Machines

Lemma

$w \in L \Leftrightarrow w \in L(M)$

Proof.

($\Rightarrow$) Assume w∈L. This means that w = $a^n b^n c^n$. Given that state invariants always hold, there is a computation that has M repeatedly mutating matching as, bs, and cs to x and ending in Y with the input word only having a multiple of 3 number of xs. Therefore, w∈L(M).
($\Leftarrow$) Assume w∈L(M). This means that M halts in Y, the only accepting state, with the input word only having a multiple of 3 number xs. Given that the state invariants always hold, the only way for M to reach Y is by the original input being in $a^n b^n c^n$ and substituting equal number of as, bs, and cs with xs. Thus, w∈L. □ □

•

Lemma

$w \notin L \Leftrightarrow w \notin L(M)$

Proof.

($\Rightarrow$) Assume w∉L. This means that w $\neq a^n b^n c^n$. Given that state invariants always hold, M cannot halt in Y. Therefore, w∉L(M).
($\Leftarrow$) Assume w∉L(M). This means that M does not halts in Y. Given that the state invariants always hold, this means that w is not in $a^n b^n c^n$. Thus, w ∉L. □ □

•

Turing Machines

- The first recipient of the prestigious Turing Award, Alan Perlis, wrote:

  Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy.

Turing Machines

- The first recipient of the prestigious Turing Award, Alan Perlis, wrote:

  Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy.

- Turing machines do not offer any of the common programming constructs that we are accustomed to as computer scientists

Turing Machines

- The first recipient of the prestigious Turing Award, Alan Perlis, wrote:

  > *Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy.*

- Turing machines do not offer any of the common programming constructs that we are accustomed to as computer scientists
- So, why bother studying Turing machines?

Turing Machines

- The first recipient of the prestigious Turing Award, Alan Perlis, wrote:

  > *Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy.*

- Turing machines do not offer any of the common programming constructs that we are accustomed to as computer scientists
- So, why bother studying Turing machines?
- Sharpen our skills to program using an `API`
- A simple model that allows us to easily reason about it and provides a lingua franca for all computer scientists to talk about computation and its limitations

Turing Machines

- HOMEWORK: 10, 12–15

Turing Machine Composition

- Turing machines can also perform computations and execute statements
- To help tame the Turing tar-pit, Turing machines may be composed
- Think of Turing machines as auxiliary functions

Turing Machine Composition

- Turing machines can also perform computations and execute statements
- To help tame the Turing tar-pit, Turing machines may be composed
- Think of Turing machines as auxiliary functions
- Move right machine

```
;;  PRE: tape = (LM w) AND i=k>0, where w in {a b BLANK}*
;; POST: tape = (LM w) AND i=k+1
(define R (make-tm '(S F)
                   '(a b)
                   `(((S a) (F ,RIGHT))
                     ((S b) (F ,RIGHT))
                     ((S ,BLANK) (F ,RIGHT)))
                   'S
                   '(F)))
```

Turing Machine Composition

- Turing machines can also perform computations and execute statements
- To help tame the Turing tar-pit, Turing machines may be composed
- Think of Turing machines as auxiliary functions
- Move right machine

```
;;  PRE: tape = (LM w) AND i=k>0, where w in {a b BLANK}*
;; POST: tape = (LM w) AND i=k+1
(define R (make-tm '(S F)
                   '(a b)
                   `(((S a) (F ,RIGHT))
                     ((S b) (F ,RIGHT))
                     ((S ,BLANK) (F ,RIGHT)))
                   'S
                   '(F)))
```

- Move left machine

```
;;  PRE: tape = (LM w) AND i=k≥1, where w in {a b BLANK}*
;; POST: tape = (LM w) AND i=k-1
(define L (make-tm '(S H)
                   '(a b)
                   `(((S a) (H ,LEFT))
                     ((S b) (H ,LEFT))
                     ((S ,BLANK) (H ,LEFT)))
                   'S
                   '(H)))
```

Turing Machine Composition

- Halt machine

```
;;  PRE: tape = (LM w), where w in {a b BLANK}*
;; POST: tape = (LM w)
(define HALT (make-tm '(S)
                      '(a b)
                      '()
                      'S
                      '(S)))
```

Turing Machine Composition

- Halt machine

```
;;  PRE: tape = (LM w), where w in {a b BLANK}*
;; POST: tape = (LM w)
(define HALT (make-tm '(S)
                      '(a b)
                      '()
                      'S
                      '(S)))
```

- Write blank machine

```
;;  PRE: tape = (LM w) AND i=k>0 AND tape[i]=s, where w in {a b
;; POST: tape = (LM w) AND i=k AND tape[i]=BLANK
(define WB (make-tm '(S H)
                    `(a b)
                    `(((S a) (H ,BLANK))
                      ((S b) (H ,BLANK))
                      ((S ,BLANK) (H ,BLANK)))
                    'S
                    '(H)))
```

Turing Machine Composition

- Move right twice

```
;;  PRE: tape = (LM w) AND i=k>0 AND w in (a b BLANK)*
;; POST: tape = (LM w) AND i=k+2 AND w in (a b BLANK)*
(define R^2 (make-tm '(S A F)
                     '(a b)
                     `(((S a) (A ,RIGHT))
                       ((S b) (A ,RIGHT))
                       ((S ,BLANK) (A ,RIGHT))
                       ((A a) (F ,RIGHT))
                       ((A b) (F ,RIGHT))
                       ((A ,BLANK) (F ,RIGHT)))
                     'S
                     '(F)))
```

Turing Machine Composition

- Move right twice

```
;;  PRE: tape = (LM w) AND i=k>0 AND w in (a b BLANK)*
;; POST: tape = (LM w) AND i=k+2 AND w in (a b BLANK)*
(define R^2 (make-tm '(S A F)
                     '(a b)
                     `(((S a) (A ,RIGHT))
                       ((S b) (A ,RIGHT))
                       ((S ,BLANK) (A ,RIGHT))
                       ((A a) (F ,RIGHT))
                       ((A b) (F ,RIGHT))
                       ((A ,BLANK) (F ,RIGHT)))
                     'S
                     '(F)))
```

- It is the composition of R with itself
- Screaming for abstraction!

Turing Machine Composition

- HOMEWORK: 1–3

Turing Machine Composition

- Despite their simplicity, Turing machines are powerful enough to be programmed
- *The universal Turing machine* (`UTM`) is a Turing machine that can simulate an arbitrary Turing machine on arbitrary input
- It is given as input (on its tape) a description of the machine to simulate as well as a description of the simulated input tape for the simulated machine

Turing Machine Composition

- Despite their simplicity, Turing machines are powerful enough to be programmed
- *The universal Turing machine* (`UTM`) is a Turing machine that can simulate an arbitrary Turing machine on arbitrary input
- It is given as input (on its tape) a description of the machine to simulate as well as a description of the simulated input tape for the simulated machine
- Must track the next action to perform and the position of the head in the simulated input
- Think of the next action to perform as a program counter
- The position of the simulated head indicates the next element to be read from the simulated input tape

Turing Machine Composition

- Moving right n times

```
`(,LM ,BLANK i r r r ,BLANK h a b a)
```

- Program, Tape, Program counter, and Head position

Turing Machine Composition

- Moving right n times

```
`(,LM ,BLANK i r r r ,BLANK h a b a)
```

- Program, Tape, Program counter, and Head position
- How does the Turing machine evaluating our program work?

Turing Machine Composition

- Moving right n times

```
`(,LM ,BLANK i r r r ,BLANK h a b a)
```

- Program, Tape, Program counter, and Head position
- How does the Turing machine evaluating our program work?
- Much like the Von Neumann architecture

Turing Machine Composition

- Moving right n times

```
`(,LM ,BLANK i r r r ,BLANK h a b a)
```

- Program, Tape, Program counter, and Head position
- How does the Turing machine evaluating our program work?
- Much like the Von Neumann architecture
- Fetches the next action to perform, moves the program counter (i.e., `i`), performs the action, and loops
- The loop stops when there are no more instructions to perform (e.g., `i` is at the position containing the blank between the simulated machine's description and the simulated input tape)

Turing Machine Composition

```
;;  PRE: tape = (LM BLANK i p BLANK w h v) AND
;;       head on first blank in position 1
;; POST: tape = (LM BLANK p i BLANK o) OR
;;              (LM BLANK p i BLANK u)
;;       AND head on second blank,
;; where p = r^n for n∈ℕ,
;;       o = w′hv′ where |w′|-|w| = n,
;;       u = w′v′BLANK*h where |w′v′BLANK*|-|w| = n,
;;       w,w′,v,v′∈Σ*, wv = w′v′
```

Turing Machine Composition

- ```
  ;;  PRE: tape = (LM BLANK i p BLANK w h v) AND
  ;;       head on first blank in position 1
  ;; POST: tape = (LM BLANK p i BLANK o) OR
  ;;              (LM BLANK p i BLANK u)
  ;;       AND head on second blank,
  ;; where p = r^n for n∈ℕ,
  ;;       o = w′hv′ where |w′|-|w| = n,
  ;;       u = w′v′BLANK*h where |w′v′BLANK*|-|w| = n,
  ;;       w,w′,v,v′∈Σ*, wv = w′v′
  ```
- The machine operates as follows:
  1. Starts with the head on the blank in position 1 of the tape
  2. Attempts to find the `i` to the right (i.e., the next instruction to execute)
  3. If done executing the program the `tm` running the program moves to halt. Otherwise, it moves to the next step.
  4. Move `i` to the right
  5. Find `h`
  6. Execute current instruction
  7. Find `i`
  8. Loop to step 3
  ```

Turing Machine Composition

- Start by skipping the program counter

```
((S ,BLANK) (A ,RIGHT))
((A i) (Q ,RIGHT))
```

Turing Machine Composition

- Start by skipping the program counter

```
((S ,BLANK) (A ,RIGHT))
((A i) (Q ,RIGHT))
```

- If done move to halt. Otherwise, move pc right

```
((Q ,BLANK) (Y ,BLANK))
((Q r) (M i))
((M i) (N ,LEFT))
((N i) (B r))
```

Turing Machine Composition

- Start by skipping the program counter

```
((S ,BLANK) (A ,RIGHT))
((A i) (Q ,RIGHT))
```

- If done move to halt. Otherwise, move pc right

```
((Q ,BLANK) (Y ,BLANK))
((Q r) (M i))
((M i) (N ,LEFT))
((N i) (B r))
```

- Find head and move UTM head to the right

```
((B r) (B ,RIGHT))
((B i) (B ,RIGHT))
((B ,BLANK) (B ,RIGHT))
((B a) (B ,RIGHT))
((B b) (B ,RIGHT))
((B h) (D ,RIGHT))
```

Turing Machine Composition

- Start by skipping the program counter

```
((S ,BLANK) (A ,RIGHT))
((A i) (Q ,RIGHT))
```

- If done move to halt. Otherwise, move pc right

```
((Q ,BLANK) (Y ,BLANK))
((Q r) (M i))
((M i) (N ,LEFT))
((N i) (B r))
```

- Find head and move UTM head to the right

```
((B r) (B ,RIGHT))
((B i) (B ,RIGHT))
((B ,BLANK) (B ,RIGHT))
((B a) (B ,RIGHT))
((B b) (B ,RIGHT))
((B h) (D ,RIGHT))
```

- Move head right

```
((D a) (E h))
((D b) (G h))
((D ,BLANK) (I h))
((E h) (F ,LEFT))
((F h) (K a))
((G h) (H ,LEFT))
((H h) (K b))
((I h) (J ,LEFT))
((J h) (K ,BLANK))
```

Turing Machine Composition

- Start by skipping the program counter

```
((S ,BLANK) (A ,RIGHT))
((A i) (Q ,RIGHT))
```

- If done move to halt. Otherwise, move pc right

```
((Q ,BLANK) (Y ,BLANK))
((Q r) (M i))
((M i) (N ,LEFT))
((N i) (B r))
```

- Find head and move UTM head to the right

```
((B r) (B ,RIGHT))
((B i) (B ,RIGHT))
((B ,BLANK) (B ,RIGHT))
((B a) (B ,RIGHT))
((B b) (B ,RIGHT))
((B h) (D ,RIGHT))
```

- Move head right

```
((D a) (E h))
((D b) (G h))
((D ,BLANK) (I h))
((E h) (F ,LEFT))
((F h) (K a))
((G h) (H ,LEFT))
((H h) (K b))
((I h) (J ,LEFT))
((J h) (K ,BLANK))
```

- Find pc and loop

```
((K a) (K ,LEFT))
((K b) (K ,LEFT))
((K ,BLANK) (K ,LEFT))
((K r) (K ,LEFT))
((K i) (Q ,RIGHT))
```

Turing Machine Composition

- We no longer need to implement `R` multiple times
- Provide the proper `tm` and input tape descriptions to move the simulated head to the right a specified number of times

Turing Machine Composition

- We no longer need to implement `R` multiple times
- Provide the proper `tm` and input tape descriptions to move the simulated head to the right a specified number of times
- Still rather difficult to program even the simplest tasks using a `tm`. This is said even though we have not looked at problems
- We need a programming language that is conducive to making it easier to express the solution to a problem
- A programming language that makes it easier to compose Turing machines

Turing Machine Composition

- Composed Turing machines may be visualized as a graphic that resembles a flow chart

$$\begin{tikzcd} \underline{M_1} & M_2 \arrow[r, "a"] \arrow[d, "b"'] & M_3 \\ & M_4 & \end{tikzcd}$$

- $\mathtt{M}_{1\text{--}4}$ are descriptions of Turing machines

Turing Machine Composition

- Composed Turing machines may be visualized as a graphic that resembles a flow chart

$$\begin{tikzcd} \underline{M_1} & M_2 \arrow[r, "a"] \arrow[d, "b"'] & M_3 \\ & M_4 & \end{tikzcd}$$

- $\mathtt{M}_{1\text{--}4}$ are descriptions of Turing machines
- Starts by executing $\mathtt{M}_1$
- When $\mathtt{M}_1$ halts $\mathtt{M}_2$ is executed starting in the machine's configuration after $\mathtt{M}_1$ halts

Turing Machine Composition

- Composed Turing machines may be visualized as a graphic that resembles a flow chart

$$\begin{tikzcd} \underline{M_1}\ M_2 \arrow[r, "a"] \arrow[d, "b"'] & M_3 \\ M_4 & \end{tikzcd}$$

- $\mathtt{M}_{1\text{--}4}$ are descriptions of Turing machines
- Starts by executing $\mathtt{M}_1$
- When $\mathtt{M}_1$ halts $\mathtt{M}_2$ is executed starting in the machine's configuration after $\mathtt{M}_1$ halts
- When $\mathtt{M}_2$ halts a decision is made
- If an a is read then $\mathtt{M}_3$ is executed and the machine halts given that there is nothing more to execute
- If a b is read then $\mathtt{M}_4$ is executed and the machine halts for the same reason
- Observe that in this example a conditional jump is required to either execute $\mathtt{M}_3$ or $\mathtt{M}_4$

Turing Machine Composition

- A `ctmd` is a list defined as follows:
  1. empty list
  2. (cons `m` `ctmd`), where `m` is either a `tm` or a `ctmd`
  3. (cons LABEL `ctmd`)
  4. (cons (list GOTO LABEL) `ctmd`)
  5. (cons (BRANCH (listof (list symbol (list GOTO LABEL)))) `ctmd`)
  6. (cons ((VAR symbol) `ctmd`) `ctmd`)
  7. (cons variable `ctmd`)

Turing Machine Composition

- A `ctmd` is a list defined as follows:
  1. empty list
  2. (cons `m` `ctmd`), where `m` is either a `tm` or a `ctmd`
  3. (cons LABEL `ctmd`)
  4. (cons (list GOTO LABEL) `ctmd`)
  5. (cons (BRANCH (listof (list symbol (list GOTO LABEL)))) `ctmd`)
  6. (cons ((VAR symbol) `ctmd`) `ctmd`)
  7. (cons variable `ctmd`)
- R R
- ```
  ;; PRE:  tape = (LM w) AND i=k>0 AND w in (a b)*
  ;; POST: tape = (LM w) AND i=k+2 AND w in (a b)*
  (define RR (combine-tms (list R R) '(a b)))

  (check-equal? (ctm-run RR `(,LM b a a) 1)
                `(F 3 (,LM b a a)))
  (check-equal? (ctm-run RR `(,LM a a a) 2)
                `(F 4 (,LM a a a ,BLANK)))
  (check-equal? (ctm-run RR `(,LM a b b a) 3)
                `(F 5 (,LM a b b a ,BLANK)))
  (check-equal? (ctm-run RR `(,LM b) 1)
                `(F 3 (,LM b ,BLANK ,BLANK)))
  ```
  ```

Turing Machine Composition

-

Turing Machine Composition

- not blank
  R
- ;; PRE: tape = (LM w) AND i=k>0 AND w in (a b BLANK)*
  ;; POST: tape = (LM w) AND i>k AND tape[i] = BLANK
  ;; AND tape[k+1..i-1] ≠ BLANK

```
(define FBR (combine-tms
              (list 0
                    R
                    (cons BRANCH
                          (list (list 'a (list GOTO 0))
                                (list 'b (list GOTO 0))
                                (list BLANK (list GOTO 10))))
                   10)
             (list 'a 'b)))

(check-equal? (ctm-run FBR `(,LM ,BLANK) 1)
              `(F 2 (,LM ,BLANK ,BLANK)))
(check-equal? (ctm-run FBR `(,LM a a b b b ,BLANK a a) 2)
              `(F 6 (,LM a a b b b ,BLANK a a)))
(check-equal? (ctm-run FBR `(,LM ,BLANK ,BLANK b b) 3)
              `(F 5 (,LM ,BLANK ,BLANK b b ,BLANK)))
```

Turing Machine Composition

- not blank
  L

Turing Machine Composition

-


- ```
  ;; PRE:  tape = (LM w) AND i=k>0 AND w in (a b BLANK)*
  ;; POST: tape = (LM w) AND i<k AND tape[i]=BLANK
  ;;       AND tape[i+1..|w|] != BLANK
  (define FBL (combine-tms
                (list 0
                       L
                       (cons BRANCH
                             (list (list 'a (list GOTO 0))
                                   (list 'b (list GOTO 0))
                                   (list BLANK (list GOTO 1))
                                   (list LM (list GOTO 0))))
                      1)
                (list 'a 'b)))

  (check-equal? (ctm-run FBL `(,LM ,BLANK a a b) 4)
                `(H 1 (,LM ,BLANK a a b)))
  (check-equal?
    (ctm-run FBL `(,LM a ,BLANK a b ,BLANK b a b b) 8)
    `(H 5 (,LM a ,BLANK a b ,BLANK b a b b)))
  ```

- The BRANCH statement in this case has a branch for processing LM, because LM may be read as it moves left
  ```

Turing Machine Composition

- HOMEWORK: 6, 7, 9, 10

Turing Machine Composition

- Consider the problem of adding two natural numbers
- Must define how to represent natural numbers
- Must define how to add two natural numbers represented using the chosen notation

Turing Machine Composition

- Consider the problem of adding two natural numbers
- Must define how to represent natural numbers
- Must define how to add two natural numbers represented using the chosen notation
- Representation

```
A natural number in unary notation (nn) is either:
  1. (BLANK)
  2. (d nn)
```

Turing Machine Composition

- Consider the problem of adding two natural numbers
- Must define how to represent natural numbers
- Must define how to add two natural numbers represented using the chosen notation
- Representation

```
A natural number in unary notation (nn) is either:
  1. (BLANK)
  2. (d nn)
```

- Given two `nn` separated on a tape by a blank:
  1. Mutate the blank between the two given natural numbers to d
  2. Mutate the last i in the second argument, if any, to a blank

Turing Machine Composition

- Consider the problem of adding two natural numbers
- Must define how to represent natural numbers
- Must define how to add two natural numbers represented using the chosen notation
- Representation

```
A natural number in unary notation (nn) is either:
  1. (BLANK)
  2. (d nn)
```

- Given two nn separated on a tape by a blank:
  1. Mutate the blank between the two given natural numbers to d
  2. Mutate the last i in the second argument, if any, to a blank
- Specification

```
;;  PRE: tape = (LM BLANK a BLANK b) AND i = 1
;; POST: tape = (LM BLANK a b) AND
;;       i = 3                          if a = b = 0
;;         = numd(a) + numd(b) + 2      otherwise,
;;       where numd(x) = the number of ds in x
```

Turing Machine Composition

- Algorithm phases:
  1. Skip a and write d in blank after a
  2. Skip b and move head left
  3. Mutate the position under the head to a blank
  4. Move the head left and decide if the result is empty:
     1. If a d is read move right and halt
     2. Otherwise, move to the head right twice to the blank after the blank representing 0 (the result) and halt
- These steps suggest 8 states are needed: the starting state, the final state, and 6 additional states for the steps above

Turing Machine Composition

- Algorithm phases:
  1. Skip a and write d in blank after a
  2. Skip b and move head left
  3. Mutate the position under the head to a blank
  4. Move the head left and decide if the result is empty:
     1. If a d is read move right and halt
     2. Otherwise, move to the head right twice to the blank after the blank representing 0 (the result) and halt
- These steps suggest 8 states are needed: the starting state, the final state, and 6 additional states for the steps above
- Starting state

```
S: i = 1 and tape = (LM BLANK a BLANK b)
```

Turing Machine Composition

- Algorithm phases:
  1. Skip a and write d in blank after a
  2. Skip b and move head left
  3. Mutate the position under the head to a blank
  4. Move the head left and decide if the result is empty:
     1. If a d is read move right and halt
     2. Otherwise, move to the head right twice to the blank after the blank representing 0 (the result) and halt
- These steps suggest 8 states are needed: the starting state, the final state, and 6 additional states for the steps above
- Starting state

```
S: i = 1 and tape = (LM BLANK a BLANK b)
```

- Skip a

```
A: i <= numd(a) + 2 and tape = (LM BLANK a BLANK b)
```

Turing Machine Composition

- Algorithm phases:
  1. Skip a and write d in blank after a
  2. Skip b and move head left
  3. Mutate the position under the head to a blank
  4. Move the head left and decide if the result is empty:
     1. If a d is read move right and halt
     2. Otherwise, move to the head right twice to the blank after the blank representing 0 (the result) and halt
- These steps suggest 8 states are needed: the starting state, the final state, and 6 additional states for the steps above
- Starting state

```
S: i = 1 and tape = (LM BLANK a BLANK b)
```

- Skip a

```
A: i <= numd(a) + 2 and tape = (LM BLANK a BLANK b)
```

- Mutate middle blank and skip b

```
B: i <= numd(a) + numd(b) + 3 and tape = (LM BLANK a d b)
```

Turing Machine Composition

- Algorithm phases:
  1. Skip a and write d in blank after a
  2. Skip b and move head left
  3. Mutate the position under the head to a blank
  4. Move the head left and decide if the result is empty:
     1. If a d is read move right and halt
     2. Otherwise, move to the head right twice to the blank after the blank representing 0 (the result) and halt
- These steps suggest 8 states are needed: the starting state, the final state, and 6 additional states for the steps above
- Starting state

  ```
  S: i = 1 and tape = (LM BLANK a BLANK b)
  ```
- Skip a

  ```
  A: i <= numd(a) + 2 and tape = (LM BLANK a BLANK b)
  ```
- Mutate middle blank and skip b

  ```
  B: i <= numd(a) + numd(b) + 3 and tape = (LM BLANK a d b)
  ```
- Move left to mutate last d

  ```
  C: i = numd(a) + numd(b) + 2 and tape = (LM BLANK a d b)
  ```

Turing Machine Composition

- Algorithm phases:
  1. Skip a and write d in blank after a
  2. Skip b and move head left
  3. Mutate the position under the head to a blank
  4. Move the head left and decide if the result is empty:
     1. If a d is read move right and halt
     2. Otherwise, move to the head right twice to the blank after the blank representing 0 (the result) and halt
- These steps suggest 8 states are needed: the starting state, the final state, and 6 additional states for the steps above
- Starting state

  ```
  S: i = 1 and tape = (LM BLANK a BLANK b)
  ```
- Skip a

  ```
  A: i <= numd(a) + 2 and tape = (LM BLANK a BLANK b)
  ```
- Mutate middle blank and skip b

  ```
  B: i <= numd(a) + numd(b) + 3 and tape = (LM BLANK a d b)
  ```
- Move left to mutate last d

  ```
  C: i = numd(a) + numd(b) + 2 and tape = (LM BLANK a d b)
  ```
- After mutating the last d

  ```
  D: i = numd(a) + numd(b) + 2 and tape = (LM BLANK a b)
  ```

Turing Machine Composition

- Algorithm phases:
  1. Skip a and write d in blank after a
  2. Skip b and move head left
  3. Mutate the position under the head to a blank
  4. Move the head left and decide if the result is empty:
     1. If a d is read move right and halt
     2. Otherwise, move to the head right twice to the blank after the blank representing 0 (the result) and halt
- These steps suggest 8 states are needed: the starting state, the final state, and 6 additional states for the steps above
- Starting state

  `S: i = 1 and tape = (LM BLANK a BLANK b)`
- Skip a

  `A: i <= numd(a) + 2 and tape = (LM BLANK a BLANK b)`
- Mutate middle blank and skip b

  `B: i <= numd(a) + numd(b) + 3 and tape = (LM BLANK a d b)`
- Move left to mutate last d

  `C: i = numd(a) + numd(b) + 2 and tape = (LM BLANK a d b)`
- After mutating the last d

  `D: i = numd(a) + numd(b) + 2 and tape = (LM BLANK a b)`
- Move left to determine if sum is 0

  `E: i = numd(a) + numd(b) + 1 and tape = (LM BLANK a b)`

Turing Machine Composition

- Algorithm phases:
  1. Skip a and write d in blank after a
  2. Skip b and move head left
  3. Mutate the position under the head to a blank
  4. Move the head left and decide if the result is empty:
     1. If a d is read move right and halt
     2. Otherwise, move to the head right twice to the blank after the blank representing 0 (the result) and halt
- These steps suggest 8 states are needed: the starting state, the final state, and 6 additional states for the steps above
- Starting state

```
S: i = 1 and tape = (LM BLANK a BLANK b)
```

- Skip a

```
A: i <= numd(a) + 2 and tape = (LM BLANK a BLANK b)
```

- Mutate middle blank and skip b

```
B: i <= numd(a) + numd(b) + 3 and tape = (LM BLANK a d b)
```

- Move left to mutate last d

```
C: i = numd(a) + numd(b) + 2 and tape = (LM BLANK a d b)
```

- After mutating the last d

```
D: i = numd(a) + numd(b) + 2 and tape = (LM BLANK a b)
```

- Move left to determine if sum is 0

```
E: i = numd(a) + numd(b) + 1 and tape = (LM BLANK a b)
```

- Place head to meet postcondition

```
G: i = numd(a) + numd(b) + 2 and tape = (LM BLANK)
F: i = 3                        if a = b = 0
     = numd(a) + numd(b) + 2    otherwise
   and tape = (LM BLANK a b)
```

Turing Machine Composition

```
;;  PRE: tape = (LM BLANK a BLANK b) AND i = 1
;; POST: tape = (LM BLANK a b) AND
;;       i = 3                      if a = b = 0
;;         = numd(a) + numd(b) + 2 otherwise,
;;       where numd(x) = the number of ds in x
(define ADD (make-tm '(S A B C D E F G)
                     '(d)
                     'S
                     '(F)))
```

Turing Machine Composition

```
;;  PRE: tape = (LM BLANK a BLANK b) AND i = 1
;; POST: tape = (LM BLANK a b) AND
;;       i = 3                        if a = b = 0
;;         = numd(a) + numd(b) + 2 otherwise,
;;       where numd(x) = the number of ds in x
(define ADD (make-tm '(S A B C D E F G)
                     '(d)
                     'S
                     '(F)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK ,BLANK ,BLANK) 1))
 `(F 3 (,LM ,BLANK ,BLANK ,BLANK)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK ,BLANK d d d ,BLANK) 1))
 `(F 5 (,LM ,BLANK d d d ,BLANK ,BLANK)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK d d ,BLANK ,BLANK) 1))
 `(F 4 (,LM ,BLANK d d ,BLANK ,BLANK)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK d d ,BLANK d d d) 1))
 `(F 7 (,LM ,BLANK d d d d d ,BLANK ,BLANK)))
```

Turing Machine Composition

```
;;  PRE: tape = (LM BLANK a BLANK b) AND i = 1
;; POST: tape = (LM BLANK a b) AND
;;       i = 3                      if a = b = 0
;;         = numd(a) + numd(b) + 2 otherwise,
;;       where numd(x) = the number of ds in x
(define ADD (make-tm '(S A B C D E F G)
                     '(d)
                     `(((S ,BLANK) (A ,RIGHT))
                       'S
                       '(F)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK ,BLANK ,BLANK) 1))
 `(F 3 (,LM ,BLANK ,BLANK ,BLANK)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK ,BLANK d d d ,BLANK) 1))
 `(F 5 (,LM ,BLANK d d d ,BLANK ,BLANK)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK d d ,BLANK ,BLANK) 1))
 `(F 4 (,LM ,BLANK d d ,BLANK ,BLANK)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK d d ,BLANK d d d) 1))
 `(F 7 (,LM ,BLANK d d d d d ,BLANK ,BLANK)))
```

Turing Machine Composition

```
;;  PRE: tape = (LM BLANK a BLANK b) AND i = 1
;; POST: tape = (LM BLANK a b) AND
;;       i = 3                        if a = b = 0
;;         = numd(a) + numd(b) + 2 otherwise,
;;       where numd(x) = the number of ds in x
(define ADD (make-tm '(S A B C D E F G)
                     '(d)
                     `(((S ,BLANK) (A ,RIGHT))
                       ((A d) (A ,RIGHT))
                       ((A ,BLANK) (B d))
                     'S
                     '(F)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK ,BLANK ,BLANK) 1))
 `(F 3 (,LM ,BLANK ,BLANK ,BLANK)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK ,BLANK d d d ,BLANK) 1))
 `(F 5 (,LM ,BLANK d d d ,BLANK ,BLANK)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK d d ,BLANK ,BLANK) 1))
 `(F 4 (,LM ,BLANK d d ,BLANK ,BLANK)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK d d ,BLANK d d d) 1))
 `(F 7 (,LM ,BLANK d d d d d ,BLANK ,BLANK)))
```

Turing Machine Composition

```
;;  PRE: tape = (LM BLANK a BLANK b) AND i = 1
;; POST: tape = (LM BLANK a b) AND
;;       i = 3                      if a = b = 0
;;         = numd(a) + numd(b) + 2 otherwise,
;;       where numd(x) = the number of ds in x
(define ADD (make-tm '(S A B C D E F G)
                     '(d)
                     `(((S ,BLANK) (A ,RIGHT))
                       ((A d) (A ,RIGHT))
                       ((A ,BLANK) (B d))
                       ((B d) (B ,RIGHT))
                       ((B ,BLANK) (C ,LEFT))
                     'S
                     '(F)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK ,BLANK ,BLANK) 1))
 `(F 3 (,LM ,BLANK ,BLANK ,BLANK)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK ,BLANK d d d ,BLANK) 1))
 `(F 5 (,LM ,BLANK d d d ,BLANK ,BLANK)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK d d ,BLANK ,BLANK) 1))
 `(F 4 (,LM ,BLANK d d ,BLANK ,BLANK)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK d d ,BLANK d d d) 1))
 `(F 7 (,LM ,BLANK d d d d d ,BLANK ,BLANK)))
```

Turing Machine Composition

```
;;  PRE: tape = (LM BLANK a BLANK b) AND i = 1
;; POST: tape = (LM BLANK a b) AND
;;       i = 3                        if a = b = 0
;;         = numd(a) + numd(b) + 2 otherwise,
;;       where numd(x) = the number of ds in x
(define ADD (make-tm '(S A B C D E F G)
                     '(d)
                     `(((S ,BLANK) (A ,RIGHT))
                       ((A d) (A ,RIGHT))
                       ((A ,BLANK) (B d))
                       ((B d) (B ,RIGHT))
                       ((B ,BLANK) (C ,LEFT))
                       ((C d) (D ,BLANK))
                     'S
                     '(F)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK ,BLANK ,BLANK) 1))
 `(F 3 (,LM ,BLANK ,BLANK ,BLANK)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK ,BLANK d d d ,BLANK) 1))
 `(F 5 (,LM ,BLANK d d d ,BLANK ,BLANK)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK d d ,BLANK ,BLANK) 1))
 `(F 4 (,LM ,BLANK d d ,BLANK ,BLANK)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK d d ,BLANK d d d) 1))
 `(F 7 (,LM ,BLANK d d d d d ,BLANK ,BLANK)))
```

Turing Machine Composition

```
;;  PRE: tape = (LM BLANK a BLANK b) AND i = 1
;; POST: tape = (LM BLANK a b) AND
;;       i = 3                        if a = b = 0
;;         = numd(a) + numd(b) + 2 otherwise,
;;       where numd(x) = the number of ds in x
(define ADD (make-tm '(S A B C D E F G)
                     '(d)
                     `(((S ,BLANK) (A ,RIGHT))
                       ((A d) (A ,RIGHT))
                       ((A ,BLANK) (B d))
                       ((B d) (B ,RIGHT))
                       ((B ,BLANK) (C ,LEFT))
                       ((C d) (D ,BLANK))
                       ((D ,BLANK) (E ,LEFT))
                     'S
                     '(F)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK ,BLANK ,BLANK) 1))
 `(F 3 (,LM ,BLANK ,BLANK ,BLANK)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK ,BLANK d d d ,BLANK) 1))
 `(F 5 (,LM ,BLANK d d d ,BLANK ,BLANK)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK d d ,BLANK ,BLANK) 1))
 `(F 4 (,LM ,BLANK d d ,BLANK ,BLANK)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK d d ,BLANK d d d) 1))
 `(F 7 (,LM ,BLANK d d d d d ,BLANK ,BLANK)))
```

Turing Machine Composition

```
;;  PRE: tape = (LM BLANK a BLANK b) AND i = 1
;; POST: tape = (LM BLANK a b) AND
;;       i = 3                        if a = b = 0
;;         = numd(a) + numd(b) + 2 otherwise,
;;       where numd(x) = the number of ds in x
(define ADD (make-tm '(S A B C D E F G)
                     '(d)
                     `(((S ,BLANK) (A ,RIGHT))
                       ((A d) (A ,RIGHT))
                       ((A ,BLANK) (B d))
                       ((B d) (B ,RIGHT))
                       ((B ,BLANK) (C ,LEFT))
                       ((C d) (D ,BLANK))
                       ((D ,BLANK) (E ,LEFT))
                       ((E d) (F ,RIGHT))
                       ((E ,BLANK) (G ,RIGHT))

                     'S
                     '(F)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK ,BLANK ,BLANK) 1))
 `(F 3 (,LM ,BLANK ,BLANK ,BLANK)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK ,BLANK d d d ,BLANK) 1))
 `(F 5 (,LM ,BLANK d d d ,BLANK ,BLANK)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK d d ,BLANK ,BLANK) 1))
 `(F 4 (,LM ,BLANK d d ,BLANK ,BLANK)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK d d ,BLANK d d d) 1))
 `(F 7 (,LM ,BLANK d d d d d ,BLANK ,BLANK)))
```

Turing Machine Composition

```
;;  PRE: tape = (LM BLANK a BLANK b) AND i = 1
;; POST: tape = (LM BLANK a b) AND
;;       i = 3                        if a = b = 0
;;         = numd(a) + numd(b) + 2 otherwise,
;;       where numd(x) = the number of ds in x
(define ADD (make-tm '(S A B C D E F G)
                     '(d)
                     `(((S ,BLANK) (A ,RIGHT))
                       ((A d) (A ,RIGHT))
                       ((A ,BLANK) (B d))
                       ((B d) (B ,RIGHT))
                       ((B ,BLANK) (C ,LEFT))
                       ((C d) (D ,BLANK))
                       ((D ,BLANK) (E ,LEFT))
                       ((E d) (F ,RIGHT))
                       ((E ,BLANK) (G ,RIGHT))
                       ((G ,BLANK) (F ,RIGHT)))
                     'S
                     '(F)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK ,BLANK ,BLANK) 1))
 `(F 3 (,LM ,BLANK ,BLANK ,BLANK)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK ,BLANK d d d ,BLANK) 1))
 `(F 5 (,LM ,BLANK d d d ,BLANK ,BLANK)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK d d ,BLANK ,BLANK) 1))
 `(F 4 (,LM ,BLANK d d ,BLANK ,BLANK)))
(check-equal?
 (last (sm-showtransitions ADD `(,LM ,BLANK d d ,BLANK d d d) 1))
 `(F 7 (,LM ,BLANK d d d d d ,BLANK ,BLANK)))
```

Turing Machine Composition

- To establish correctness we use Hoare Logic (if you have read *Animated Program Design* then you are familiar with it)

Turing Machine Composition

- To establish correctness we use Hoare Logic (if you have read *Animated Program Design* then you are familiar with it)
- To establish the correctness of mutation-based computations use a triple for each rule:

```
((X i) (Y a))
```

- The corresponding triple is:

```
<Assertion about X's role>
((X i) (Y a))
<Assertion about Y's role>
```

- A triple is valid if assuming the precondition and the execution of the transition implies the postcondition

Turing Machine Composition

Theorem

ADD computes the addition of two natural numbers and leaves the head on the first blank after the result.

Proof.

- Proof by induction on the number of transitions performed by a computation

Turing Machine Composition

Theorem

ADD computes the addition of two natural numbers and leaves the head on the first blank after the result.

Proof.

- Proof by induction on the number of transitions performed by a computation
- Base case ($n = 0$): The machine starts in S and, by assumption, the precondition is true. Therefore, the role of S is satisfied.
- Inductive Step:
  Assume: State roles satisfied for $n = k$
  Show: that State roles satisfied for $n = k + 1$

Turing Machine Composition

Theorem

ADD computes the addition of two natural numbers and leaves the head on the first blank after the result.

Proof.

- Proof by induction on the number of transitions performed by a computation
- Base case ($n = 0$): The machine starts in S and, by assumption, the precondition is true. Therefore, the role of S is satisfied.
- Inductive Step:
  Assume: State roles satisfied for $n = k$
  Show: that State roles satisfied for $n = k + 1$
-
```
i = 1 and tape = (LM BLANK a BLANK b)
((S ,BLANK) (A ,RIGHT))
i <= numd(a) + 2 and tape = (LM BLANK a BLANK b)
```

Turing Machine Composition

Theorem

ADD computes the addition of two natural numbers and leaves the head on the first blank after the result.

Proof.

- Proof by induction on the number of transitions performed by a computation
- Base case (n = 0): The machine starts in S and, by assumption, the precondition is true. Therefore, the role of S is satisfied.
- Inductive Step:
  Assume: State roles satisfied for n = k
  Show: that State roles satisfied for n = k + 1
- ```
  i = 1 and tape = (LM BLANK a BLANK b)
  ((S ,BLANK) (A ,RIGHT))
  i <= numd(a) + 2 and tape = (LM BLANK a BLANK b)
  ```
- ```
  i <= numd(a) + 2 and tape = (LM BLANK a BLANK b)
  ((A d) (A ,RIGHT))
  i <= numd(a) + 2 and tape = (LM BLANK a BLANK b)
  ```

Turing Machine Composition

Theorem

ADD computes the addition of two natural numbers and leaves the head on the first blank after the result.

Proof.

- Proof by induction on the number of transitions performed by a computation
- Base case (n = 0): The machine starts in S and, by assumption, the precondition is true. Therefore, the role of S is satisfied.
- Inductive Step:
  Assume: State roles satisfied for n = k
  Show: that State roles satisfied for n = k + 1
- ```
  i = 1 and tape = (LM BLANK a BLANK b)
  ((S ,BLANK) (A ,RIGHT))
  i <= numd(a) + 2 and tape = (LM BLANK a BLANK b)
  ```
- ```
  i <= numd(a) + 2 and tape = (LM BLANK a BLANK b)
  ((A d) (A ,RIGHT))
  i <= numd(a) + 2 and tape = (LM BLANK a BLANK b)
  ```
- ```
  i <= numd(a) + 2 and tape = (LM BLANK a BLANK b)
  ((A ,BLANK) (B d))
  i <= numd(a) + numd(b) + 3 and tape = (LM BLANK a d b)
  ```
  ```

Turing Machine Composition

Theorem

ADD computes the addition of two natural numbers and leaves the head on the first blank after the result.

Proof.

- Proof by induction on the number of transitions performed by a computation
- Base case (n = 0): The machine starts in S and, by assumption, the precondition is true. Therefore, the role of S is satisfied.
- Inductive Step:
  Assume: State roles satisfied for n = k
  Show: that State roles satisfied for n = k + 1
- ```
  i = 1 and tape = (LM BLANK a BLANK b)
  ((S ,BLANK) (A ,RIGHT))
  i <= numd(a) + 2 and tape = (LM BLANK a BLANK b)
  ```
- ```
  i <= numd(a) + 2 and tape = (LM BLANK a BLANK b)
  ((A d) (A ,RIGHT))
  i <= numd(a) + 2 and tape = (LM BLANK a BLANK b)
  ```
- ```
  i <= numd(a) + 2 and tape = (LM BLANK a BLANK b)
  ((A ,BLANK) (B d))
  i <= numd(a) + numd(b) + 3 and tape = (LM BLANK a d b)
  ```
- ```
  i <= numd(a) + numd(b) + 3 and tape = (LM BLANK a d b)
  ((B d) (B ,RIGHT))
  i <= numd(a) + numd(b) + 3 and tape = (LM BLANK a d b)
  ```

Turing Machine Composition

Theorem

ADD computes the addition of two natural numbers and leaves the head on the first blank after the result.

Proof.

- Proof by induction on the number of transitions performed by a computation
- Base case (n = 0): The machine starts in S and, by assumption, the precondition is true. Therefore, the role of S is satisfied.
- Inductive Step:
  Assume: State roles satisfied for n = k
  Show: that State roles satisfied for n = k + 1
- ```
  i = 1 and tape = (LM BLANK a BLANK b)
  ((S ,BLANK) (A ,RIGHT))
  i <= numd(a) + 2 and tape = (LM BLANK a BLANK b)
  ```
- ```
  i <= numd(a) + 2 and tape = (LM BLANK a BLANK b)
  ((A d) (A ,RIGHT))
  i <= numd(a) + 2 and tape = (LM BLANK a BLANK b)
  ```
- ```
  i <= numd(a) + 2 and tape = (LM BLANK a BLANK b)
  ((A ,BLANK) (B d))
  i <= numd(a) + numd(b) + 3 and tape = (LM BLANK a d b)
  ```
- ```
  i <= numd(a) + numd(b) + 3 and tape = (LM BLANK a d b)
  ((B d) (B ,RIGHT))
  i <= numd(a) + numd(b) + 3 and tape = (LM BLANK a d b)
  ```
- ```
  i <= numd(a) + numd(b) + 3 and tape = (LM BLANK a d b)
  ((B ,BLANK) (C ,LEFT))
  i = numd(a) + numd(b) + 2 and tape = (LM BLANK a d b)
  ```
  ```

Turing Machine Composition

Theorem

ADD computes the addition of two natural numbers and leaves the head on the first blank after the result.

Proof.

- Inductive Step:
  Assume: State roles satisfied for n = k
  Show: that State roles satisfied for n = k + 1

  ⋮

- ```
  i = numd(a) + numd(b) + 2 and tape = (LM BLANK a d b)
  ((C d) (D ,BLANK))
  i = numd(a) + numd(b) + 2 and tape = (LM BLANK a b)
  ```
  ```

Turing Machine Composition

Theorem

ADD computes the addition of two natural numbers and leaves the head on the first blank after the result.

Proof.

- Inductive Step:
  Assume: State roles satisfied for n = k
  Show: that State roles satisfied for n = k + 1

  ⋮

- ```
  i = numd(a) + numd(b) + 2 and tape = (LM BLANK a d b)
  ((C d) (D ,BLANK))
  i = numd(a) + numd(b) + 2 and tape = (LM BLANK a b)
  ```
- ```
  i = numd(a) + numd(b) + 2 and tape = (LM BLANK a b)
  ((D ,BLANK) (E ,LEFT))
  i = numd(a) + numd(b) + 1 and tape = (LM BLANK a b)
  ```

Turing Machine Composition

Theorem

ADD computes the addition of two natural numbers and leaves the head on the first blank after the result.

Proof.

- Inductive Step:
  Assume: State roles satisfied for n = k
  Show: that State roles satisfied for n = k + 1

  ⋮

- ```
  i = numd(a) + numd(b) + 2 and tape = (LM BLANK a d b)
  ((C d) (D ,BLANK))
  i = numd(a) + numd(b) + 2 and tape = (LM BLANK a b)
  ```
- ```
  i = numd(a) + numd(b) + 2 and tape = (LM BLANK a b)
  ((D ,BLANK) (E ,LEFT))
  i = numd(a) + numd(b) + 1 and tape = (LM BLANK a b)
  ```
- ```
  i = numd(a) + numd(b) + 1 and tape = (LM BLANK a b)
  ((E d) (F ,RIGHT))
  i = 3                          if a = b = 0
    = numd(a) + numd(b) + 2 otherwise
  ∧ tape = (LM BLANK a b)
  ```
  ```

Turing Machine Composition

Theorem

ADD computes the addition of two natural numbers and leaves the head on the first blank after the result.

Proof.

- Inductive Step:
  Assume: State roles satisfied for n = k
  Show: that State roles satisfied for n = k + 1

  ⋮

- ```
  i = numd(a) + numd(b) + 2 and tape = (LM BLANK a d b)
  ((C d) (D ,BLANK))
  i = numd(a) + numd(b) + 2 and tape = (LM BLANK a b)
  ```
- ```
  i = numd(a) + numd(b) + 2 and tape = (LM BLANK a b)
  ((D ,BLANK) (E ,LEFT))
  i = numd(a) + numd(b) + 1 and tape = (LM BLANK a b)
  ```
- ```
  i = numd(a) + numd(b) + 1 and tape = (LM BLANK a b)
  ((E d) (F ,RIGHT))
  i = 3                          if a = b = 0
    = numd(a) + numd(b) + 2 otherwise
  ∧ tape = (LM BLANK a b)
  ```
- ```
  i = numd(a) + numd(b) + 1 and tape = (LM BLANK a b)
  ((E ,BLANK) (G ,RIGHT))
  i = numd(a) + numd(b) + 2 and tape = (LM BLANK BLANK)
  ```

Turing Machine Composition

Theorem

ADD computes the addition of two natural numbers and leaves the head on the first blank after the result.

Proof.

- Inductive Step:
  Assume: State roles satisfied for n = k
  Show: that State roles satisfied for n = k + 1

  ⋮

- ```
  i = numd(a) + numd(b) + 2 and tape = (LM BLANK a d b)
  ((C d) (D ,BLANK))
  i = numd(a) + numd(b) + 2 and tape = (LM BLANK a b)
  ```
- ```
  i = numd(a) + numd(b) + 2 and tape = (LM BLANK a b)
  ((D ,BLANK) (E ,LEFT))
  i = numd(a) + numd(b) + 1 and tape = (LM BLANK a b)
  ```
- ```
  i = numd(a) + numd(b) + 1 and tape = (LM BLANK a b)
  ((E d) (F ,RIGHT))
  i = 3                          if a = b = 0
    = numd(a) + numd(b) + 2 otherwise
  ∧ tape = (LM BLANK a b)
  ```
- ```
  i = numd(a) + numd(b) + 1 and tape = (LM BLANK a b)
  ((E ,BLANK) (G ,RIGHT))
  i = numd(a) + numd(b) + 2 and tape = (LM BLANK BLANK)
  ```
- ```
  i = numd(a) + numd(b) + 2 and tape = (LM BLANK BLANK)
  ((G ,BLANK) (F ,RIGHT))
  i = 3                          if a = b = 0
    = numd(a) + numd(b) + 2 otherwise
  ∧ tape = (LM BLANK a b)
  ```
  ```

Turing Machine Composition

- Copy machine

```
;PRE: tape=(LM BLANK w BLANK) && i=|w|+2
;POST: tape=(LM BLANK w BLANK w BLANK) && i=2|w|+3
```

Turing Machine Composition

- Copy machine

```
;PRE: tape=(LM BLANK w BLANK) && i=|w|+2
;POST: tape=(LM BLANK w BLANK w BLANK) && i=2|w|+3
```

- Move to the first blank to the left and then move right

Turing Machine Composition

- Copy machine

```
;PRE: tape=(LM BLANK w BLANK) && i=|w|+2
;POST: tape=(LM BLANK w BLANK w BLANK) && i=2|w|+3
```

- Move to the first blank to the left and then move right
- Branch on the element read
  - the next element to copy
  - the blank after `w`

Turing Machine Composition

- Copy machine

```
;PRE: tape=(LM BLANK w BLANK) && i=|w|+2
;POST: tape=(LM BLANK w BLANK w BLANK) && i=2|w|+3
```

- Move to the first blank to the left and then move right
- Branch on the element read
  - the next element to copy
  - the blank after `w`
- If a blank is read then the tape contains two copies of `w` separated by a blank and the machine must place the head correctly to satisfy the postcondition
  - The head must be moved to the first blank to the right and then moved left
  - A second branch is needed to determine if `w` is empty
  - If a blank is read then `w` is empty and the head is moved twice to the right to satisfy the postcondition
  - Otherwise, the head is moved once to the right to satisfy the postcondition

Turing Machine Composition

- Copy machine

```
;PRE: tape=(LM BLANK w BLANK) && i=|w|+2
;POST: tape=(LM BLANK w BLANK w BLANK) && i=2|w|+3
```

- Move to the first blank to the left and then move right
- Branch on the element read
  - the next element to copy
  - the blank after `w`
- If a blank is read then the tape contains two copies of `w` separated by a blank and the machine must place the head correctly to satisfy the postcondition
  - The head must be moved to the first blank to the right and then moved left
  - A second branch is needed to determine if `w` is empty
  - If a blank is read then `w` is empty and the head is moved twice to the right to satisfy the postcondition
  - Otherwise, the head is moved once to the right to satisfy the postcondition
- If an alphabet element is read at the first branching point then the machine must make a copy of the element and loop to repeat the process for any remaining elements to copy
  - The read value is captured in a variable and then overwritten with a blank as a placeholder to remember the read element's position
  - The machine moves to the second blank to the right and mutates this blank to be the value of the variable
  - It then moves to the second blank to the left and restores the value previously overwritten with a blank
  - The machine loops to move right and reach the first branching point again

Turing Machine Composition

Turing Machine Composition

- Correctness

  $(LM\ BLANK\ a_1 \ldots a_n\ \underline{BLANK}) \vee (LM\ BLANK\ BLANK\ \underline{BLANK})$
  FBL
  $(LM\ \underline{BLANK}\ a_1 \ldots a_n\ BLANK) \vee (LM\ BLANK\ \underline{BLANK}\ BLANK)$

Turing Machine Composition

- Correctness

  $(LM\ BLANK\ a_1 \ldots a_n\ \underline{BLANK}) \vee (LM\ BLANK\ BLANK\ \underline{BLANK})$
  FBL
  $(LM\ \underline{BLANK}\ a_1 \ldots a_n\ BLANK) \vee (LM\ BLANK\ \underline{BLANK}\ BLANK)$

- $(LM\ \underline{BLANK}\ a_1 \ldots a_n\ BLANK) \vee (LM\ BLANK\ \underline{BLANK}\ BLANK)$
  $\vee$ (LM BLANK $a_1 \ldots \underline{a_i} \ldots a_n$ BLANK $a_1 \ldots a_{i-1}$)
  R
  $(LM\ BLANK\ a_1 \ldots \underline{a_j} \ldots a_n\ BLANK)$
  $\vee\ (LM\ BLANK\ BLANK\ \underline{BLANK})$

Turing Machine Composition

- Correctness

  (LM BLANK $a_1 \ldots a_n$ *BLANK)* $\vee$ *(LM BLANK BLANK BLANK)*
  FBL
  (LM BLANK $a_1 \ldots a_n$ *BLANK)* $\vee$ *(LM BLANK BLANK BLANK)*

- *(LM BLANK* $a_1 \ldots a_n$ *BLANK)* $\vee$ *(LM BLANK BLANK BLANK)*
  $\vee$ (LM BLANK $a_1 \ldots \underline{a_i} \ldots a_n$ BLANK $a_1 \ldots a_{i-1}$)
  R
  (LM BLANK $a_1 \ldots \underline{a_j} \ldots a_n$ *BLANK)*
  $\vee$ *(LM BLANK BLANK BLANK)*

- *(LM BLANK* $a_1 \ldots \underline{a_i} \ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$*)*
  (list VAR 'k)
  (LM BLANK $a_1 \ldots \underline{a_i} \ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$*)* $\wedge$ $k{=}a_i$

Turing Machine Composition

- Correctness

  (LM BLANK $a_1 \ldots a_n$ *BLANK) ∨ (LM BLANK BLANK BLANK)*
  FBL
  (LM BLANK $a_1 \ldots a_n$ *BLANK) ∨ (LM BLANK BLANK BLANK)*
- *(LM BLANK* $a_1 \ldots a_n$ *BLANK) ∨ (LM BLANK BLANK BLANK)*
  ∨ (LM BLANK $a_1 \ldots \underline{a_i} \ldots a_n$ BLANK $a_1 \ldots a_{i-1}$)
  R
  (LM BLANK $a_1 \ldots \underline{a_j} \ldots a_n$ *BLANK)*
  ∨ *(LM BLANK BLANK BLANK)*
- *(LM BLANK* $a_1 \ldots \underline{a_i} \ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$*)*
  (list VAR 'k)
  (LM BLANK $a_1 \ldots \underline{a_i} \ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$*) ∧ k=*a_i
- FBR
  (LM BLANK $a_1 \ldots a_i \ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$*) ∧ k=*a_i

Turing Machine Composition

- Correctness

  (LM BLANK $a_1 \ldots a_n$ *BLANK)* $\vee$ *(LM BLANK BLANK BLANK)*
  FBL
  (LM BLANK $a_1 \ldots a_n$ *BLANK)* $\vee$ *(LM BLANK BLANK BLANK)*
- *(LM BLANK* $a_1 \ldots a_n$ *BLANK)* $\vee$ *(LM BLANK BLANK BLANK)*
  $\vee$ (LM BLANK $a_1 \ldots \underline{a_i} \ldots a_n$ BLANK $a_1 \ldots a_{i-1}$)
  R
  (LM BLANK $a_1 \ldots \underline{a_j} \ldots a_n$ *BLANK)*
  $\vee$ *(LM BLANK BLANK BLANK)*
- *(LM BLANK* $a_1 \ldots \underline{a_i} \ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$*)*
  (list VAR 'k)
  (LM BLANK $a_1 \ldots \underline{a_i} \ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$*)* $\wedge$ $k{=}a_i$
- FBR
  (LM BLANK $a_1 \ldots a_i \ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$*)* $\wedge$ $k{=}a_i$
- FBR
  (LM BLANK $a_1 \ldots$*BLANK*$\ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$ *BLANK)* $\wedge$ $k{=}a_i$

Turing Machine Composition

- Correctness

  (LM BLANK $a_1 \ldots a_n$ *BLANK)* $\vee$ *(LM BLANK BLANK BLANK)*
  FBL
  (LM BLANK $a_1 \ldots a_n$ *BLANK)* $\vee$ *(LM BLANK BLANK BLANK)*
- *(LM BLANK* $a_1 \ldots a_n$ *BLANK)* $\vee$ *(LM BLANK BLANK BLANK)*
  $\vee$ (LM BLANK $a_1 \ldots a_i \ldots a_n$ BLANK $a_1 \ldots a_{i-1}$)
  R
  (LM BLANK $a_1 \ldots a_j \ldots a_n$ *BLANK)*
  $\vee$ *(LM BLANK BLANK BLANK)*
- *(LM BLANK* $a_1 \ldots a_i \ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$*)*
  (list VAR 'k)
  (LM BLANK $a_1 \ldots a_i \ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$*)* $\wedge$ $k{=}a_i$
- FBR
  (LM BLANK $a_1 \ldots a_i \ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$*)* $\wedge$ $k{=}a_i$
- FBR
  (LM BLANK $a_1 \ldots$*BLANK*$\ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$ *BLANK)* $\wedge$ $k{=}a_i$
- 'k
  (LM BLANK $a_1 \ldots$*BLANK*$\ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$ a_i*)* $\wedge$ $k{=}a_i$

Turing Machine Composition

- Correctness

  (LM BLANK $a_1 \ldots a_n$ *BLANK)* $\vee$ *(LM BLANK BLANK BLANK)*
  FBL
  (LM BLANK $a_1 \ldots a_n$ *BLANK)* $\vee$ *(LM BLANK BLANK BLANK)*
- *(LM BLANK* $a_1 \ldots a_n$ *BLANK)* $\vee$ *(LM BLANK BLANK BLANK)*
  $\vee$ (LM BLANK $a_1 \ldots a_i \ldots a_n$ BLANK $a_1 \ldots a_{i-1}$)
  R
  (LM BLANK $a_1 \ldots a_j \ldots a_n$ *BLANK)*
  $\vee$ *(LM BLANK BLANK BLANK)*
- *(LM BLANK* $a_1 \ldots a_i \ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$*)*
  (list VAR 'k)
  (LM BLANK $a_1 \ldots a_i \ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$*)* $\wedge$ $k{=}a_i$
- FBR
  (LM BLANK $a_1 \ldots a_i \ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$*)* $\wedge$ $k{=}a_i$
- FBR
  (LM BLANK $a_1 \ldots$*BLANK*$\ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$ *BLANK)* $\wedge$ $k{=}a_i$
- 'k
  (LM BLANK $a_1 \ldots$*BLANK*$\ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$ a_i*)* $\wedge$ $k{=}a_i$
- FBL
  (LM BLANK $a_1 \ldots$*BLANK*$\ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$ a_i*)* && $k{=}a_i$

Turing Machine Composition

- FBL
  (LM BLANK $a_1 \ldots$ *BLANK*$\ldots a_n$ $\underline{\textit{BLANK}}$ $a_1 \ldots a_{i-1}$ a_i*) && k=*a_i
- FBL
  (LM BLANK $a_1 \ldots \underline{\textit{BLANK}} \ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$ a_i*) && k=*a_i

Turing Machine Composition

- FBL
  (LM BLANK $a_1 \ldots$ *BLANK* $\ldots a_n$ <u>*BLANK*</u> $a_1 \ldots a_{i-1}$ a_i*)* && *k=*a_i
- FBL
  (LM BLANK $a_1 \ldots$ <u>*BLANK*</u> $\ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$ a_i*)* && *k=*a_i
- `'k`
  (LM BLANK $a_1 \ldots$ <u>a_i</u> $\ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$ a_i*)*
- Precondition for label 0, starting with `R`, is met

Turing Machine Composition

- FBL
  (LM BLANK $a_1 \ldots$ *BLANK*$\ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$ a_i*)* && k=a_i
- FBL
  (LM BLANK $a_1 \ldots$ *BLANK*$\ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$ a_i*)* && k=a_i
- 'k
  (LM BLANK $a_1 \ldots a_i \ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$ a_i*)*
- Precondition for label 0, starting with R, is met
- Label 2 is only reached by reading a blank in the branch of the machine at label 0

  (LM BLANK $a_1 \ldots a_n$ *BLANK* $a_1 \ldots a_n$*)* $\vee$
  (LM BLANK BLANK BLANK)
  FBR
  (LM BLANK $a_1 \ldots a_n$ *BLANK* $a_1 \ldots a_n$ *BLANK)* $\vee$
  (LM BLANK BLANK BLANK BLANK)

Turing Machine Composition

- FBL
  (LM BLANK $a_1 \ldots$ *BLANK*$\ldots a_n$ $\underline{\textit{BLANK}}$ $a_1 \ldots a_{i-1}$ a_i*)* && *k=*a_i
- FBL
  (LM BLANK $a_1 \ldots \underline{\textit{BLANK}} \ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$ a_i*)* && *k=*a_i
- 'k
  (LM BLANK $a_1 \ldots \underline{a_i} \ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$ a_i*)*
- Precondition for label 0, starting with R, is met
- Label 2 is only reached by reading a blank in the branch of the machine at label 0

  (LM BLANK $a_1 \ldots a_n$ $\underline{\textit{BLANK}}$ $a_1 \ldots a_n$*)* $\vee$
  (LM BLANK BLANK $\underline{\textit{BLANK}}$*)*
  FBR
  (LM BLANK $a_1 \ldots a_n$ *BLANK* $a_1 \ldots a_n$ $\underline{\textit{BLANK}}$*)* $\vee$
  (LM BLANK BLANK BLANK $\underline{\textit{BLANK}}$*)*
- L
  (LM BLANK $a_1 \ldots a_n$ *BLANK* $a_1 \ldots \underline{a_n}$ *BLANK)* $\vee$
  (LM BLANK BLANK $\underline{\textit{BLANK}}$ *BLANK)*

Turing Machine Composition

- FBL
  (LM BLANK $a_1 \ldots$ *BLANK*$\ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$ a_i*)* && $k{=}a_i$
- FBL
  (LM BLANK $a_1 \ldots$ *BLANK*$\ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$ a_i*)* && $k{=}a_i$
- 'k
  (LM BLANK $a_1 \ldots a_i \ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$ a_i*)*
- Precondition for label 0, starting with R, is met
- Label 2 is only reached by reading a blank in the branch of the machine at label 0

  (LM BLANK $a_1 \ldots a_n$ *BLANK* $a_1 \ldots a_n$*)* $\vee$
  (LM BLANK BLANK BLANK)
  FBR
  (LM BLANK $a_1 \ldots a_n$ *BLANK* $a_1 \ldots a_n$ *BLANK)* $\vee$
  (LM BLANK BLANK BLANK BLANK)
- L
  (LM BLANK $a_1 \ldots a_n$ *BLANK* $a_1 \ldots a_n$ *BLANK)* $\vee$
  (LM BLANK BLANK BLANK BLANK)
- Label 3 is only reached if a blank is read at the end of the machine, L, at label 2: input word is empty and the head is over it:

  (LM BLANK BLANK BLANK BLANK)
  RR
  (LM BLANK BLANK BLANK BLANK BLANK)

Turing Machine Composition

- FBL
  (LM BLANK $a_1 \ldots$ *BLANK*$\ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$ a_i*)* && *k=*a_i
- FBL
  (LM BLANK $a_1 \ldots$ *BLANK*$\ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$ a_i*)* && *k=*a_i
- 'k
  (LM BLANK $a_1 \ldots a_i \ldots a_n$ *BLANK* $a_1 \ldots a_{i-1}$ a_i*)*
- Precondition for label 0, starting with R, is met
- Label 2 is only reached by reading a blank in the branch of the machine at label 0

  (LM BLANK $a_1 \ldots a_n$ *BLANK* $a_1 \ldots a_n$*)* $\vee$
  (LM BLANK BLANK BLANK)
  FBR
  (LM BLANK $a_1 \ldots a_n$ *BLANK* $a_1 \ldots a_n$ *BLANK)* $\vee$
  (LM BLANK BLANK BLANK BLANK)
- L
  (LM BLANK $a_1 \ldots a_n$ *BLANK* $a_1 \ldots a_n$ *BLANK)* $\vee$
  (LM BLANK BLANK BLANK BLANK)
- Label 3 is only reached if a blank is read at the end of the machine, L, at label 2: input word is empty and the head is over it:

  (LM BLANK BLANK BLANK BLANK)
  RR
  (LM BLANK BLANK BLANK BLANK BLANK)
- Label 4 is only reached if the head is over the last element of the nonempty input word's copy at the end of label 2:

  (LM BLANK $a_1 \ldots a_n$ *BLANK* $a_1 \ldots a_n$ *BLANK)*
  R
  (LM BLANK $a_1 \ldots a_n$ *BLANK* $a_1 \ldots a_n$ *BLANK)*

Turing Machine Composition

- HOMEWORK: 12–15

Turing Machine Extensions

- Turing machines are powerful enough to recognize languages that are not context-free and to compute arbitrary functions
- Hard to program!

Turing Machine Extensions

- Turing machines are powerful enough to recognize languages that are not context-free and to compute arbitrary functions
- Hard to program!
- An unanswered question in Computer Science is whether or not there are (undiscovered) machines that are more powerful
- Many attempts to strengthen Turing machines akin to strengthening `ndfa`'s with a stack
- None have been successful
- Many computer scientists today believe that the Turing machine is the most powerful computational device

Turing Machine Extensions

- Turing machines are powerful enough to recognize languages that are not context-free and to compute arbitrary functions
- Hard to program!
- An unanswered question in Computer Science is whether or not there are (undiscovered) machines that are more powerful
- Many attempts to strengthen Turing machines akin to strengthening `ndfa`'s with a stack
- None have been successful
- Many computer scientists today believe that the Turing machine is the most powerful computational device
- Why study new automata models that are not more powerful than the Turing machine?

Turing Machine Extensions

- Turing machines are powerful enough to recognize languages that are not context-free and to compute arbitrary functions
- Hard to program!
- An unanswered question in Computer Science is whether or not there are (undiscovered) machines that are more powerful
- Many attempts to strengthen Turing machines akin to strengthening `ndfa`'s with a stack
- None have been successful
- Many computer scientists today believe that the Turing machine is the most powerful computational device
- Why study new automata models that are not more powerful than the Turing machine?
- Make problem solving easier

Turing Machine Extensions

Turing Machine Extensions

- A multitape Turing machine (`mttm`) with `n`-tapes is an instance of:

  `(make-mttm K` Σ `S F` δ `n [Y])`

Turing Machine Extensions

- A multitape Turing machine (`mttm`) with `n`-tapes is an instance of:

  `(make-mttm K` Σ `S F` δ `n [Y])`

- K: A list of states. Each state is denoted by a symbol that represents a capital letter in the Roman alphabet
- Σ: A list of symbols or digits. Each symbol represents a lowercase letter in the Roman alphabet.
- S: The starting state. It must be a member of `K`.
- F: A list of final states. Each state must be a member of `K`.

Turing Machine Extensions

- A multitape Turing machine (`mttm`) with `n`-tapes is an instance of:

  `(make-mttm K Σ S F δ n [Y])`

- K: A list of states. Each state is denoted by a symbol that represents a capital letter in the Roman alphabet
- Σ: A list of symbols or digits. Each symbol represents a lowercase letter in the Roman alphabet.
- S: The starting state. It must be a member of `K`.
- F: A list of final states. Each state must be a member of `K`.
- δ:] A list of transition rules defining a transition relation. Each transition rule has the following type:

```
(list (list state (list symbol^n))
      (list state (list action^n)))
```

Turing Machine Extensions

- A multitape Turing machine (`mttm`) with `n`-tapes is an instance of:

  `(make-mttm K Σ S F δ n [Y])`

- K: A list of states. Each state is denoted by a symbol that represents a capital letter in the Roman alphabet
- Σ: A list of symbols or digits. Each symbol represents a lowercase letter in the Roman alphabet.
- S: The starting state. It must be a member of `K`.
- F: A list of final states. Each state must be a member of `K`.
- δ:] A list of transition rules defining a transition relation. Each transition rule has the following type:

  `(list (list state (list symbol`^{n}`))`
  `(list state (list action`^{n}`)))`

- n: A natural number greater than or equal to 1 representing the number of tapes.
- Y: An optional argument representing the accepting state for a language recognizer

Turing Machine Extensions

- A configuration

```
`(D
  (9 (,LM ,BLANK a a b b c c d d ,BLANK))
  (3 (,BLANK b b ,BLANK))
  (3 (,BLANK c c ,BLANK))
  (3 (,BLANK d d ,BLANK)))
```

- The machine is in state D. On tape 0 the head is on position 9 and on tapes 1–3 the heads are on position 3

Turing Machine Extensions

- A configuration

```
`(D
  (9 (,LM ,BLANK a a b b c c d d ,BLANK))
  (3 (,BLANK b b ,BLANK))
  (3 (,BLANK c c ,BLANK))
  (3 (,BLANK d d ,BLANK)))
```

- The machine is in state D. On tape 0 the head is on position 9 and on tapes 1–3 the heads are on position 3
- A transition made by the machine is denoted using $\vdash$
- $C_i \vdash C_j$ is valid for M if and only if M can move from C_i to C_j using a single transition

Turing Machine Extensions

- A configuration

```
`(D
  (9 (,LM ,BLANK a a b b c c d d ,BLANK))
  (3 (,BLANK b b ,BLANK))
  (3 (,BLANK c c ,BLANK))
  (3 (,BLANK d d ,BLANK)))
```

- The machine is in state D. On tape 0 the head is on position 9 and on tapes 1–3 the heads are on position 3
- A transition made by the machine is denoted using $\vdash$
- $C_i \vdash C_j$ is valid for M if and only if M can move from C_i to C_j using a single transition
- Zero or more moves by M is denoted using $\vdash^*$
- $C_i \vdash^* C_j$ is valid for M if and only if M can move from C_i to C_j using zero or more transitions

Turing Machine Extensions

- A configuration

```
`(D
  (9 (,LM ,BLANK a a b b c c d d ,BLANK))
  (3 (,BLANK b b ,BLANK))
  (3 (,BLANK c c ,BLANK))
  (3 (,BLANK d d ,BLANK)))
```

- The machine is in state `D`. On tape 0 the head is on position 9 and on tapes 1–3 the heads are on position 3
- A transition made by the machine is denoted using $\vdash$
- $C_i \vdash C_j$ is valid for `M` if and only if `M` can move from C_i to C_j using a single transition
- Zero or more moves by `M` is denoted using $\vdash^*$
- $C_i \vdash^* C_j$ is valid for `M` if and only if `M` can move from C_i to C_j using zero or more transitions
- An `mttm` language recognizer accepts a word, `w`, if there is a computation that reaches its accepting state

Turing Machine Extensions

- $L = \{w \mid w \text{ has equal number of as, bs, and cs}\}$

Turing Machine Extensions

- $L = \{w \mid w$ has equal number of as, bs, and cs$\}$
- Step 1
- The machine is named `EQABC` and $\Sigma = \{a\ b\ c\}$.

Turing Machine Extensions

- L = {w | w has equal number of as, bs, and cs}
- Step 1
- The machine is named EQABC and Σ = {a b c}.
- Let t0h be the position of the head on tape 0
- The precondition for EQABC is:

```
;; PRE: (LM BLANK w) AND t0h = 1 AND tapes 1-3 are empty AND
;;      t1h-t3h = 0
```

Turing Machine Extensions

- Step 2

```
(check-equal? (sm-apply EQABC `(,LM ,BLANK a a b b a c c) 1)
              'reject)
(check-equal? (sm-apply EQABC `(,LM ,BLANK a a a) 1)
              'reject)
(check-equal? (sm-apply EQABC `(,LM ,BLANK c c a b b) 1)
              'reject)
(check-equal? (sm-apply EQABC `(,LM ,BLANK) 1) 'accept)
(check-equal? (sm-apply EQABC `(,LM ,BLANK a c c b a b) 1)
              'accept)
(check-equal?
 (sm-apply EQABC
           `(,LM ,BLANK c c c a b b a a c b a b b c a) 1)
 'accept)
```

Turing Machine Extensions

- Step 3: Conditions and States

Turing Machine Extensions

- Step 3: Conditions and States
- Design Idea
- A 4-tape machine
- `T0`, contains the input word and is never mutated
- `T1`, `T2`, and `T3`, are used to store copies, respectively, of the as, bs, and cs

Turing Machine Extensions

- Step 3: Conditions and States
- Design Idea
- A 4-tape machine
- `T0`, contains the input word and is never mutated
- `T1`, `T2`, and `T3`, are used to store copies, respectively, of the as, bs, and cs
- The machine operates in two phases
- First phase, the input word elements are copied to the auxiliary tapes

Turing Machine Extensions

- Step 3: Conditions and States
- Design Idea
- A 4-tape machine
- `T0`, contains the input word and is never mutated
- `T1`, `T2`, and `T3`, are used to store copies, respectively, of the `a`s, `b`s, and `c`s
- The machine operates in two phases
- First phase, the input word elements are copied to the auxiliary tapes
- Second phase, the auxiliary tapes are simultaneously traversed to determine if for each `a` there is a matching `b` and a matching `c`

Turing Machine Extensions

- Step 3: Conditions and States
- Design Idea
- A 4-tape machine
- `T0`, contains the input word and is never mutated
- `T1`, `T2`, and `T3`, are used to store copies, respectively, of the as, bs, and cs
- The machine operates in two phases
- First phase, the input word elements are copied to the auxiliary tapes
- Second phase, the auxiliary tapes are simultaneously traversed to determine if for each a there is a matching b and a matching c
- If a blank is read on all three auxiliary tapes then the machine moves to accept
- If a blank is read on any tape, but not all tapes, then the machines moves to reject
- Otherwise, the machine moves the heads on each auxiliary tape to the left.

Turing Machine Extensions

- In more detail ...
- The machine starts by moving all 4 heads to the right: `T0` over the next element, if any, to copy and auxiliary heads on the next position to copy to

Turing Machine Extensions

- In more detail ...
- The machine starts by moving all 4 heads to the right: `T0` over the next element, if any, to copy and auxiliary heads on the next position to copy to
- Phase 1 that operates as follows:
  1. If a blank is read on `T0` then move the heads on the auxiliary tapes to the left and goto to Phase 2.
  2. Otherwise, copy the read symbol on `T0` to the respective auxiliary tape.
  3. Move the head of the auxiliary tape copied to and move `T0`'s head to the right and goto Phase 1's first step.

Turing Machine Extensions

- In more detail ...
- The machine starts by moving all 4 heads to the right: `T0` over the next element, if any, to copy and auxiliary heads on the next position to copy to
- Phase 1 that operates as follows:
  1. If a blank is read on `T0` then move the heads on the auxiliary tapes to the left and goto to Phase 2.
  2. Otherwise, copy the read symbol on `T0` to the respective auxiliary tape.
  3. Move the head of the auxiliary tape copied to and move `T0`'s head to the right and goto Phase 1's first step.
- Phase 2 operates as follows:
  1. If a blank is read on all tapes move to accept
  2. If an `a` is read on `T1`, a `b` is read on `T2`, and a `c` is read on `T3` then move the heads on all auxiliary tapes to the left and goto Phase 2's second step.
  3. Otherwise, halt and reject.

Turing Machine Extensions

- S:

```
tape 0 = (LM BLANK w) AND t0h = 1
tape 1-3 = (BLANK) AND t1h = t2h = t3h = 0
starting state
```

Turing Machine Extensions

- S:
  ```
  tape 0 = (LM BLANK w) AND t0h = 1
  tape 1-3 = (BLANK) AND t1h = t2h = t3h = 0
  starting state
  ```
- C:
  ```
  tape 0 = (LM BLANK w)
  t0h >= 2
  ```

Turing Machine Extensions

- S:
  ```
  tape 0 = (LM BLANK w) AND t0h = 1
  tape 1-3 = (BLANK) AND t1h = t2h = t3h = 0
  starting state
  ```
- C:
  ```
  tape 0 = (LM BLANK w)
  t0h >= 2
  ```
- ```
  tape 1 = (BLANK a* BLANK)
  tape 1 num a = if (eq? tape1[t1h] BLANK)
                    num a in tape0[2..t0h-1]
                    num a in tape0[2..t0h]
  t1h = if (eq? tape1[t1h] BLANK)
           num a in tape1[2..t0h-1] + 1
           num a in tape1[2..t0h] + 1
  ```
  ```

Turing Machine Extensions

- S:
  ```
  tape 0 = (LM BLANK w) AND t0h = 1
  tape 1-3 = (BLANK) AND t1h = t2h = t3h = 0
  starting state
  ```
- C:
  ```
  tape 0 = (LM BLANK w)
  t0h >= 2
  ```
- ```
  tape 1 = (BLANK a* BLANK)
  tape 1 num a = if (eq? tape1[t1h] BLANK)
                    num a in tape0[2..t0h-1]
                    num a in tape0[2..t0h]
  t1h = if (eq? tape1[t1h] BLANK)
           num a in tape1[2..t0h-1] + 1
           num a in tape1[2..t0h] + 1
  ```
- ```
  tape 2 = (BLANK b* BLANK)
  tape 2 num b = if (eq? tape2[t2h] BLANK)
                    num b in tape0[2..t0h-1]
                    num b in tape0[2..t0h]
  t2h = if (eq? tape2[t2h] BLANK)
           num b in tape2[2..t0h-1] + 1
           num b in tape2[2..t0h] + 1
  ```

Turing Machine Extensions

- S:
  ```
  tape 0 = (LM BLANK w) AND t0h = 1
  tape 1-3 = (BLANK) AND t1h = t2h = t3h = 0
  starting state
  ```
- C:
  ```
  tape 0 = (LM BLANK w)
  t0h >= 2
  ```
- ```
  tape 1 = (BLANK a* BLANK)
  tape 1 num a = if (eq? tape1[t1h] BLANK)
                    num a in tape0[2..t0h-1]
                    num a in tape0[2..t0h]
  t1h = if (eq? tape1[t1h] BLANK)
           num a in tape1[2..t0h-1] + 1
           num a in tape1[2..t0h] + 1
  ```
- ```
  tape 2 = (BLANK b* BLANK)
  tape 2 num b = if (eq? tape2[t2h] BLANK)
                    num b in tape0[2..t0h-1]
                    num b in tape0[2..t0h]
  t2h = if (eq? tape2[t2h] BLANK)
           num b in tape2[2..t0h-1] + 1
           num b in tape2[2..t0h] + 1
  ```
- ```
  tape 3 = (BLANK c* BLANK)
  tape 3 num c = if (eq? tape3[t3h] BLANK)
                    num c in tape0[2..t0h-1]
                    num c in tape0[2..t0h]
  t3h = if (eq? tape3[t3h] BLANK)
           num c in tape3[2..t0h-1] + 1
           num c in tape3[2..t0h] + 1
  ```
  ```

Turing Machine Extensions

- G:

```
tape 0 = (LM BLANK w)
t1 = (BLANK a*)
t2 = (BLANK b*)
t3 = (BLANK c*)
num a in t0 = num a in t1
num b in t0 = num b in t2
num c in t0 = num c in t3
(= |as matched| |bs matched| |cs matched|)
```

Turing Machine Extensions

- G:

```
tape 0 = (LM BLANK w)
t1 = (BLANK a*)
t2 = (BLANK b*)
t3 = (BLANK c*)
num a in t0 = num a in t1
num b in t0 = num b in t2
num c in t0 = num c in t3
(= |as matched| |bs matched| |cs matched|)
```

- Y:

```
num a in t0 = num a in t1
num b in t0 = num b in t2
num c in t0 = num c in t3
num a in t1 = num b in t2 = num c in t3
final and accepting state
```

Turing Machine Extensions

- Step 4: Transition Relation
-
```
(list (list 'S (list BLANK BLANK BLANK BLANK))
      (list 'C (list RIGHT RIGHT RIGHT RIGHT)))
```

Turing Machine Extensions

- Step 4: Transition Relation
-
```
(list (list 'S (list BLANK BLANK BLANK BLANK))
      (list 'C (list RIGHT RIGHT RIGHT RIGHT)))
```
-
```
;; read a on t0, copy to t1 and then move R on t0 and t1
(list (list 'C (list 'a BLANK BLANK BLANK))
      (list 'C (list 'a 'a BLANK BLANK)))
(list (list 'C (list 'a 'a BLANK BLANK))
      (list 'C (list RIGHT RIGHT BLANK BLANK)))
```

Turing Machine Extensions

- Step 4: Transition Relation
-
```
(list (list 'S (list BLANK BLANK BLANK BLANK))
      (list 'C (list RIGHT RIGHT RIGHT RIGHT)))
```
-
```
;; read a on t0, copy to t1 and then move R on t0 and t1
(list (list 'C (list 'a BLANK BLANK BLANK))
      (list 'C (list 'a 'a BLANK BLANK)))
(list (list 'C (list 'a 'a BLANK BLANK))
      (list 'C (list RIGHT RIGHT BLANK BLANK)))
```
-
```
(list (list 'C (list 'b BLANK BLANK BLANK))
      (list 'C (list 'b BLANK 'b BLANK)))
(list (list 'C (list 'b BLANK 'b BLANK))
      (list 'C (list RIGHT BLANK RIGHT BLANK)))
```

Turing Machine Extensions

- Step 4: Transition Relation
-
```
(list (list 'S (list BLANK BLANK BLANK BLANK))
      (list 'C (list RIGHT RIGHT RIGHT RIGHT)))
```
-
```
;; read a on t0, copy to t1 and then move R on t0 and t1
(list (list 'C (list 'a BLANK BLANK BLANK))
      (list 'C (list 'a 'a BLANK BLANK)))
(list (list 'C (list 'a 'a BLANK BLANK))
      (list 'C (list RIGHT RIGHT BLANK BLANK)))
```
-
```
(list (list 'C (list 'b BLANK BLANK BLANK))
      (list 'C (list 'b BLANK 'b BLANK)))
(list (list 'C (list 'b BLANK 'b BLANK))
      (list 'C (list RIGHT BLANK RIGHT BLANK)))
```
-
```
(list (list 'C (list 'c BLANK BLANK BLANK))
      (list 'C (list 'c BLANK BLANK 'c)))
(list (list 'C (list 'c BLANK BLANK 'c))
      (list 'C (list RIGHT BLANK BLANK RIGHT)))
```

Turing Machine Extensions

- Step 4: Transition Relation
-
```
(list (list 'S (list BLANK BLANK BLANK BLANK))
      (list 'C (list RIGHT RIGHT RIGHT RIGHT)))
```
-
```
;; read a on t0, copy to t1 and then move R on t0 and t1
(list (list 'C (list 'a BLANK BLANK BLANK))
      (list 'C (list 'a 'a BLANK BLANK)))
(list (list 'C (list 'a 'a BLANK BLANK))
      (list 'C (list RIGHT RIGHT BLANK BLANK)))
```
-
```
(list (list 'C (list 'b BLANK BLANK BLANK))
      (list 'C (list 'b BLANK 'b BLANK)))
(list (list 'C (list 'b BLANK 'b BLANK))
      (list 'C (list RIGHT BLANK RIGHT BLANK)))
```
-
```
(list (list 'C (list 'c BLANK BLANK BLANK))
      (list 'C (list 'c BLANK BLANK 'c)))
(list (list 'C (list 'c BLANK BLANK 'c))
      (list 'C (list RIGHT BLANK BLANK RIGHT)))
```
-
```
(list (list 'C (list BLANK BLANK BLANK BLANK))
      (list 'G (list BLANK LEFT LEFT LEFT)))
```

Turing Machine Extensions

- ```
  (list (list 'G (list BLANK 'a 'b 'c))
        (list 'G (list BLANK LEFT LEFT LEFT)))
  ```
  ```

Turing Machine Extensions

- ```
  (list (list 'G (list BLANK 'a 'b 'c))
        (list 'G (list BLANK LEFT LEFT LEFT)))
  ```
- ```
  (list (list 'G (list BLANK BLANK BLANK BLANK))
        (list 'Y (list BLANK BLANK BLANK BLANK)))
  ```

Turing Machine Extensions

- Steps 5 & 6: Implement and test

```
#lang fsm
;; L = w | w has an equal number of a, b, and c  PRE: t0 = (LM BLANK w) AND i = 1
(define EQABC (make-mttm '(S C G Y)
                         '(a b c)
                         'S
                         '(Y)
                         (list
                          (list (list 'S (list BLANK BLANK BLANK BLANK))
                                (list 'C (list RIGHT RIGHT RIGHT RIGHT)))
                          (list (list 'C (list 'a BLANK BLANK BLANK))
                                (list 'C (list 'a 'a BLANK BLANK)))
                          (list (list 'C (list 'a 'a BLANK BLANK))
                                (list 'C (list RIGHT RIGHT BLANK BLANK)))
                          (list (list 'C (list 'b BLANK BLANK BLANK))
                                (list 'C (list 'b BLANK 'b BLANK)))
                          (list (list 'C (list 'b BLANK 'b BLANK))
                                (list 'C (list RIGHT BLANK RIGHT BLANK)))
                          (list (list 'C (list 'c BLANK BLANK BLANK))
                                (list 'C (list 'c BLANK BLANK 'c)))
                          (list (list 'C (list 'c BLANK BLANK 'c))
                                (list 'C (list RIGHT BLANK BLANK RIGHT)))
                          (list (list 'C (list BLANK BLANK BLANK BLANK))
                                (list 'G (list BLANK LEFT LEFT LEFT)))
                          (list (list 'G (list BLANK BLANK BLANK BLANK))
                                (list 'Y (list BLANK BLANK BLANK BLANK)))
                          (list (list 'G (list BLANK 'a 'b 'c))
                                (list 'G (list BLANK LEFT LEFT LEFT))))
                         4
                         'Y))
(check-equal? (sm-apply EQABC `(,LM ,BLANK a a b b a c c) 1) 'reject)
(check-equal? (sm-apply EQABC `(,LM ,BLANK a a a) 1) 'reject)
(check-equal? (sm-apply EQABC `(,LM ,BLANK c c a b b) 1) 'reject)
```

Turing Machine Extensions

- Step 7: Invariant Predicates
- For `mttms`, an invariant predicate takes a (listof tape-config)
- Tape 0–Tape N-1

Turing Machine Extensions

- Step 7: Invariant Predicates
- For `mttms`, an invariant predicate takes a (listof tape-config)
- Tape 0–Tape N-1
- A tape configuration: (tape-head tape-value)

Turing Machine Extensions

- Step 7: Invariant Predicates

```
S: tape 0 = (LM BLANK w) AND t0h = 1
   tape 1-3 = (BLANK) AND t1h = t2h = t3h = 0
   starting state
```

Turing Machine Extensions

- Step 7: Invariant Predicates

```
S: tape 0 = (LM BLANK w) AND t0h = 1
   tape 1-3 = (BLANK) AND t1h = t2h = t3h = 0
   starting state
```

- ;; (listof tape-config) → Boolean

```
;; Purpose: Determine if S's conditions are met
(define (S-INV tape-configs)
  (let* [(t0c (first tape-configs))
         (t1c (second tape-configs))
         (t2c (third tape-configs))
         (t3c (fourth tape-configs))
         (t0h (first t0c))
         (t0 (second t0c))
         (t1h (first t1c))
         (t1 (second t1c))
         (t2h (first t2c))
         (t2 (second t2c))
         (t3h (first t3c))
         (t3 (second t3c))]
```

Turing Machine Extensions

- Step 7: Invariant Predicates

```
S: tape 0 = (LM BLANK w) AND t0h = 1
   tape 1-3 = (BLANK) AND t1h = t2h = t3h = 0
   starting state
```

- ;; (listof tape-config) → Boolean

```
;; Purpose: Determine if S's conditions are met
(define (S-INV tape-configs)
  (let* [(t0c (first tape-configs))
         (t1c (second tape-configs))
         (t2c (third tape-configs))
         (t3c (fourth tape-configs))
         (t0h (first t0c))
         (t0 (second t0c))
         (t1h (first t1c))
         (t1 (second t1c))
         (t2h (first t2c))
         (t2 (second t2c))
         (t3h (first t3c))
         (t3 (second t3c))]
```

-

```
    (and (= t0h 1)
         (= t1h 0)
         (= t2h 0)
         (= t3h 0)
         (eq? (list-ref t0 t0h) BLANK)
         (equal? `(,BLANK) t1)
         (equal? `(,BLANK) t2)
         (equal? `(,BLANK) t3))))
```

Turing Machine Extensions

- C:
```
tape 0 = (LM BLANK w)
t0h >= 2

tape 1 = (BLANK a* BLANK)
tape 1 num a = if (eq? tape1[t1h] BLANK)
                  num a in tape0[2..t0h-1]
                  num a in tape0[2..t0h]
t1h = if (eq? tape1[t1h] BLANK)
         num a in tape1[2..t0h-1] + 1
         num a in tape1[2..t0h] + 1

tape 2 = (BLANK b* BLANK)
tape 2 num b = if (eq? tape2[t2h] BLANK)
                  num b in tape0[2..t0h-1]
                  num b in tape0[2..t0h]
t2h = if (eq? tape2[t2h] BLANK)
         num b in tape2[2..t0h-1] + 1
         num b in tape2[2..t0h] + 1

tape 3 = (BLANK c* BLANK)
tape 3 num c = if (eq? tape3[t3h] BLANK)
                  num c in tape0[2..t0h-1]
                  num c in tape0[2..t0h]
t3h = if (eq? tape3[t3h] BLANK)
         num c in tape3[2..t0h-1] + 1
         num c in tape3[2..t0h] + 1
```

Turing Machine Extensions

```
(define (C-INV tape-configs)
  (let* [...
         (readt0 (if (or (eq? (list-ref t0 t0h) (list-ref t1 t1h))
                         (eq? (list-ref t0 t0h) (list-ref t2 t2h))
                         (eq? (list-ref t0 t0h) (list-ref t3 t3h)))
                     (take (rest (rest t0)) (- t0h 1))   ;; head over read element
                     (take (rest (rest t0)) (- t0h 2)))) ;; head over unread element
         (written-t1 (if (eq? (list-ref t1 t1h) BLANK)
                         (rest (drop-right t1 1))
                         (rest t1)))
         (written-t2 (if (eq? (list-ref t2 t2h) BLANK)
                         (rest (drop-right t2 1))
                         (rest t2)))
         (written-t3 (if (eq? (list-ref t3 t3h) BLANK)
                         (rest (drop-right t3 1))
                         (rest t3)))]
```

Turing Machine Extensions

- ```
  (define (C-INV tape-configs)
    (let* [...
           (readt0 (if (or (eq? (list-ref t0 t0h) (list-ref t1 t1h))
                           (eq? (list-ref t0 t0h) (list-ref t2 t2h))
                           (eq? (list-ref t0 t0h) (list-ref t3 t3h)))
                       (take (rest (rest t0)) (- t0h 1))   ;; head over read element
                       (take (rest (rest t0)) (- t0h 2)))) ;; head over unread element
           (written-t1 (if (eq? (list-ref t1 t1h) BLANK)
                           (rest (drop-right t1 1))
                           (rest t1)))
           (written-t2 (if (eq? (list-ref t2 t2h) BLANK)
                           (rest (drop-right t2 1))
                           (rest t2)))
           (written-t3 (if (eq? (list-ref t3 t3h) BLANK)
                           (rest (drop-right t3 1))
                           (rest t3)))]
  ```
- ```
      (and (>= t0h 2)
           (or (eq? (list-ref t1 t1h) BLANK)
               (eq? (list-ref t1 t1h) (list-ref t0 t0h)))
           (or (eq? (list-ref t2 t2h) BLANK)
               (eq? (list-ref t2 t2h) (list-ref t0 t0h)))
           (or (eq? (list-ref t3 t3h) BLANK)
               (eq? (list-ref t3 t3h) (list-ref t0 t0h)))
  ```

Turing Machine Extensions

```
(define (C-INV tape-configs)
  (let* [...
         (readt0 (if (or (eq? (list-ref t0 t0h) (list-ref t1 t1h))
                         (eq? (list-ref t0 t0h) (list-ref t2 t2h))
                         (eq? (list-ref t0 t0h) (list-ref t3 t3h)))
                     (take (rest (rest t0)) (- t0h 1))   ;; head over read element
                     (take (rest (rest t0)) (- t0h 2)))) ;; head over unread element
         (written-t1 (if (eq? (list-ref t1 t1h) BLANK)
                         (rest (drop-right t1 1))
                         (rest t1)))
         (written-t2 (if (eq? (list-ref t2 t2h) BLANK)
                         (rest (drop-right t2 1))
                         (rest t2)))
         (written-t3 (if (eq? (list-ref t3 t3h) BLANK)
                         (rest (drop-right t3 1))
                         (rest t3)))]
    (and (>= t0h 2)
         (or (eq? (list-ref t1 t1h) BLANK)
             (eq? (list-ref t1 t1h) (list-ref t0 t0h)))
         (or (eq? (list-ref t2 t2h) BLANK)
             (eq? (list-ref t2 t2h) (list-ref t0 t0h)))
         (or (eq? (list-ref t3 t3h) BLANK)
             (eq? (list-ref t3 t3h) (list-ref t0 t0h)))
         (andmap (λ (s) (eq? s 'a)) written-t1)
         (andmap (λ (s) (eq? s 'b)) written-t2)
         (andmap (λ (s) (eq? s 'c)) written-t3)
```

Turing Machine Extensions

```
(define (C-INV tape-configs)
  (let* [...
         (readt0 (if (or (eq? (list-ref t0 t0h) (list-ref t1 t1h))
                         (eq? (list-ref t0 t0h) (list-ref t2 t2h))
                         (eq? (list-ref t0 t0h) (list-ref t3 t3h)))
                     (take (rest (rest t0)) (- t0h 1))   ;; head over read element
                     (take (rest (rest t0)) (- t0h 2)))) ;; head over unread element
         (written-t1 (if (eq? (list-ref t1 t1h) BLANK)
                         (rest (drop-right t1 1))
                         (rest t1)))
         (written-t2 (if (eq? (list-ref t2 t2h) BLANK)
                         (rest (drop-right t2 1))
                         (rest t2)))
         (written-t3 (if (eq? (list-ref t3 t3h) BLANK)
                         (rest (drop-right t3 1))
                         (rest t3)))]
    (and (>= t0h 2)
         (or (eq? (list-ref t1 t1h) BLANK)
             (eq? (list-ref t1 t1h) (list-ref t0 t0h)))
         (or (eq? (list-ref t2 t2h) BLANK)
             (eq? (list-ref t2 t2h) (list-ref t0 t0h)))
         (or (eq? (list-ref t3 t3h) BLANK)
             (eq? (list-ref t3 t3h) (list-ref t0 t0h)))
         (andmap (λ (s) (eq? s 'a)) written-t1)
         (andmap (λ (s) (eq? s 'b)) written-t2)
         (andmap (λ (s) (eq? s 'c)) written-t3)
         (eq? (list-ref t1 0) BLANK) (eq? (list-ref t2 0) BLANK)
         (eq? (list-ref t3 0) BLANK)
```

Turing Machine Extensions

```
(define (C-INV tape-configs)
  (let* [...
         (readt0 (if (or (eq? (list-ref t0 t0h) (list-ref t1 t1h))
                         (eq? (list-ref t0 t0h) (list-ref t2 t2h))
                         (eq? (list-ref t0 t0h) (list-ref t3 t3h)))
                     (take (rest (rest t0)) (- t0h 1))   ;; head over read element
                     (take (rest (rest t0)) (- t0h 2)))) ;; head over unread element
         (written-t1 (if (eq? (list-ref t1 t1h) BLANK)
                         (rest (drop-right t1 1))
                         (rest t1)))
         (written-t2 (if (eq? (list-ref t2 t2h) BLANK)
                         (rest (drop-right t2 1))
                         (rest t2)))
         (written-t3 (if (eq? (list-ref t3 t3h) BLANK)
                         (rest (drop-right t3 1))
                         (rest t3)))]
    (and (>= t0h 2)
         (or (eq? (list-ref t1 t1h) BLANK)
             (eq? (list-ref t1 t1h) (list-ref t0 t0h)))
         (or (eq? (list-ref t2 t2h) BLANK)
             (eq? (list-ref t2 t2h) (list-ref t0 t0h)))
         (or (eq? (list-ref t3 t3h) BLANK)
             (eq? (list-ref t3 t3h) (list-ref t0 t0h)))
         (andmap (λ (s) (eq? s 'a)) written-t1)
         (andmap (λ (s) (eq? s 'b)) written-t2)
         (andmap (λ (s) (eq? s 'c)) written-t3)
         (eq? (list-ref t1 0) BLANK) (eq? (list-ref t2 0) BLANK)
         (eq? (list-ref t3 0) BLANK)
         (equal? (filter (λ (s) (eq? s 'a)) readt0)
                 (filter (λ (s) (eq? s 'a)) t1)) ;;same other tapes
         (equal? (filter (λ (s) (eq? s 'b)) readt0)
                 (filter (λ (s) (eq? s 'b)) t2))
         (equal? (filter (λ (s) (eq? s 'c)) readt0)
```

Turing Machine Extensions

- ```
  G: tape 0 = (LM BLANK w)
     t1 = (BLANK a*)
     t2 = (BLANK b*)
     t3 = (BLANK c*)
     num a in t0 = num a in t1
     num b in t0 = num b in t2
     num c in t0 = num c in t3
     (= |as matched| |bs matched| |cs matched|)
  ```
  ```

Turing Machine Extensions

- ```
  G: tape 0 = (LM BLANK w)
     t1 = (BLANK a*)
     t2 = (BLANK b*)
     t3 = (BLANK c*)
     num a in t0 = num a in t1
     num b in t0 = num b in t2
     num c in t0 = num c in t3
     (= |as matched| |bs matched| |cs matched|)
  ```
- ```
  (define (G-INV tape-configs)
    (let* [...
           (as-matched (if (= (length t1) 2)
                           '()
                           (drop-right (drop t1 (add1 t1h)) 1)))
           (bs-matched (if (= (length t2) 2)
                           '()
                           (drop-right (drop t2 (add1 t2h)) 1)))
           (cs-matched (if (= (length t3) 2)
                           '()
                           (drop-right (drop t3 (add1 t3h)) 1)))]
  ```
- ```
      (and (>= t0h 2)
           (< t1h (length t1))
           (< t2h (length t2))
           (< t3h (length t3))
           (andmap (λ (s) (eq? s 'a)) (rest (drop-right t1 1)))
           (andmap (λ (s) (eq? s 'b)) (rest (drop-right t2 1)))
           (andmap (λ (s) (eq? s 'c)) (rest (drop-right t3 1)))
           (equal? (filter (λ (s) (eq? s 'a)) t0)
                   (filter (λ (s) (eq? s 'a)) t1))
           (equal? (filter (λ (s) (eq? s 'b)) t0)
                   (filter (λ (s) (eq? s 'b)) t2))
           (equal? (filter (λ (s) (eq? s 'c)) t0)
                   (filter (λ (s) (eq? s 'c)) t3))
           (= (length as-matched) (length bs-matched) (length cs-matched)))))
  ```
  ```

Turing Machine Extensions

- ```
  Y: num a in t0 = num a in t1
     num b in t0 = num b in t2
     num c in t0 = num c in t3
     num a in t1 = num b in t2 = num c in t3
     final and accepting state
  ```
  ```

Turing Machine Extensions

- ```
  Y: num a in t0 = num a in t1
     num b in t0 = num b in t2
     num c in t0 = num c in t3
     num a in t1 = num b in t2 = num c in t3
     final and accepting state
  ```
- ```
  (define (Y-INV tape-configs)
    (let* [...
           (t0as (filter (λ (s) (eq? s 'a)) t0))
           (t0bs (filter (λ (s) (eq? s 'b)) t0))
           (t0cs (filter (λ (s) (eq? s 'c)) t0))]
      (and (>= t0h 2)
           (eq? (list-ref t0 t0h) BLANK)
           (= (length t0as) (length t0bs) (length t0cs)))))
  ```

Turing Machine Extensions

- Before attempting the proof validate!

Turing Machine Extensions

- Before attempting the proof validate!
- Machine correctness is achieved in two steps

Turing Machine Extensions

- Before attempting the proof validate!
- Machine correctness is achieved in two steps
- Prove that invariants always hold

Turing Machine Extensions

- Before attempting the proof validate!
- Machine correctness is achieved in two steps
- Prove that invariants always hold
- Prove L = L(EQABC)

Turing Machine Extensions

Theorem

For `EQABC` state invariant predicates hold

-

Turing Machine Extensions

Theorem

For `EQABC` state invariant predicates hold

-
- Base case: n = 0. By assumption, when the machine starts, the precondition holds and it is in state `S`. This means that tape 0 = (LM BLANK w) and tape 0's head position = 1. In addition, the heads on the auxiliary tapes start at position 0. This configuration suffices to establish that `S-INV` holds.

Turing Machine Extensions

Theorem

For `EQABC` state invariant predicates hold

-
- Base case: n = 0. By assumption, when the machine starts, the precondition holds and it is in state `S`. This means that tape 0 = (LM BLANK w) and tape 0's head position = 1. In addition, the heads on the auxiliary tapes start at position 0. This configuration suffices to establish that `S-INV` holds.
- Inductive Step:
  Assume: State invariant predicates hold for n = k.
  Show: State invariant predicates hold for n = k+1.
  For every transition rule, `(A a B)`, we must show that `A-INV` and the execution of the rule establish `B-INV`.

Turing Machine Extensions

- `((S (_ _ _ _)) (C (R R R R)))`
  By inductive hypothesis, S-INV holds. This means that head positions are [1,0,0,0] and that [BLANK,BLANK,BLANK,BLANK] are read. Using this rule moves all heads to the right. C-INV holds, because tape 0's head position is 2, at t1-t3's head positions a BLANK is read, position 0 on the auxiliary tapes contains a blank, all the as (i.e., 0) in t0[1..t0h-1] are copied to t1, all the bs in t0[1..t0h-1] are copied to t2, and all the cs in t0[1..t0h-1] are copied to t3.

Turing Machine Extensions

- `((C (a _ _ _)) (C (a a _ _)))`
  By inductive hypothesis, C-INV holds. This means tape 0's head position is greater than or equal to 2, the head position on tape 0 contains an a (by use of this rule), the head position on every auxiliary tape contains a blank (by use of this rule), after the initial blank tape 1 only contains as, after the initial blank tape 2 only contains bs, after the initial blank tape 3 only contains cs, and the number of elements on each auxiliary tape equals the number of corresponding elements in the read part of tape 0. Using this rule copies tape 0's a to tape 1. C-INV holds, because t0h ≥ 2, the elements read on tapes 0 and 1 match, the element read on both tapes 2 and 3 is a blank, each auxiliary tape starts with a blank, after the initial blank tape 1 only contains as, after the initial blank tape 2 only contains bs, after the initial blank tape 3 only contains cs, and the number of elements on each auxiliary tape equals the number of corresponding elements in the read part of tape 0 (including the newly read a).

Turing Machine Extensions

- `((C (a _ _ _)) (C (a a _ _)))`
  By inductive hypothesis, C-INV holds. This means tape 0's head position is greater than or equal to 2, the head position on tape 0 contains an a (by use of this rule), the head position on every auxiliary tape contains a blank (by use of this rule), after the initial blank tape 1 only contains as, after the initial blank tape 2 only contains bs, after the initial blank tape 3 only contains cs, and the number of elements on each auxiliary tape equals the number of corresponding elements in the read part of tape 0. Using this rule copies tape 0's a to tape 1. C-INV holds, because t0h ≥ 2, the elements read on tapes 0 and 1 match, the element read on both tapes 2 and 3 is a blank, each auxiliary tape starts with a blank, after the initial blank tape 1 only contains as, after the initial blank tape 2 only contains bs, after the initial blank tape 3 only contains cs, and the number of elements on each auxiliary tape equals the number of corresponding elements in the read part of tape 0 (including the newly read a).
- `((C (a a _ _)) (C (R R _ _)))`
  By inductive hypothesis, C-INV holds. This means tape 0's head position is greater than or equal to 2, the head positions on tape 0 and tape 1 contain an a (by use of this rule), the head position on tapes 2 and 3 contain a blank (by use of this rule), after the initial blank tape 1 only contains as, after the initial blank tape 2 only contains bs, after the initial blank tape 3 only contains cs, and the number of elements on each auxiliary tape equals the number of corresponding elements in the read part of tape 0. Using this rules moves the heads on tapes 0 and 1 to the right. C-INV holds, because t0h ≥ 2, a blank is read on all auxiliary tapes, after the initial blank tape 1 only contains as, after the initial blank tape 2 only contains bs, after the initial blank tape 3 only contains cs, and the number of elements on each auxiliary tape equals the number of corresponding elements in the read part of tape 0 (that does not contain the current element under the head).

Turing Machine Extensions

- `((C (a _ _ _)) (C (a a _ _)))`
  By inductive hypothesis, C-INV holds. This means tape 0's head position is greater than or equal to 2, the head position on tape 0 contains an a (by use of this rule), the head position on every auxiliary tape contains a blank (by use of this rule), after the initial blank tape 1 only contains as, after the initial blank tape 2 only contains bs, after the initial blank tape 3 only contains cs, and the number of elements on each auxiliary tape equals the number of corresponding elements in the read part of tape 0. Using this rule copies tape 0's a to tape 1. C-INV holds, because t0h ≥ 2, the elements read on tapes 0 and 1 match, the element read on both tapes 2 and 3 is a blank, each auxiliary tape starts with a blank, after the initial blank tape 1 only contains as, after the initial blank tape 2 only contains bs, after the initial blank tape 3 only contains cs, and the number of elements on each auxiliary tape equals the number of corresponding elements in the read part of tape 0 (including the newly read a).
- `((C (a a _ _)) (C (R R _ _)))`
  By inductive hypothesis, C-INV holds. This means tape 0's head position is greater than or equal to 2, the head positions on tape 0 and tape 1 contain an a (by use of this rule), the head position on tapes 2 and 3 contain a blank (by use of this rule), after the initial blank tape 1 only contains as, after the initial blank tape 2 only contains bs, after the initial blank tape 3 only contains cs, and the number of elements on each auxiliary tape equals the number of corresponding elements in the read part of tape 0. Using this rules moves the heads on tapes 0 and 1 to the right. C-INV holds, because t0h ≥ 2, a blank is read on all auxiliary tapes, after the initial blank tape 1 only contains as, after the initial blank tape 2 only contains bs, after the initial blank tape 3 only contains cs, and the number of elements on each auxiliary tape equals the number of corresponding elements in the read part of tape 0 (that does not contain the current element under the head).
- The correctness for copying bs and cs from tape 0 to, respectively, tapes 2 and 3 and for subsequently moving the corresponding heads is the same as done for an a and are in the textbook.

Turing Machine Extensions

- `((C (_ _ _ _)) (G (_ L L L)))`
  By inductive hypothesis, C-INV holds. This means tape 0's head position is greater than or equal to 2, under the head of each tape there is a blank (by use of this rule), after the initial blank tape 1 only contains as, after the initial blank tape 2 only contains bs, after the initial blank tape 3 only contains cs, and the number of elements on each auxiliary tape equals the number of corresponding elements in the read part of tape 0. Using this rule means that all word elements on tape 0 have been copied to the corresponding auxiliary tapes and that the heads on tapes 1–3 are moved left to the last element copied or to the initial blank if no elements have been copied. G-INV holds, because tape 0's head position is greater than or equal to 2, the head position on each auxiliary tape is less than the tape's length, between the initial blank and the ending blank tape 1 only contains as, between the initial blank and the ending blank tape 2 only contains bs, between the initial blank and the ending blank tape 3 only contains cs, the auxiliary tapes contain the number of corresponding elements found of tape 0 (which is the read part of tape 0), and the length of matched as, bs, and cs (i.e., 0) on tapes 1–3 is the same.

Turing Machine Extensions

- `((C (_ _ _ _)) (G (_ L L L)))`
  By inductive hypothesis, C-INV holds. This means tape 0's head position is greater than or equal to 2, under the head of each tape there is a blank (by use of this rule), after the initial blank tape 1 only contains as, after the initial blank tape 2 only contains bs, after the initial blank tape 3 only contains cs, and the number of elements on each auxiliary tape equals the number of corresponding elements in the read part of tape 0. Using this rule means that all word elements on tape 0 have been copied to the corresponding auxiliary tapes and that the heads on tapes 1–3 are moved left to the last element copied or to the initial blank if no elements have been copied. G-INV holds, because tape 0's head position is greater than or equal to 2, the head position on each auxiliary tape is less than the tape's length, between the initial blank and the ending blank tape 1 only contains as, between the initial blank and the ending blank tape 2 only contains bs, between the initial blank and the ending blank tape 3 only contains cs, the auxiliary tapes contain the number of corresponding elements found of tape 0 (which is the read part of tape 0), and the length of matched as, bs, and cs (i.e., 0) on tapes 1–3 is the same.
- `((G (_ a b c)) (G (_ L L L)))`
  By inductive hypothesis, G-INV holds. This means tape 0's head position is greater than or equal to 2, the head position on each auxiliary tape is less than the tape's length, between the initial blank and the ending blank tape 1 only contains as, between the initial blank and the ending blank tape 2 only contains bs, between the initial blank and the ending blank tape 3 only contains cs, the auxiliary tapes contain the number of corresponding elements found of tape 0 (which is the read part of tape 0), and the length of matched as, bs, and cs on tapes 1–3 is the same. These facts and reading a blank on tape 0 establishes that all word elements on tape 0 have been copied to the corresponding auxiliary tapes. Using this rule means that one more a, b, and c are matched on tapes 1–3 and the heads are moved to the left. Therefore, G-INV golds after using this rule.

Turing Machine Extensions

- `((C (_ _ _ _)) (G (_ L L L)))`
  By inductive hypothesis, C-INV holds. This means tape 0's head position is greater than or equal to 2, under the head of each tape there is a blank (by use of this rule), after the initial blank tape 1 only contains as, after the initial blank tape 2 only contains bs, after the initial blank tape 3 only contains cs, and the number of elements on each auxiliary tape equals the number of corresponding elements in the read part of tape 0. Using this rule means that all word elements on tape 0 have been copied to the corresponding auxiliary tapes and that the heads on tapes 1–3 are moved left to the last element copied or to the initial blank if no elements have been copied. G-INV holds, because tape 0's head position is greater than or equal to 2, the head position on each auxiliary tape is less than the tape's length, between the initial blank and the ending blank tape 1 only contains as, between the initial blank and the ending blank tape 2 only contains bs, between the initial blank and the ending blank tape 3 only contains cs, the auxiliary tapes contain the number of corresponding elements found of tape 0 (which is the read part of tape 0), and the length of matched as, bs, and cs (i.e., 0) on tapes 1–3 is the same.
- `((G (_ a b c)) (G (_ L L L)))`
  By inductive hypothesis, G-INV holds. This means tape 0's head position is greater than or equal to 2, the head position on each auxiliary tape is less than the tape's length, between the initial blank and the ending blank tape 1 only contains as, between the initial blank and the ending blank tape 2 only contains bs, between the initial blank and the ending blank tape 3 only contains cs, the auxiliary tapes contain the number of corresponding elements found of tape 0 (which is the read part of tape 0), and the length of matched as, bs, and cs on tapes 1–3 is the same. These facts and reading a blank on tape 0 establishes that all word elements on tape 0 have been copied to the corresponding auxiliary tapes. Using this rule means that one more a, b, and c are matched on tapes 1–3 and the heads are moved to the left. Therefore, G-INV golds after using this rule.
- `((G (_ _ _ _)) (Y (_ _ _ _)))`
  By IH, G-INV holds. Tape 0's head position ≥ 2, the head position on each auxiliary tape is less than the tape's length, between the initial blank and the ending blank: tape 1 only contains as, tape 2 only contains bs, tape 3 only contains cs, the auxiliary tapes contain the number of corresponding elements found of tape 0 (which is the read part of tape 0), and the length of matched as, bs, and cs on tapes 1–3 is the same. Using this rule means that all elements on the auxiliary tapes have been matched. Given that all word elements on tape 0 have been copied to the corresponding auxiliary tapes, the word has an equal number of as, bs, and cs. In addition, the head on tape 0 remains greater than or equal to 2, and a blank is read on tape 0. Thus, Y-INV holds.

Turing Machine Extensions

Lemma

$w \in L \Leftrightarrow w \in L(EQABC)$

•

Turing Machine Extensions

Lemma

$w \in L \Leftrightarrow w \in L(EQABC)$

-

Proof.

($\Rightarrow$) Assume w∈L
This means that w has the same number of as, bs, and cs. Given that invariants always hold, there is a computation on w that copies all its elements to the corresponding auxiliary tapes, matches the elements, and ends in Y. Thus, w in L(EQABC).
($\Leftarrow$) Assume w∈L(EQABC) This means that, after processing w, EQABC halts in Y. Given that invariants always hold, w has the same number of as, bs, and cs. Thus, w∈L. □

-

Turing Machine Extensions

Lemma

$w \notin L \Leftrightarrow w \notin L(EQABC)$

•

Turing Machine Extensions

Lemma

$w \notin L \Leftrightarrow w \notin L(EQABC)$

-

Proof.

($\Rightarrow$) Assume $w \notin L$
This means that w does not have the same number of as, bs, and cs. Given that invariants always hold, there is no EQABC computation that halts in Y. Thus, $w \notin L(EQABC)$.
($\Leftarrow$) Assume $w \notin L(EQABC)$
This means there is no EQABC computation that ends in Y. Given that invariants always hold assuming the precondition holds, there must be at least one a, b, or c that cannot be matched and the machine halts in G. Thus, $w \notin L$. □

-

Turing Machine Extensions

Theorem

$L = L(EQABC)$

Proof.

The proof follows from the previous two lemmas. □

•

Turing Machine Extensions

- HOMEWORK: 1, 3, 4

Turing Machine Extensions

- Are multitape Turing machines more powerful than standard Turing machines?

Turing Machine Extensions

- Are multitape Turing machines more powerful than standard Turing machines?
- The answer is no.
- This means that a standard Turing machine can simulate any multitape Turing machine
- We shall sketch how to build a standard Turing machine from a multitape Turing machine

Turing Machine Extensions

- A `tm` must represent all the information contained in the `mttm`'s tape configurations

Turing Machine Extensions

- A `tm` must represent all the information contained in the `mttm`'s tape configurations
- How can this information be represented on a single tape?

Turing Machine Extensions

- A `tm` must represent all the information contained in the `mttm`'s tape configurations
- How can this information be represented on a single tape?
- We shall draw inspiration from hard disk technology and imagine that a standard `tm`'s tape is divided into tracks
- A k–tape `mttm` is simulated using 2k tracks
- The even tracks, numbered 2j, represent tape j's content in the `mttm`
- The odd tracks, numbered 2j+1, capture the head's position on tape j

Turing Machine Extensions

Tape 0	LM	a	a		a	b			b
Tape 1		b	a	b	a				
Tape 2		b	a	a	b	b	b	a	

LM	LM	a	a		a	b			b	
	0	0	0	0	1	0	0	0	0	
		b	a	b	a					
	0	0	1	0	0	0	0	0	0	
		b	a	a	b	b	b	a		
	0	0	0	0	0	0	1	0	0	

Turing Machine Extensions

LM	LM	a	a		a	b			b	
	0	0	0	0	1	0	0	0	0	
		b	a	b	a					
	0	0	1	0	0	0	0	0	0	
		b	a	a	b	b	b	a		
	0	0	0	0	0	0	1	0	0	

LM	p	q	r	s	t	u	v	w	x	

- The division into tracks, of course, is an abstraction that must be implemented
- The simulating `tm` must contain the `mttm`'s alphabet to receive the same
- Each position in the simulating `tm`'s tape is either `LM`, `BLANK`, or a symbol representing an element in $(\Sigma \times \{0\ 1\})^k$,
- The simulating `tm`'s alphabet must have a symbol for each possible column in the track representation of the multiple tapes

Turing Machine Extensions

Theorem

Let M = (make-mttm K Σ S F R n [Y]). There exists a Turing machine, M' = (make-tm K' Σ' S' F R' [Y]), that simulates M.

Turing Machine Extensions

Theorem

Let M = (make-mttm K Σ S F R n [Y]). There exists a Turing machine, M' = (make-tm K' Σ' S' F R' [Y]), that simulates M.

Proof.

- Assume the precondition for M is: PRE: tape 0 = (LM BLANK w) and t0h = 1

Turing Machine Extensions

Theorem

Let M = (make-mttm K Σ S F R n [Y]). There exists a Turing machine, M' = (make-tm K' Σ' S' F R' [Y]), that simulates M.

Proof.

- Assume the precondition for M is: PRE: tape 0 = (LM BLANK w) and t0h = 1
- Three primary phases M' operates in:

```
1. Make M''s tape represent M's initial configuration.
   a. Shift w to the right one space
   b. To represent the beginning of the k tapes and write Σ''s symbol for:
      LM
      0
      BLANK
      1
      .
      .
      .
   This symbol captures: head on tape 0 is not on the left-end marker
   c. Move right to capture tape 0's head position by writing symbol for:
      BLANK
      1
      BLANK
      0
      .
      .
      .
   d. Move right until a blank is read. For each e∈Σ read, write Σ''s symbol for:
      e
      0
      BLANK
      0
      .
      .
```

Turing Machine Extensions

Theorem

Let M = (make-mttm K Σ S F R n [Y]). There exists a Turing machine, M' = (make-tm K' Σ' S' F R' [Y]), that simulates M.

Proof.

- Assume the precondition for M is: PRE: tape 0 = (LM BLANK w) and t0h = 1
- Three primary phases M' operates in:

```
2. Simulate M. For each transition that would be made by M, M' starts on the
   first blank that is not encoded into tracks and performs the
   following operations:

        1. Move left and determine what would be read by each of M's head as
           described in the design idea and return the head to the first
           blank that is not encoded into tracks.

        2. Move left to update the tracks as M would update its tapes
           and heads.
```

□

Turing Machine Extensions

Theorem

Let M = (make-mttm K Σ S F R n [Y]). There exists a Turing machine, M' = (make-tm K' Σ' S' F R' [Y]), that simulates M.

Proof.

- Assume the precondition for M is: PRE: tape 0 = (LM BLANK w) and t0h = 1
- Three primary phases M' operates in:

```
2. Simulate M. For each transition that would be made by M, M' starts on the
   first blank that is not encoded into tracks and performs the
   following operations:

        1. Move left and determine what would be read by each of M's head as
           described in the design idea and return the head to the first
           blank that is not encoded into tracks.

        2. Move left to update the tracks as M would update its tapes
           and heads.
```

□

- Many design choices are not described for phase 2, but it ought to be clear that M' simulates M
- M' performs many transitions to simulate one transition performed by M

Turing Machine Extensions

Theorem

Let M = (make-mttm K Σ S F R n [Y]). There exists a Turing machine, M' = (make-tm K' Σ' S' F R' [Y]), that simulates M.

Proof.

- Assume the precondition for M is: PRE: tape 0 = (LM BLANK w) and t0h = 1
- Three primary phases M' operates in:

```
3. If M would halt, convert the contents of the tape (i.e., M''s tape) to
   single track
  a. Ignores all tracks except tracks 0 and 1
  b. The content of track 0 is written to the tape
  c. the head is placed at the position indicated by track 1
  d. M' moves to the state M would halt in and halts itself.
```

□

Turing Machine Extensions

- Does a Turing machine with one tape and multiple heads operating on the tape provide us with more computational power?
- Each head operates independently of the others
- If a head moves right or left it does not interfere with other heads
- A synchronization mechanism must be implemented when two or more heads try to write to the same tape position (assume it exists)
- How can a standard `tm` simulate a Turing machine with one tape and multiple heads?

Turing Machine Extensions

- Does a Turing machine with one tape and multiple heads operating on the tape provide us with more computational power?
- Each head operates independently of the others
- If a head moves right or left it does not interfere with other heads
- A synchronization mechanism must be implemented when two or more heads try to write to the same tape position (assume it exists)
- How can a standard `tm` simulate a Turing machine with one tape and multiple heads?
- A representation strategy similar to the one used to simulate multiple tapes may be used
- The standard Turing machine operates on an encoded multiple track tape
- The first track represents the input tape and the rest of the tracks record each head's position

Turing Machine Extensions

- Does a Turing machine with one tape and multiple heads operating on the tape provide us with more computational power?
- Each head operates independently of the others
- If a head moves right or left it does not interfere with other heads
- A synchronization mechanism must be implemented when two or more heads try to write to the same tape position (assume it exists)
- How can a standard `tm` simulate a Turing machine with one tape and multiple heads?
- A representation strategy similar to the one used to simulate multiple tapes may be used
- The standard Turing machine operates on an encoded multiple track tape
- The first track represents the input tape and the rest of the tracks record each head's position
- To simulate one step of the multiple heads machine, the standard `tm` scans the tape twice
- The first to determine the elements read by each of the multiple heads
- The second to simulate the action of each head (i.e., mutate the first track or move the head)

Turing Machine Extensions

- Is a Turing machine with a tape that is infinite both to the right and the left more powerful than a standard Turing machine?
- In such a machine, at the beginning only the input is written on the tape and the rest of the positions are blank
- There is no left-end marker in such a machine.

Turing Machine Extensions

- Is a Turing machine with a tape that is infinite both to the right and the left more powerful than a standard Turing machine?
- In such a machine, at the beginning only the input is written on the tape and the rest of the positions are blank
- There is no left-end marker in such a machine.
- A two-way infinite input tape Turing machine may be simulated by a 2-tape `mttm`
- Initially, the first tape contains the input and the second tape is blank
- During a computation, if the head of the two-way infinite input tape Turing machine is on the first element of the input or to right of it then the standard `tm` operates on tape 0
- Otherwise, it operates on tape 1 where the elements are written in reversed order (right and left moves are simulated, respectively, by left and right moves on tape 1)

Turing Machine Extensions

FINAL EXAM PROJECT

- Design, build, validate, and verify a constructor that given a `pda` builds a `mttm` that simulates it
- Problems: 6–12

Context-Sensitive Grammars

- Turing machines are automata that are capable of deciding languages that are not context-free
- We have seen that less powerful automata, `dfa`s and `pda`s, have language generator counterparts called, respectively, regular grammars/expressions and context-free grammars
- Are there grammars for the languages that Turing machines decide or semidecide?

Context-Sensitive Grammars

- Turing machines are automata that are capable of deciding languages that are not context-free
- We have seen that less powerful automata, `dfa`s and `pda`s, have language generator counterparts called, respectively, regular grammars/expressions and context-free grammars
- Are there grammars for the languages that Turing machines decide or semidecide?
- We shall now explore *context-sensitive grammars* (`csg`s).
- Unlike previous grammars, the left hand side of a rule may consist of any number of terminal and nonterminal symbols as long as there is at least one nonterminal symbol
- The length of a partially generated word may arbitrarily decrease in a single step

Context-Sensitive Grammars

- Formally:

  `A context-free grammar is an instance of (make-csg N Σ R S)`

Context-Sensitive Grammars

- Formally:

  `A context-free grammar is an instance of (make-csg N Σ R S)`

- Each production rule is of the form:

  $$(N \cup \Sigma)^* N (N \cup \Sigma)^* \rightarrow (N \cup \Sigma)^*$$

  $$(N \cup \Sigma)^* N (N \cup \Sigma)^* \rightarrow \mathtt{EMP}$$

- The set of regular grammars and the set of context-free grammars are both proper subsets of the set of context-sensitive grammars

Context-Sensitive Grammars

- Formally:

  `A context-free grammar is an instance of (make-csg N Σ R S)`

- Each production rule is of the form:

  $$(N \cup \Sigma)^* N (N \cup \Sigma)^* \rightarrow (N \cup \Sigma)^*$$

  $$(N \cup \Sigma)^* N (N \cup \Sigma)^* \rightarrow \mathtt{EMP}$$

- The set of regular grammars and the set of context-free grammars are both proper subsets of the set of context-sensitive grammars
- L(G) denotes the language generated by G: $\{w \mid w \in \Sigma^* \wedge S \rightarrow_G^* w\}$
- A language, L, is context-sensitive if L = L(G) for some context-sensitive grammar G

Context-Sensitive Grammars

- $\mathtt{L} = \mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$

Context-Sensitive Grammars

- $\mathtt{L} = \mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$
- We shall design the grammar following the steps of the design recipe
- The syntactic category represented by a nonterminal may be less clear-cut than their counterparts in `cfg`s and `rg`s
- For instance, a nonterminal may not represent a syntactic category on its own given that it may only generate something in the proper context
- It is sometimes useful to think of a nonterminal as a promise to generate a desired subword in the proper context or as partially defining a syntactic category

Context-Sensitive Grammars

- $L = a^n b^n c^n$
- We shall design the grammar following the steps of the design recipe
- The syntactic category represented by a nonterminal may be less clear-cut than their counterparts in `cfgs` and `rgs`
- For instance, a nonterminal may not represent a syntactic category on its own given that it may only generate something in the proper context
- It is sometimes useful to think of a nonterminal as a promise to generate a desired subword in the proper context or as partially defining a syntactic category
- Finding a derivation may take very long making some tests unpractical
- Attempting to find derivations for a word not in the language may take forever. Again, this is especially true when the grammar uses nondeterminism
- Unlike for `rgs` and `cfgs`, there is no algorithm to distinguish when a search has become futile. This has rather important consequences for us as programmers
- Careful design is of paramount importance

Context-Sensitive Grammars

- Design Idea
- `A`, `B`, and `C` represent promises to generate as, bs, and cs in the proper context
- This generation ends with a nonterminal, say `G`, that represents a promise to generate cs in the proper context
- The proper context, of course, is that all the `A`s must be before the all the `B`s which must be before all the `C`s with the `G` at the end

Context-Sensitive Grammars

- Design Idea
- `A`, `B`, and `C` represent promises to generate as, bs, and cs in the proper context
- This generation ends with a nonterminal, say `G`, that represents a promise to generate cs in the proper context
- The proper context, of course, is that all the `A`s must be before the all the `B`s which must be before all the `C`s with the `G` at the end
- Before generating terminal symbols, the grammar nondeterministically rearranges the `A`s, `B`s, and `C`s in the right order

Context-Sensitive Grammars

- Design Idea
- `A`, `B`, and `C` represent promises to generate as, bs, and cs in the proper context
- This generation ends with a nonterminal, say `G`, that represents a promise to generate cs in the proper context
- The proper context, of course, is that all the `A`s must be before the all the `B`s which must be before all the `C`s with the `G` at the end
- Before generating terminal symbols, the grammar nondeterministically rearranges the `A`s, `B`s, and `C`s in the right order
- Finally, the grammar traverses the partially generated word, from right to left, to generate the cs
- If there is `CG` in the partially generated word then the `G` may be moved left and a `c` is generated: `CG` generates `Gc`

Context-Sensitive Grammars

- Design Idea
- `A`, `B`, and `C` represent promises to generate as, bs, and cs in the proper context
- This generation ends with a nonterminal, say `G`, that represents a promise to generate cs in the proper context
- The proper context, of course, is that all the `A`s must be before the all the `B`s which must be before all the `C`s with the `G` at the end
- Before generating terminal symbols, the grammar nondeterministically rearranges the `A`s, `B`s, and `C`s in the right order
- Finally, the grammar traverses the partially generated word, from right to left, to generate the cs
- If there is `CG` in the partially generated word then the `G` may be moved left and a c is generated: `CG` generates `Gc`
- After the last c is generated, the grammar generates a nonterminal, say `H`, that represents a promise to generate the bs in the proper context and continues the traversal to generate all the bs

Context-Sensitive Grammars

- Design Idea
- `A`, `B`, and `C` represent promises to generate as, bs, and cs in the proper context
- This generation ends with a nonterminal, say `G`, that represents a promise to generate cs in the proper context
- The proper context, of course, is that all the `A`s must be before the all the `B`s which must be before all the `C`s with the `G` at the end
- Before generating terminal symbols, the grammar nondeterministically rearranges the `A`s, `B`s, and `C`s in the right order
- Finally, the grammar traverses the partially generated word, from right to left, to generate the cs
- If there is `CG` in the partially generated word then the `G` may be moved left and a `c` is generated: `CG` generates `Gc`
- After the last `c` is generated, the grammar generates a nonterminal, say `H`, that represents a promise to generate the `b`s in the proper context and continues the traversal to generate all the `b`s
- When the last `b` is generated, the grammar generates a nonterminal, say `I`, that represents a promise to generate the as in the proper context and continues the traversal to generate all the as

Context-Sensitive Grammars

- Design Idea
- `A`, `B`, and `C` represent promises to generate as, bs, and cs in the proper context
- This generation ends with a nonterminal, say `G`, that represents a promise to generate cs in the proper context
- The proper context, of course, is that all the `A`s must be before the all the `B`s which must be before all the `C`s with the `G` at the end
- Before generating terminal symbols, the grammar nondeterministically rearranges the `A`s, `B`s, and `C`s in the right order
- Finally, the grammar traverses the partially generated word, from right to left, to generate the cs
- If there is `CG` in the partially generated word then the `G` may be moved left and a `c` is generated: `CG` generates `Gc`
- After the last `c` is generated, the grammar generates a nonterminal, say `H`, that represents a promise to generate the `b`s in the proper context and continues the traversal to generate all the `b`s
- When the last `b` is generated, the grammar generates a nonterminal, say `I`, that represents a promise to generate the as in the proper context and continues the traversal to generate all the as
- When the last `a` is generated `I` generates `EMP` and the derivation is done

Context-Sensitive Grammars

- Design Idea
- Consider the derivation for aaabbbccc
- Nondeterministically generate `ABC` three times ending with `G`:

  `ABCABCABCG`

Context-Sensitive Grammars

- Design Idea
- Consider the derivation for aaabbbccc
- Nondeterministically generate ABC three times ending with G:

  ABCABCABCG
- Rearrange the As, Bs, and Cs to be in the right order:

  AAABBBCCCG

Context-Sensitive Grammars

- Design Idea
- Consider the derivation for `aaabbbccc`
- Nondeterministically generate `ABC` three times ending with `G`:

  `ABCABCABCG`
- Rearrange the `A`s, `B`s, and `C`s to be in the right order:

  `AAABBBCCCG`
- Generate the `c`s:

  `AAABBBCCGc`
  `AAABBBCGcc`
  `AAABBBGccc`

Context-Sensitive Grammars

- Design Idea
- Consider the derivation for aaabbbccc
- Nondeterministically generate ABC three times ending with G:

  ABCABCABCG
- Rearrange the As, Bs, and Cs to be in the right order:

  AAABBBCCCG
- Generate the cs:

  AAABBBCCGc
  AAABBBCGcc
  AAABBBGccc
- Nondeterministically generates an H from G:

  AAABBBHccc

Context-Sensitive Grammars

- Design Idea
- Consider the derivation for aaabbbccc
- Nondeterministically generate ABC three times ending with G:

  ```
  ABCABCABCG
  ```

- Rearrange the As, Bs, and Cs to be in the right order:

  ```
  AAABBBCCCG
  ```

- Generate the cs:

  ```
  AAABBBCCGc
  AAABBBCGcc
  AAABBBGccc
  ```

- Nondeterministically generates an H from G:

  ```
  AAABBBHccc
  ```

- Generate bs:

  ```
  AAABBHbccc
  AAABHbbccc
  AAAHbbbccc
  ```

Context-Sensitive Grammars

- Design Idea
- Consider the derivation for aaabbbccc
- Nondeterministically generate ABC three times ending with G:

  ```
  ABCABCABCG
  ```

- Rearrange the As, Bs, and Cs to be in the right order:

  ```
  AAABBBCCCG
  ```

- Generate the cs:

  ```
  AAABBBCCGc
  AAABBBCGcc
  AAABBBGccc
  ```

- Nondeterministically generates an H from G:

  ```
  AAABBBHccc
  ```

- Generate bs:

  ```
  AAABBHbccc
  AAABHbbccc
  AAAHbbbccc
  ```

- Nondeterministically generates an I:

  ```
  AAAIbbbccc
  ```

Context-Sensitive Grammars

- Design Idea
- Consider the derivation for aaabbbccc
- Nondeterministically generate ABC three times ending with G:

  ```
  ABCABCABCG
  ```

- Rearrange the As, Bs, and Cs to be in the right order:

  ```
  AAABBBCCCG
  ```

- Generate the cs:

  ```
  AAABBBCCGc
  AAABBBCGcc
  AAABBBGccc
  ```

- Nondeterministically generates an H from G:

  ```
  AAABBBHccc
  ```

- Generate bs:

  ```
  AAABBHbccc
  AAABHbbccc
  AAAHbbbccc
  ```

- Nondeterministically generates an I:

  ```
  AAAIbbbccc
  ```

- Generate as:

  ```
  AAIabbbccc
  AIaabbbccc
  Iaaabbbccc
  ```

Context-Sensitive Grammars

- Design Idea
- Consider the derivation for aaabbbccc
- Nondeterministically generate ABC three times ending with G:

  ABCABCABCG
- Rearrange the As, Bs, and Cs to be in the right order:

  AAABBBCCCG
- Generate the cs:

  AAABBBCCGc
  AAABBBCGcc
  AAABBBGccc
- Nondeterministically generates an H from G:

  AAABBBHccc
- Generate bs:

  AAABBHbccc
  AAABHbbccc
  AAAHbbbccc
- Nondeterministically generates an I:

  AAAIbbbccc
- Generate as:

  AAIabbbccc
  AIaabbbccc
  Iaaabbbccc
- Nondeterministically generate EMP:

  aaabbbccc

Context-Sensitive Grammars

- A descriptive name for the grammar is `anbncn`
- The alphabet is {`a b c`}.

Context-Sensitive Grammars

- A descriptive name for the grammar is `anbncn`
- The alphabet is {`a b c`}.
- `A` and `I` together may generate an `a` in the context `AI`

Context-Sensitive Grammars

- A descriptive name for the grammar is `anbncn`
- The alphabet is {`a b c`}.
- `A` and `I` together may generate an `a` in the context `AI`
- `B` and `H` together may generate a `b` in the context `BH`

Context-Sensitive Grammars

- A descriptive name for the grammar is `anbncn`
- The alphabet is {`a b c`}.
- `A` and `I` together may generate an `a` in the context `AI`
- `B` and `H` together may generate a `b` in the context `BH`
- `C` and `G` together may generate an `a` in the context `CG`

Context-Sensitive Grammars

- A descriptive name for the grammar is `anbncn`
- The alphabet is {`a b c`}.
- `A` and `I` together may generate an `a` in the context `AI`
- `B` and `H` together may generate a `b` in the context `BH`
- `C` and `G` together may generate an `a` in the context `CG`
- We document the syntactic categories as follows:

```
;; Syntactic Categories
;; S: generates words in a^nb^nc^n
;; A,I: A promise to generate an a in the context AI
;; B,H: A promise to generate an b in the context BH
;; C,G: A promise to generate an c in the context CG
```

Context-Sensitive Grammars

- Production Rules
- S needs to generate an arbitrary number of ABC and needs to generate G:

```
(S ,ARROW ,EMP)
(S ,ARROW ABCS)   S ⇒ |A|=|B|=|C|
(S ,ARROW G)
```

Context-Sensitive Grammars

- Production Rules
- S needs to generate an arbitrary number of ABC and needs to generate G:

```
(S ,ARROW ,EMP)
(S ,ARROW ABCS)   S ⇒ |A|=|B|=|C|
(S ,ARROW G)
```

- To rearrange the As, Bs, and Cs, nonterminals out of order are put in the right order:

```
(BA ,ARROW AB)
(CA ,ARROW AC)
(CB ,ARROW BC)
```

Context-Sensitive Grammars

- Production Rules
- S needs to generate an arbitrary number of ABC and needs to generate G:

```
(S ,ARROW ,EMP)
(S ,ARROW ABCS)   S ⇒ |A|=|B|=|C|
(S ,ARROW G)
```

- To rearrange the As, Bs, and Cs, nonterminals out of order are put in the right order:

```
(BA ,ARROW AB)
(CA ,ARROW AC)
(CB ,ARROW BC)
```

- C and G are used to generate cs, move G right to left, and nondeterministically generate an H:

```
(CG ,ARROW Gc)   G ⇒ only c* to the right
(BG  ,ARROW BH)
```

Context-Sensitive Grammars

- Production Rules
- S needs to generate an arbitrary number of ABC and needs to generate G:

```
(S ,ARROW ,EMP)
(S ,ARROW ABCS)   S ⇒ |A|=|B|=|C|
(S ,ARROW G)
```

- To rearrange the As, Bs, and Cs, nonterminals out of order are put in the right order:

```
(BA ,ARROW AB)
(CA ,ARROW AC)
(CB ,ARROW BC)
```

- C and G are used to generate cs, move G right to left, and nondeterministically generate an H:

```
(CG ,ARROW Gc)   G ⇒ only c* to the right
(BG  ,ARROW BH)
```

- B and H are used to generate bs, move H right to left, and nondeterministically generate an I:

```
(BH ,ARROW Hb)   H ⇒ only b*c+ to the right
(AH ,ARROW AI)
```

Context-Sensitive Grammars

- Production Rules
- S needs to generate an arbitrary number of ABC and needs to generate G:

```
(S ,ARROW ,EMP)
(S ,ARROW ABCS)   S ⇒ |A|=|B|=|C|
(S ,ARROW G)
```

- To rearrange the As, Bs, and Cs, nonterminals out of order are put in the right order:

```
(BA ,ARROW AB)
(CA ,ARROW AC)
(CB ,ARROW BC)
```

- C and G are used to generate cs, move G right to left, and nondeterministically generate an H:

```
(CG ,ARROW Gc)   G ⇒ only c* to the right
(BG  ,ARROW BH)
```

- B and H are used to generate bs, move H right to left, and nondeterministically generate an I:

```
(BH ,ARROW Hb)   H ⇒ only b*c+ to the right
(AH ,ARROW AI)
```

- A and I are used to generate as, move I right to left, and nondeterministically generate EMP:

```
(AI ,ARROW Ia)   I ⇒ only a*b+c+ to the right
(I ,ARROW ,EMP)
```

Context-Sensitive Grammars

- Some trial and error may be needed to make sure tests run in a reasonable amount of time
- A good heuristic is to test relatively short words that are in the language:

```
(check-derive? anbncn '(a b c) '(a a b b c c) '())
;; (check=not=derive? anbncn '(a) '(b) '(c) '(c b a) '(a a b b c c c))
```

Context-Sensitive Grammars

- Writing tests using words that are not in L is ill-advised in this case
- The search space for finding a derivation may be visualized as a tree
- Consider trying to find a derivation for aa:

- The partially generated word down rightmost branches is always made longer and leads to an infinite search
- We are unable to write tests using words that are not in L

Context-Sensitive Grammars

- Writing tests using words that are not in `L` is ill-advised in this case
- The search space for finding a derivation may be visualized as a tree
- Consider trying to find a derivation for aa:

- The partially generated word down rightmost branches is always made longer and leads to an infinite search
- We are unable to write tests using words that are not in `L`
- `grammar-test` may easily be caught in an infinite recursion when used to test a `csg`
- Its use with `csgs` is disabled in `FSM`

Context-Sensitive Grammars

- To prove correctness think of the derivation process as follows:

```
while (contains-nonterminals? yield)
    apply-rule()
```

Context-Sensitive Grammars

- To prove correctness think of the derivation process as follows:

```
while (contains-nonterminals? yield)
    apply-rule()
```

- Develop a predicate for the loop invariant

Context-Sensitive Grammars

- To prove correctness think of the derivation process as follows:

```
while (contains-nonterminals? yield)
    apply-rule()
```

- Develop a predicate for the loop invariant
-
  - $|\mathtt{A}$ and $\mathtt{a}|=|\mathtt{B}$ and $\mathtt{b}|=|\mathtt{C}$ and $\mathtt{c}|$
  - $\mathtt{S}\in\mathtt{yield} \Rightarrow \mathtt{yield}$ ends with $\mathtt{S}$
  - $\mathtt{G}\in\mathtt{yield} \Rightarrow \mathtt{yield}$ ends with $\mathtt{Gc}^*$
  - $\mathtt{H}\in\mathtt{yield} \Rightarrow \mathtt{yield}$ ends with $\mathtt{Hb}^*\mathtt{c}^+$
  - $\mathtt{I}\in\mathtt{yield} \Rightarrow \mathtt{yield}$ ends with $\mathtt{Ia}^*\mathtt{b}^+\mathtt{c}^+$
  - $\mathtt{yield}\in\Sigma^* \Rightarrow \mathtt{yield}\in\mathtt{L}$

Context-Sensitive Grammars

- ```
  ;; (listof (N ∪ Σ) → Boolean
  ;; Purpose: Determine if loop invariant holds for the given yield
  (define (anbncn-inv yield)
    (and (equal-num-abc? yield) equal number A/a,B/b,C/c
         (implies (member 'S yield) (S-INV yield))
         (implies (member 'G yield) (G-INV yield))
         (implies (member 'H yield) (H-INV yield))
         (implies (member 'I yield) (I-INV yield))
         (implies (no-nt? yield)    (in-lang? yield))))
  ```
  ```

Context-Sensitive Grammars

- ;; (listof (N ∪ Σ) → Boolean
  ;; Purpose: Determine if loop invariant holds for the given yield
  (define (anbncn-inv yield)
    (and (equal-num-abc? yield) equal number A/a,B/b,C/c
         (implies (member 'S yield) (S-INV yield))
         (implies (member 'G yield) (G-INV yield))
         (implies (member 'H yield) (H-INV yield))
         (implies (member 'I yield) (I-INV yield))
         (implies (no-nt? yield)   (in-lang? yield))))
- (define (S-INV yield) (eq? (last yield) 'S))

Context-Sensitive Grammars

- ;; (listof (N ∪ Σ) → Boolean
  ;; Purpose: Determine if loop invariant holds for the given yield
  (define (anbncn-inv yield)
    (and (equal-num-abc? yield) equal number A/a,B/b,C/c
         (implies (member 'S yield) (S-INV yield))
         (implies (member 'G yield) (G-INV yield))
         (implies (member 'H yield) (H-INV yield))
         (implies (member 'I yield) (I-INV yield))
         (implies (no-nt? yield)   (in-lang? yield))))
- (define (S-INV yield) (eq? (last yield) 'S))
- (define (G-INV yield) Only c's right of G
    (= (count (lambda (x) (not (eq? 'c x)))
              (rest (dropf yield (lambda (x) (not (eq? 'G x))))))
       0))

Context-Sensitive Grammars

- ;; (listof (N ∪ Σ) → Boolean
  ;; Purpose: Determine if loop invariant holds for the given yield
  (define (anbncn-inv yield)
    (and (equal-num-abc? yield) equal number A/a,B/b,C/c
         (implies (member 'S yield) (S-INV yield))
         (implies (member 'G yield) (G-INV yield))
         (implies (member 'H yield) (H-INV yield))
         (implies (member 'I yield) (I-INV yield))
         (implies (no-nt? yield)   (in-lang? yield))))
- (define (S-INV yield) (eq? (last yield) 'S))
- (define (G-INV yield) Only c's right of G
    (= (count (lambda (x) (not (eq? 'c x)))
              (rest (dropf yield (lambda (x) (not (eq? 'G x))))))
       0))
- (define (H-INV yield) Only b's and c's right of H
    (let* ([bs-and-cs (rest (dropf yield
                                   (lambda (x) (not (eq? 'H x)))))]
           [cs (dropf bs-and-cs (lambda (x) (eq? x 'b)))])
      (andmap (lambda (x) (eq? x 'c)) cs)))

Context-Sensitive Grammars

- ```
  (define (I-INV yield)
      (let* ([as-bs-cs (rest (dropf yield (lambda (x) (not (eq? 'I x))
             [bs-and-cs (dropf as-bs-cs (lambda (x) (eq? 'a x)))]
             [cs (dropf bs-and-cs (lambda (x) (eq? x 'b)))])
        (andmap (lambda (x) (eq? 'c x)) cs)))
  ```
  ```

Context-Sensitive Grammars

- ```
  (define (I-INV yield)
      (let* ([as-bs-cs (rest (dropf yield (lambda (x) (not (eq? 'I x)
             [bs-and-cs (dropf as-bs-cs (lambda (x) (eq? 'a x)))]
             [cs (dropf bs-and-cs (lambda (x) (eq? x 'b)))])
        (andmap (lambda (x) (eq? 'c x)) cs)))
  ```
- ```
  (define (in-lang? yield)
      (let* ([as (takef yield (lambda (x) (eq? 'a x)))]
             [bs-and-cs (dropf yield (lambda (x) (eq? 'a x)))]
             [bs (takef bs-and-cs (lambda (x) (eq? 'b x)))]
             [cs (takef (dropf bs-and-cs (lambda (x) (eq? x 'b))) (lan
        (= (length as) (length bs) (length cs))))
  ```

Context-Sensitive Grammars

- ```
  (define (I-INV yield)
      (let* ([as-bs-cs (rest (dropf yield (lambda (x) (not (eq? 'I x)
             [bs-and-cs (dropf as-bs-cs (lambda (x) (eq? 'a x)))]
             [cs (dropf bs-and-cs (lambda (x) (eq? x 'b)))])
        (andmap (lambda (x) (eq? 'c x)) cs)))
  ```
- ```
  (define (in-lang? yield)
      (let* ([as (takef yield (lambda (x) (eq? 'a x)))]
             [bs-and-cs (dropf yield (lambda (x) (eq? 'a x)))]
             [bs (takef bs-and-cs (lambda (x) (eq? 'b x)))]
             [cs (takef (dropf bs-and-cs (lambda (x) (eq? x 'b))) (lan
        (= (length as) (length bs) (length cs))))
  ```
- ```
  (define (equal-num-abc? word)
      (= (+ (count (lambda (x) (eq? 'A x)) word)
            (count (lambda (x) (eq? 'a x)) word))
         (+ (count (lambda (x) (eq? 'B x)) word)
            (count (lambda (x) (eq? 'b x)) word))
         (+ (count (lambda (x) (eq? 'C x)) word)
            (count (lambda (x) (eq? 'c x)) word))))
  ```
  ```

Context-Sensitive Grammars

- ```
  (define (I-INV yield)
      (let* ([as-bs-cs (rest (dropf yield (lambda (x) (not (eq? 'I x)
             [bs-and-cs (dropf as-bs-cs (lambda (x) (eq? 'a x)))]
             [cs (dropf bs-and-cs (lambda (x) (eq? x 'b)))])
        (andmap (lambda (x) (eq? 'c x)) cs)))
  ```
- ```
  (define (in-lang? yield)
      (let* ([as (takef yield (lambda (x) (eq? 'a x)))]
             [bs-and-cs (dropf yield (lambda (x) (eq? 'a x)))]
             [bs (takef bs-and-cs (lambda (x) (eq? 'b x)))]
             [cs (takef (dropf bs-and-cs (lambda (x) (eq? x 'b))) (lan
        (= (length as) (length bs) (length cs))))
  ```
- ```
  (define (equal-num-abc? word)
      (= (+ (count (lambda (x) (eq? 'A x)) word)
            (count (lambda (x) (eq? 'a x)) word))
         (+ (count (lambda (x) (eq? 'B x)) word)
            (count (lambda (x) (eq? 'b x)) word))
         (+ (count (lambda (x) (eq? 'C x)) word)
            (count (lambda (x) (eq? 'c x)) word))))
  ```
- ```
  (define (no-nt? yield)
      (andmap (lambda (x) (member x (grammar-sigma anbncn))) yield))
  ```

Context-Sensitive Grammars

- Proof by induction on n=number of derivation steps

```
Base case:
When derivation starts, yield=S.
S-INV holds a S is the rightmost element of the yield) ∧
the number of As, as, Bs, bs, Cs, and cs are all  0 ∧
other implications are true (i.e., false ⇒ X is true)
```

Context-Sensitive Grammars

- Proof by induction on n=number of derivation steps

```
Base case:
When derivation starts, yield=S.
S-INV holds a S is the rightmost element of the yield) ∧
the number of As, as, Bs, bs, Cs, and cs are all  0 ∧
other implications are true (i.e., false ⇒ X is true)
```

- Inductive Step

```
Assume: INV holds for n=k
  Show: INV holds for n=k+1
```

Context-Sensitive Grammars

- Proof by induction on n=number of derivation steps

```
Base case:
When derivation starts, yield=S.
S-INV holds a S is the rightmost element of the yield) ∧
the number of As, as, Bs, bs, Cs, and cs are all  0 ∧
other implications are true (i.e., false ⇒ X is true)
```

- Inductive Step

```
Assume: INV holds for n=k
  Show: INV holds for n=k+1
```

- S → EMP

```
By inductive hypothesis INV holds
After applying this production rule:
the number of As, as, Bs, bs, Cs, and cs are all  0 ∧
(implies (no-nt? yield) (in-lang? yield)) holds ∧
other implications are true (i.e., false ⇒ X is true)
```

Context-Sensitive Grammars

- Proof by induction on n=number of derivation steps

```
Base case:
When derivation starts, yield=S.
S-INV holds a S is the rightmost element of the yield) ∧
the number of As, as, Bs, bs, Cs, and cs are all  0 ∧
other implications are true (i.e., false ⇒ X is true)
```

- Inductive Step

```
Assume: INV holds for n=k
  Show: INV holds for n=k+1
```

- S → EMP

```
By inductive hypothesis INV holds
After applying this production rule:
the number of As, as, Bs, bs, Cs, and cs are all  0 ∧
(implies (no-nt? yield) (in-lang? yield)) holds ∧
other implications are true (i.e., false ⇒ X is true)
```

- S → ABCS

```
By inductive hypothesis INV holds
After applying this production rule:
|A & a|=|B & b|=|C & c| ∧
S ends the yield ∧
other implications are true (i.e., false ⇒ X is true)
```

Context-Sensitive Grammars

- BA $\rightarrow$ AB, CB $\rightarrow$ BC, CA $\rightarrow$ AC

```
By inductive hypothesis INV holds
After applying any of these production rules:
INV holds because only the ordering of A, B, & C change
```

Context-Sensitive Grammars

- BA $\rightarrow$ AB, CB $\rightarrow$ BC, CA $\rightarrow$ AC

  By inductive hypothesis INV holds
  After applying any of these production rules:
  INV holds because only the ordering of A, B, & C change
- S $\rightarrow$ G

  By inductive hypothesis INV holds
  After applying this production rule:
  |A & a|=|B & b|=|C & c| $\wedge$
  Everything to the right of G is c $\wedge$
  other implications are true (i.e., false $\Rightarrow$ X is true)

Context-Sensitive Grammars

- $BA \rightarrow AB$, $CB \rightarrow BC$, $CA \rightarrow AC$

  By inductive hypothesis INV holds
  After applying any of these production rules:
  INV holds because only the ordering of A, B, & C change

- $S \rightarrow G$

  By inductive hypothesis INV holds
  After applying this production rule:
  |A & a|=|B & b|=|C & c| $\wedge$
  Everything to the right of G is c $\wedge$
  other implications are true (i.e., false $\Rightarrow$ X is true)

- $CG \rightarrow Gc$

  By inductive hypothesis INV holds
  After applying this production rule:
  |A & a|=|B & b|=|C & c| $\wedge$
  Everything to the right of G is c^+ $\wedge$
  other implications are true (i.e., false $\Rightarrow$ X is true)

Context-Sensitive Grammars

- $BA \rightarrow AB$, $CB \rightarrow BC$, $CA \rightarrow AC$

  By inductive hypothesis INV holds
  After applying any of these production rules:
  INV holds because only the ordering of A, B, & C change

- $S \rightarrow G$

  By inductive hypothesis INV holds
  After applying this production rule:
  $|A \ \& \ a| = |B \ \& \ b| = |C \ \& \ c| \ \wedge$
  Everything to the right of G is c $\wedge$
  other implications are true (i.e., false $\Rightarrow$ X is true)

- $CG \rightarrow Gc$

  By inductive hypothesis INV holds
  After applying this production rule:
  $|A \ \& \ a| = |B \ \& \ b| = |C \ \& \ c| \ \wedge$
  Everything to the right of G is c^+ $\wedge$
  other implications are true (i.e., false $\Rightarrow$ X is true)

- $BG \rightarrow BH$

  By inductive hypothesis INV holds
  After applying this production rule:
  $|A \ \& \ a| = |B \ \& \ b| = |C \ \& \ c| \ \wedge$
  Everything to the right of H is b^*c^+ $\wedge$
  other implications are true (i.e., false $\Rightarrow$ X is true)

Context-Sensitive Grammars

- BH $\rightarrow$ Hb

  By inductive hypothesis INV holds
  After applying this production rule:
  |A & a|=|B & b|=|C & c| $\wedge$
  Everything to the right of H is b^+c^+ $\wedge$
  other implications are true (i.e., false $\Rightarrow$ X is true)

Context-Sensitive Grammars

- BH $\rightarrow$ Hb

  By inductive hypothesis INV holds
  After applying this production rule:
  |A & a|=|B & b|=|C & c| $\wedge$
  Everything to the right of H is b^+c^+ $\wedge$
  other implications are true (i.e., false $\Rightarrow$ X is true)
- AH $\rightarrow$ AI

  By inductive hypothesis INV holds
  After applying this production rule:
  |A & a|=|B & b|=|C & c| $\wedge$
  Everything to the right of I is $a^*b^+c^+$ $\wedge$
  other implications are true (i.e., false $\Rightarrow$ X is true)

Context-Sensitive Grammars

- $BH \rightarrow Hb$

  By inductive hypothesis INV holds
  After applying this production rule:
  |A & a|=|B & b|=|C & c| $\wedge$
  Everything to the right of H is b^+c^+ $\wedge$
  other implications are true (i.e., false $\Rightarrow$ X is true)
- $AH \rightarrow AI$

  By inductive hypothesis INV holds
  After applying this production rule:
  |A & a|=|B & b|=|C & c| $\wedge$
  Everything to the right of I is $a^*b^+c^+$ $\wedge$
  other implications are true (i.e., false $\Rightarrow$ X is true)
- $AI \rightarrow Ia$

  By inductive hypothesis INV holds
  After applying this production rule:
  |A & a|=|B & b|=|C & c| $\wedge$
  Everything to the right of I is $a^+b^+c^+$ $\wedge$
  other implications are true (i.e., false $\Rightarrow$ X is true)

Context-Sensitive Grammars

- $BH \rightarrow Hb$

  By inductive hypothesis INV holds
  After applying this production rule:
  $|A \& a|=|B \& b|=|C \& c| \wedge$
  Everything to the right of H is $b^+c^+ \wedge$
  other implications are true (i.e., false $\Rightarrow$ X is true)
- $AH \rightarrow AI$

  By inductive hypothesis INV holds
  After applying this production rule:
  $|A \& a|=|B \& b|=|C \& c| \wedge$
  Everything to the right of I is $a^*b^+c^+ \wedge$
  other implications are true (i.e., false $\Rightarrow$ X is true)
- $AI \rightarrow Ia$

  By inductive hypothesis INV holds
  After applying this production rule:
  $|A \& a|=|B \& b|=|C \& c| \wedge$
  Everything to the right of I is $a^+b^+c^+ \wedge$
  other implications are true (i.e., false $\Rightarrow$ X is true)
- $I \rightarrow EMP$

  After applying this production rule:
  $|a|=|b|=|c| \wedge$
  yield is $a^+b^+c^+ \wedge$
  other implications are true (i.e., false $\Rightarrow$ X is true)

Context-Sensitive Grammars

- Proof that L=L(anbncn)

  w∈L ⇔ w∈L(anbncn)
  (⇒) Assume w∈L
  This means w=$a^n b^n c^n$. Given that INV always holds, there is a derivation such that S generates n ABC, rearranges ABCs to $A^n B^n C^n$, and generates w.

  (⇐)) Assume w∈L(anbncn)
  This means anbncn generated w. Since INV always holds w=$a^n b^n c^n$. Thus, w∈L.

Context-Sensitive Grammars

- Proof that L=L(anbncn)

  w∈L ⇔ w∈L(anbncn)
  (⇒) Assume w∈L
  This means $w=a^nb^nc^n$. Given that INV always holds, there is a derivation such that S generates n ABC, rearranges ABCs to $A^nB^nC^n$, and generates w.

  (⇐)) Assume w∈L(anbncn)
  This means anbncn generated w. Since INV always holds $w=a^nb^nc^n$. Thus, w∈L.

- w∉L ⇔ w∉L(anbncn)

  Contraposition

Context-Sensitive Grammars

DOUBLE BONUS QUIZ

- Design and implement a csg for $L = a^n b^n c^n d^n$. **Follow all the steps of the DR to get credit**.

Context-Sensitive Grammars

- Consider verifying valid adding expressions for numbers in unary notation:

  `L = {AbBbAB | A,B∈i*}`

- Valid arithmetic expressions:

  `iibibiii  bb  iiibiibiiiii`

Context-Sensitive Grammars

- Consider verifying valid adding expressions for numbers in unary notation:

  `L = {AbBbAB | A,B∈i*}`

- Valid arithmetic expressions:

  `iibibiii` `bb` `iiibiibiiiii`

- Observe that in `L` context matters
- An expression is valid only if it ends with a number of `i`s that is equal to the number of `i`s before the second `b`
- Suggests implementing a csg

Context-Sensitive Grammars

- Design Idea
- The grammar starts by generating `AbBbE`
- The `A` and `B` are used to nondeterministically generate the unary numbers in the sum
- For every `i` in these, `I`, a promise to generate an `i` for the result is generated

Context-Sensitive Grammars

- Design Idea
- The grammar starts by generating `AbBbE`
- The `A` and `B` are used to nondeterministically generate the unary numbers in the sum
- For every `i` in these, `I`, a promise to generate an `i` for the result is generated
- Every `I` may be bubbled to the right until it reaches `E`
- Upon reaching `E`, an `I` becomes an `i`
- Nondeterministically, `E` generates `EMP`.

Context-Sensitive Grammars

- A descriptive name for the grammar is `ADD-CSG`
- The alphabet in {`b i`}.

Context-Sensitive Grammars

- A descriptive name for the grammar is `ADD-CSG`
- The alphabet in {`b i`}.
- `S` generates three unary numbers separated by `b`s such that the sum of the first two numbers is equal to the third numbers:

```
;;  S: generates words in i^nbi^mbi^ni^m
```

Context-Sensitive Grammars

- A descriptive name for the grammar is `ADD-CSG`
- The alphabet in {`b i`}.
- `S` generates three unary numbers separated by `b`s such that the sum of the first two numbers is equal to the third numbers:

```
;;  S: generates words in i^nbi^mbi^ni^m
```

- A nonterminal, `A`, is needed to generate a unary number:

```
;; A: generates words in i^* and, for every i generated,
;;    a promise to generate a matching i for the result
```

Context-Sensitive Grammars

- A descriptive name for the grammar is `ADD-CSG`
- The alphabet in {`b i`}.
- `S` generates three unary numbers separated by `b`s such that the sum of the first two numbers is equal to the third numbers:

```
;;  S: generates words in i^nbi^mbi^ni^m
```

- A nonterminal, `A`, is needed to generate a unary number:

```
;; A: generates words in i^* and, for every i generated,
;;    a promise to generate a matching i for the result
```

- Two nonterminals, `I` and `E`, are needed to generate the sum's result
- `I` represents the promise to generate an `i` in, `IE`, the proper context:

```
;;  I: generates an i for the result in the context IE
;;  E: generates zero i or generates one i for the result
;;     in the context IE
```

Context-Sensitive Grammars

- Production Rules
- S must generate three numbers separated by bs such that the third number is the sum of the first two numbers:

```
(S ,ARROW AbAbE)
```

Context-Sensitive Grammars

- Production Rules
- `S` must generate three numbers separated by `b`s such that the third number is the sum of the first two numbers:

```
(S ,ARROW AbAbE)
```

- The nonterminal `A` must generate an arbitrary number of `i`s and a matching promise for each to generate an `i` for the number representing the result:

```
(A ,ARROW ,EMP)
(A ,ARROW iIA)
```

Context-Sensitive Grammars

- Production Rules
- `S` must generate three numbers separated by `b`s such that the third number is the sum of the first two numbers:

```
(S ,ARROW AbAbE)
```

- The nonterminal `A` must generate an arbitrary number of `i`s and a matching promise for each to generate an `i` for the number representing the result:

```
(A ,ARROW ,EMP)
(A ,ARROW iIA)
```

- To generate `i`s for the result number, the `I`s must be bubbled to the end of the partially generated word after the second `b` and before `E`:

```
(Ii ,ARROW iI)
(Ib ,ARROW bI)
```

Context-Sensitive Grammars

- Production Rules
- S must generate three numbers separated by bs such that the third number is the sum of the first two numbers:

```
(S ,ARROW AbAbE)
```

- The nonterminal A must generate an arbitrary number of is and a matching promise for each to generate an i for the number representing the result:

```
(A ,ARROW ,EMP)
(A ,ARROW iIA)
```

- To generate is for the result number, the Is must be bubbled to the end of the partially generated word after the second b and before E:

```
(Ii ,ARROW iI)
(Ib ,ARROW bI)
```

- E nondeterministically generates 0 is or an i in the context IE:

```
(IE ,ARROW Ei)
(E ,ARROW ,EMP)
```

Context-Sensitive Grammars

- Tests

```
;; Tests
(check-equal? (grammar-derive ADD-CSG2 '(b b))
              '(S -> AbAbE -> AbAb -> Abb -> bb))
(check-equal? (last (grammar-derive ADD-CSG2 '(b i b i)))
              'bibi)
(check-equal? (last (grammar-derive ADD-CSG2 '(i b b i)))
              'ibbi)
(check-equal?
  (last (grammar-derive ADD-CSG2 '(i i b i i b i i i i)))
  'iibiibiiii)
(check-equal?
  (last (grammar-derive ADD-CSG2 '(i i b i i i b i i i i i)))
  'iibiiibiiiii)
```

- The last test may take long to evaluate

Context-Sensitive Grammars

- Equivalence of `tms` and `csgs`
- If a derivation is found by the grammar simulated, the `tm` halts
- On the other hand, if a derivation is not found then the `tm` never halts
- The simulating `tm` can only semi-decide `L`.

Context-Sensitive Grammars

- Equivalence of `tms` and `csgs`
- If a derivation is found by the grammar simulated, the `tm` halts
- On the other hand, if a derivation is not found then the `tm` never halts
- The simulating `tm` can only semi-decide `L`.
- Given a `tm`, `M`, that semi-decides `L`, how is a grammar to generate words in `L` be constructed?
- Intuitively, the grammar mimics any computation in reverse order
- We shall assume that `M` always erases its tape before halting: `(Y 1 `(,LM ,BLANK))`

Context-Sensitive Grammars

- Equivalence of `tms` and `csgs`
- If a derivation is found by the grammar simulated, the `tm` halts
- On the other hand, if a derivation is not found then the `tm` never halts
- The simulating `tm` can only semi-decide `L`.
- Given a `tm`, `M`, that semi-decides `L`, how is a grammar to generate words in `L` be constructed?
- Intuitively, the grammar mimics any computation in reverse order
- We shall assume that `M` always erases its tape before halting: `(Y 1 `(,LM ,BLANK))`
- We shall only *sketch* the proof that `L` is generated by a `csg` if and only if `L` is semi-decided by a `tm`
- That is, we shall not implement the constructors
-
  - implementation choices get messy fairly quickly and provide little insight into the equivalence of `csgs` and `tms`
  - The proof involves creating an `mttm` and may require an unreasonably rich alphabet

Context-Sensitive Grammars

Theorem

L is generated by a `csg` $\Leftrightarrow$ *L is semi-decided by a* `tm`.

Proof.

($\Rightarrow$) Assume L is generated by a `csg`.
Let G = (make-csg N Σ R S) be the grammar that generates L. We shall design a nondeterministic 3-tape `mttm` Tape 0 contains, w, the input and is never mutated. Tape 1 is used to reconstruct a derivation for w starting with S. Therefore, M starts by writing S on tape 1. Tape 2 contains R and is never mutated.

M operates in steps as follows:

$\square$

Context-Sensitive Grammars

Theorem

L is generated by a `csg` $\Leftrightarrow$ *L is semi-decided by a* `tm`.

Proof.

($\Rightarrow$) Assume L is generated by a `csg`.
Let G = (make-csg N Σ R S) be the grammar that generates L. We shall design a nondeterministic 3-tape `mttm` Tape 0 contains, w, the input and is never mutated. Tape 1 is used to reconstruct a derivation for w starting with S. Therefore, M starts by writing S on tape 1. Tape 2 contains R and is never mutated.

M operates in steps as follows:

1. M nondeterministically picks a rule, y $\rightarrow$ x, to apply from tape 2 or chooses to match the contents of tape 2 with w on tape 0.

$\square$

Context-Sensitive Grammars

Theorem

L is generated by a `csg` $\Leftrightarrow$ *L is semi-decided by a* `tm`.

Proof.

($\Rightarrow$) Assume L is generated by a `csg`.
Let G = (make-csg N Σ R S) be the grammar that generates L. We shall design a nondeterministic 3-tape `mttm` Tape 0 contains, w, the input and is never mutated. Tape 1 is used to reconstruct a derivation for w starting with S. Therefore, M starts by writing S on tape 1. Tape 2 contains R and is never mutated.

M operates in steps as follows:

1. M nondeterministically picks a rule, y $\rightarrow$ x, to apply from tape 2 or chooses to match the contents of tape 2 with w on tape 0.
2. If a rule is chosen:
   1. M scans tape 1 and nondeterministically stops at a symbol.
   2. M matches y and tape 1 is mutated to replace y with x. Tape 1's contents is shifted as necessary to fit x.

□

Context-Sensitive Grammars

Theorem

L is generated by a `csg` $\Leftrightarrow$ *L is semi-decided by a* `tm`.

Proof.

($\Rightarrow$) Assume L is generated by a `csg`.
Let G = (make-csg N Σ R S) be the grammar that generates L. We shall design a nondeterministic 3-tape `mttm` Tape 0 contains, w, the input and is never mutated. Tape 1 is used to reconstruct a derivation for w starting with S. Therefore, M starts by writing S on tape 1. Tape 2 contains R and is never mutated.

M operates in steps as follows:

1. M nondeterministically picks a rule, y $\rightarrow$ x, to apply from tape 2 or chooses to match the contents of tape 2 with w on tape 0.
2. If a rule is chosen:
   1. M scans tape 1 and nondeterministically stops at a symbol.
   2. M matches y and tape 1 is mutated to replace y with x. Tape 1's contents is shifted as necessary to fit x.
3. If a rule is not chosen M matches w and accepts or it runs forever
4. It is not difficult to see that M semi-decides L

$\square$

Context-Sensitive Grammars

Theorem

L is generated by a `csg` $\Leftrightarrow$ *L is semi-decided by a* `tm`.

Proof.

($\Leftarrow$) Assume L is semi-decided by a `tm`.
We shall construct a `csg`, G, that generates L.

Let G = (make-csg N Σ' R' S') such that N∩Σ=∅. The components are described as follows:

Context-Sensitive Grammars

Theorem

L is generated by a `csg` $\Leftrightarrow$ *L is semi-decided by a* `tm`.

Proof.

($\Leftarrow$) Assume L is semi-decided by a `tm`.
We shall construct a `csg`, G, that generates L.

Let G = (make-csg N Σ' R' S') such that N∩Σ=∅. The components are described as follows:

1. N contains K, a start symbol S', a right-end marker `RM`, and nonterminals to represent an M configuration. The configuration `(q i (,LM i ua¡v)) is represented by the symbol `LMua¡qvRM`

Context-Sensitive Grammars

Theorem

L is generated by a `csg` $\Leftrightarrow$ *L is semi-decided by a* `tm`.

Proof.

($\Leftarrow$) Assume L is semi-decided by a `tm`.
We shall construct a `csg`, G, that generates L.

Let G = (make-csg N Σ' R' S') such that N∩Σ=∅. The components are described as follows:

1. N contains K, a start symbol S', a right-end marker `RM`, and nonterminals to represent an M configuration. The configuration `(q i (,LM i ua;v)) is represented by the symbol `LMua;qvRM`
2. Σ' = Σ

Context-Sensitive Grammars

Theorem

L is generated by a `csg` $\Leftrightarrow$ *L is semi-decided by a* `tm`.

Proof.

($\Leftarrow$) Assume L is semi-decided by a `tm`.
We shall construct a csg, G, that generates L.

Let G = (make-csg N Σ' R' S') such that N∩Σ=∅. The components are described as follows:

1. N contains K, a start symbol S', a right-end marker `RM`, and nonterminals to represent an M configuration. The configuration `(q i (,LM i ua;v)) is represented by the symbol `LMua;qvRM`
2. Σ' = Σ
3. ∀k∈K and ∀a∈Σ, R' has rules built as follows:

```
1.If ((Q a) (P b))∈R, where b∈Σ, then R has: Pb → aQ
2. If ((Q a) (P RIGHT))∈R then R' has the following rules:
 a. ∀b∈Σ R' has: abP → aQb
 b. To reverse extending the touched part of the tape to the right, R' has:
   aBLANKP → aQRM
3. If ((Q a) (P LEFT))∈R and a≠BLANK then R' has: Pa → aQ
4. If ((Q BLANK) (P LEFT)) the R' has the following rules:
 a. ∀ b∈Σ R' has: Pab → aQb
 b. To reverse erasing blanks, R' has: PRM → BLANKQ
```

Context-Sensitive Grammars

Theorem

L is generated by a `csg` $\Leftrightarrow$ *L is semi-decided by a* `tm`.

Proof.

($\Leftarrow$) Assume L is semi-decided by a `tm`.
We shall construct a csg, G, that generates L.

Let G = (make-csg N Σ' R' S') such that N∩Σ=∅. The components are described as follows:

1. N contains K, a start symbol S', a right-end marker `RM`, and nonterminals to represent an M configuration. The configuration `(q i (,LM i ua¡v)) is represented by the symbol `LMua¡qvRM`
2. Σ' = Σ
3. ∀k∈K and ∀a∈Σ, R' has rules built as follows:

```
1.If ((Q a) (P b))∈R, where b∈Σ, then R has: Pb → aQ
2. If ((Q a) (P RIGHT))∈R then R' has the following rules:
 a. ∀b∈Σ R' has: abP → aQb
 b. To reverse extending the touched part of the tape to the right, R' has:
   aBLANKP → aQRM
3. If ((Q a) (P LEFT))∈R and a≠BLANK then R' has: Pa → aQ
4. If ((Q BLANK) (P LEFT)) the R' has the following rules:
  a. ∀ b∈Σ R' has: Pab → aQb
  b. To reverse erasing blanks, R' has: PRM → BLANKQ
```

4. To start the derivation, R' has: S → `BLANKYRM` (i.e., the derivation starts where M halts)

Context-Sensitive Grammars

Theorem

L is generated by a `csg` $\Leftrightarrow$ *L is semi-decided by a* `tm`.

Proof.

($\Leftarrow$) Assume L is semi-decided by a `tm`.
We shall construct a csg, G, that generates L.

Let G = (make-csg N Σ' R' S') such that N∩Σ=∅. The components are described as follows:

1. N contains K, a start symbol S', a right-end marker `RM`, and nonterminals to represent an M configuration. The configuration `(q i (,LM i ua;v)) is represented by the symbol `LMua;qvRM`
2. Σ' = Σ
3. ∀k∈K and ∀a∈Σ, R' has rules built as follows:

```
1.If ((Q a) (P b))∈R, where b∈Σ, then R has: Pb → aQ
2. If ((Q a) (P RIGHT))∈R then R' has the following rules:
 a. ∀b∈Σ R' has: abP → aQb
 b. To reverse extending the touched part of the tape to the right, R' has:
   aBLANKP → aQRM
3. If ((Q a) (P LEFT))∈R and a≠BLANK then R' has: Pa → aQ
4. If ((Q BLANK) (P LEFT)) the R' has the following rules:
 a. ∀ b∈Σ R' has: Pab → aQb
 b. To reverse erasing blanks, R' has: PRM → BLANKQ
```

4. To start the derivation, R' has: S → `BLANKYRM` (i.e., the derivation starts where M halts)
5. To erase the start state when the derivation is completed, R' has: `LMBLANKs` → EMP

Context-Sensitive Grammars

Theorem

L is generated by a `csg` $\Leftrightarrow$ *L is semi-decided by a* `tm`.

Proof.

($\Leftarrow$) Assume L is semi-decided by a `tm`.
We shall construct a csg, G, that generates L.

Let G = (make-csg N Σ' R' S') such that N∩Σ=∅. The components are described as follows:

1. N contains K, a start symbol S', a right-end marker `RM`, and nonterminals to represent an M configuration. The configuration `(q i (,LM i ua¡v)) is represented by the symbol `LMua¡qvRM`
2. Σ' = Σ
3. ∀k∈K and ∀a∈Σ, R' has rules built as follows:

```
1.If ((Q a) (P b))∈R, where b∈Σ, then R has: Pb → aQ
2. If ((Q a) (P RIGHT))∈R then R' has the following rules:
 a. ∀b∈Σ R' has: abP → aQb
 b. To reverse extending the touched part of the tape to the right, R' has:
   aBLANKP → aQRM
3. If ((Q a) (P LEFT))∈R and a≠BLANK then R' has: Pa → aQ
4. If ((Q BLANK) (P LEFT)) the R' has the following rules:
 a. ∀ b∈Σ R' has: Pab → aQb
 b. To reverse erasing blanks, R' has: PRM → BLANKQ
```

4. To start the derivation, R' has: S → `BLANKYRM` (i.e., the derivation starts where M halts)
5. To erase the start state when the derivation is completed, R' has: `LMBLANKs` → EMP
6. To erase the right-end marker when the derivation is done, R' has: RM → EMP
7. It is not difficult to see that G only generates words that are accepted by M

Context-Sensitive Grammars

- HOMEWORK: 2–4

Church-Turing Thesis and Undecidability

- What is an algorithm?

Church-Turing Thesis and Undecidability

- What is an algorithm?
- An algorithm is a Turing machine

Church-Turing Thesis and Undecidability

- What is an algorithm?
- An algorithm is a Turing machine
- Turing machines that decide a language or that compute a function are algorithms
- Turing machines that may not halt are not considered algorithms

Church-Turing Thesis and Undecidability

- What is an algorithm?
- An algorithm is a Turing machine
- Turing machines that decide a language or that compute a function are algorithms
- Turing machines that may not halt are not considered algorithms
- *A Turing machine that halts on all inputs is the formal notion of an algorithm*
- This principle is known as the *Church-Turing thesis*

Church-Turing Thesis and Undecidability

- What is an algorithm?
- An algorithm is a Turing machine
- Turing machines that decide a language or that compute a function are algorithms
- Turing machines that may not halt are not considered algorithms
- *A Turing machine that halts on all inputs is the formal notion of an algorithm*
- This principle is known as the *Church-Turing thesis*
- As exciting as having a formal definition of an algorithm is, why else should we care about reaching this intellectual milestone?

Church-Turing Thesis and Undecidability

- What is an algorithm?
- An algorithm is a Turing machine
- Turing machines that decide a language or that compute a function are algorithms
- Turing machines that may not halt are not considered algorithms
- *A Turing machine that halts on all inputs is the formal notion of an algorithm*
- This principle is known as the *Church-Turing thesis*
- As exciting as having a formal definition of an algorithm is, why else should we care about reaching this intellectual milestone?
- Opens the door to proving that there are computational problems for which a solution does not exist
- A problem for which a solution does not exists is known as an *undecidable* or *unsolvable* problem

Church-Turing Thesis and Undecidability

- What is an algorithm?
- An algorithm is a Turing machine
- Turing machines that decide a language or that compute a function are algorithms
- Turing machines that may not halt are not considered algorithms
- *A Turing machine that halts on all inputs is the formal notion of an algorithm*
- This principle is known as the *Church-Turing thesis*
- As exciting as having a formal definition of an algorithm is, why else should we care about reaching this intellectual milestone?
- Opens the door to proving that there are computational problems for which a solution does not exist
- A problem for which a solution does not exists is known as an *undecidable* or *unsolvable* problem
- Pondering the limitations of language representations may not be an every day primary concern in industrial Computer Science.
- Even industrial computer scientists may face an unsolvable problem and then it becomes important to be able to recognize such problems and to prove they are undecidable

Church-Turing Thesis and Undecidability

- What is an algorithm?
- An algorithm is a Turing machine
- Turing machines that decide a language or that compute a function are algorithms
- Turing machines that may not halt are not considered algorithms
- *A Turing machine that halts on all inputs is the formal notion of an algorithm*
- This principle is known as the *Church-Turing thesis*
- As exciting as having a formal definition of an algorithm is, why else should we care about reaching this intellectual milestone?
- Opens the door to proving that there are computational problems for which a solution does not exist
- A problem for which a solution does not exists is known as an *undecidable* or *unsolvable* problem
- Pondering the limitations of language representations may not be an every day primary concern in industrial Computer Science.
- Even industrial computer scientists may face an unsolvable problem and then it becomes important to be able to recognize such problems and to prove they are undecidable
- For instance, would it not be wonderful to have a program that analyze a program we write and tells us if our program goes or does not go into an infinite recursion or an infinite loop?
- Can such a program be written?

Church-Turing Thesis and Undecidability

- The Halting Problem

```
;; program value → Boolean
;; Purpose: Determine if the given program halts on
;;          the given input
(define (halts? a-prog an-input) ...)
```

Church-Turing Thesis and Undecidability

- The Halting Problem

```
;; program value → Boolean
;; Purpose: Determine if the given program halts on
;;          the given input
(define (halts? a-prog an-input) ...)
```

- Let us assume that `halts?` can be implemented
- Use it to write a predicate that takes as input a program, `p`, and that goes into an infinite recursion if `(halts? p p)` returns true. Otherwise, it returns a false

Church-Turing Thesis and Undecidability

- The Halting Problem

```
;; program value → Boolean
;; Purpose: Determine if the given program halts on
;;          the given input
(define (halts? a-prog an-input) ...)
```

- Let us assume that `halts?` can be implemented
- Use it to write a predicate that takes as input a program, `p`, and that goes into an infinite recursion if `(halts? p p)` returns true. Otherwise, it returns a false
- This ought to be reminiscent of a diagonalization proof
- In essence, it is asking if `p` is not related to itself in terms of `halt?`

```
;; program → Boolean
;; Purpose: Return #f if the given program does not
;;          halt on itself
(define (p-halts-on-p? p)
  (if (halts? p p)
      (p-halts-on-p? p)
      #f))
```

- Why is `p-halts-on-p?` interesting?

Church-Turing Thesis and Undecidability

- The Halting Problem

```
;; program value → Boolean
;; Purpose: Determine if the given program halts on
;;          the given input
(define (halts? a-prog an-input) ...)
```

- Let us assume that `halts?` can be implemented
- Use it to write a predicate that takes as input a program, `p`, and that goes into an infinite recursion if `(halts? p p)` returns true. Otherwise, it returns a false
- This ought to be reminiscent of a diagonalization proof
- In essence, it is asking if `p` is not related to itself in terms of `halt?`

```
;; program → Boolean
;; Purpose: Return #f if the given program does not
;;          halt on itself
(define (p-halts-on-p? p)
  (if (halts? p p)
      (p-halts-on-p? p)
      #f))
```

- Why is `p-halts-on-p?` interesting?
- Consider calling `p-halts-on-p?` with `p-halts-on-p?` as input:

```
(p-halts-on-p? p-halts-on-p?)
```

- When does `(p-halts-on-p? p-halts-on-p?)` return `#f` and halt?

Church-Turing Thesis and Undecidability

- The Halting Problem

```
;; program value → Boolean
;; Purpose: Determine if the given program halts on
;;          the given input
(define (halts? a-prog an-input) ...)
```

- Let us assume that `halts?` can be implemented
- Use it to write a predicate that takes as input a program, `p`, and that goes into an infinite recursion if `(halts? p p)` returns true. Otherwise, it returns a false
- This ought to be reminiscent of a diagonalization proof
- In essence, it is asking if `p` is not related to itself in terms of `halt?`

```
;; program → Boolean
;; Purpose: Return #f if the given program does not
;;          halt on itself
(define (p-halts-on-p? p)
  (if (halts? p p)
      (p-halts-on-p? p)
      #f))
```

- Why is `p-halts-on-p?` interesting?
- Consider calling `p-halts-on-p?` with `p-halts-on-p?` as input:

```
(p-halts-on-p? p-halts-on-p?)
```

- When does `(p-halts-on-p? p-halts-on-p?)` return `#f` and halt?
- It does so when `(p-halts-on-p? p-halts-on-p?)` does not halt

Church-Turing Thesis and Undecidability

- The Halting Problem

```
;; program value → Boolean
;; Purpose: Determine if the given program halts on
;;          the given input
(define (halts? a-prog an-input) ...)
```

- Let us assume that `halts?` can be implemented
- Use it to write a predicate that takes as input a program, `p`, and that goes into an infinite recursion if `(halts? p p)` returns true. Otherwise, it returns a false
- This ought to be reminiscent of a diagonalization proof
- In essence, it is asking if `p` is not related to itself in terms of `halt?`

```
;; program → Boolean
;; Purpose: Return #f if the given program does not
;;          halt on itself
(define (p-halts-on-p? p)
  (if (halts? p p)
      (p-halts-on-p? p)
      #f))
```

- Why is `p-halts-on-p?` interesting?
- Consider calling `p-halts-on-p?` with `p-halts-on-p?` as input:

```
(p-halts-on-p? p-halts-on-p?)
```

- When does `(p-halts-on-p? p-halts-on-p?)` return `#f` and halt?
- It does so when `(p-halts-on-p? p-halts-on-p?)` does not halt
- This is clearly a contradiction and we are forced to conclude that our initial assumption is wrong
- The predicate `halts?` cannot be implemented
- The Halting Problem is undecidable

Church-Turing Thesis and Undecidability

- Noteworthy that our discussion started by asking you to think about implementing a program in your favorite programming language
- This programming language may be `FSM`

Church-Turing Thesis and Undecidability

- Noteworthy that our discussion started by asking you to think about implementing a program in your favorite programming language
- This programming language may be `FSM`
- The problem becomes to implement a Turing machine to determine if on a given input a given Turing machine halts
- Such a machine may be encoded as a `ctmand` is a language recognizer for:

  `L = {(M w) | M=(make-tm K Σ R S Y)` $\wedge$ `w∈Σ*` $\wedge$ M halts on w}
- Based on the argument above we can formally prove that `L` is undecidable.

Church-Turing Thesis and Undecidability

Theorem

The halting problem is undecidable.

Proof.

Church-Turing Thesis and Undecidability

Theorem

The halting problem is undecidable.

Proof.

- Assume L is decidable. This means that there is `M`, that decides `L`
- Consider a `tm`, `N`, that decides `L(N)`: PRE: tape = `(,LM BLANK w) ∧ head position = 1

Church-Turing Thesis and Undecidability

Theorem

The halting problem is undecidable.

Proof.

- Assume L is decidable. This means that there is `M`, that decides `L`
- Consider a `tm`, `N`, that decides `L(N)`: PRE: tape = `(,LM BLANK w) ∧ head position = 1
- Easily edit `N` to create `R` that semidecides `L(N)`
- We can build a `ctm` that decides `L(N)` using `M` as follows:

```
1. Transform the input tape from `(,LM BLANK w) to `(,LM R ,BLANK w)
2. Run M on the transformed input tape
```

- By assumption, this `ctm` returns `'accept` if `R` halts on `w` and `'reject` otherwise.

Church-Turing Thesis and Undecidability

Theorem

The halting problem is undecidable.

Proof.

- Assume L is decidable. This means that there is `M`, that decides `L`
- Consider a `tm`, `N`, that decides `L(N)`: PRE: tape = `(,LM BLANK w) ∧ head position = 1
- Easily edit `N` to create `R` that semidecides `L(N)`
- We can build a `ctm` that decides `L(N)` using `M` as follows:

```
1. Transform the input tape from `(,LM BLANK w) to `(,LM R ,BLANK w)
2. Run M on the transformed input tape
```

- By assumption, this `ctm` returns `'accept` if `R` halts on `w` and `'reject` otherwise.
- Consider the following language:

```
H = {M | M is a tm encoding that halts when given itself as input}
```

Church-Turing Thesis and Undecidability

Theorem

The halting problem is undecidable.

Proof.

- Assume L is decidable. This means that there is `M`, that decides `L`
- Consider a `tm`, `N`, that decides `L(N)`: PRE: tape = `(,LM BLANK w) ∧ head position = 1
- Easily edit `N` to create `R` that semidecides `L(N)`
- We can build a `ctm` that decides `L(N)` using `M` as follows:

```
1. Transform the input tape from `(,LM BLANK w) to `(,LM R ,BLANK w)
2. Run M on the transformed input tape
```

- By assumption, this `ctm` returns `'accept` if `R` halts on `w` and `'reject` otherwise.
- Consider the following language:

```
H = {M | M is a tm encoding that halts when given itself as input}
```

- The complement of `H` is:

```
Ĥ = {w|w not a tm encoding ∨ w does not halt when given itself as input}
```

Church-Turing Thesis and Undecidability

Theorem

The halting problem is undecidable.

Proof.

- Assume L is decidable. This means that there is `M`, that decides `L`
- Consider a `tm`, `N`, that decides `L(N)`: PRE: tape = `(,LM BLANK w) ∧ head position = 1
- Easily edit `N` to create `R` that semidecides `L(N)`
- We can build a `ctm` that decides `L(N)` using `M` as follows:

```
1. Transform the input tape from `(,LM BLANK w) to `(,LM R ,BLANK w)
2. Run M on the transformed input tape
```

- By assumption, this `ctm` returns `'accept` if `R` halts on `w` and `'reject` otherwise.
- Consider the following language:

```
H = {M | M is a tm encoding that halts when given itself as input}
```

- The complement of `H` is:

```
Ĥ = {w|w not a tm encoding ∨ w does not halt when given itself as input}
```

- `Ĥ` is not decidable. Let `P` be a `tm` encoding that semidecides `Ĥ`

Church-Turing Thesis and Undecidability

Theorem

The halting problem is undecidable.

Proof.

- Assume L is decidable. This means that there is `M`, that decides `L`
- Consider a `tm`, `N`, that decides `L(N)`: PRE: tape = `(,LM BLANK w) ∧ head position = 1
- Easily edit `N` to create `R` that semidecides `L(N)`
- We can build a `ctm` that decides `L(N)` using `M` as follows:

```
1. Transform the input tape from `(,LM BLANK w) to `(,LM R ,BLANK w)
2. Run M on the transformed input tape
```

- By assumption, this `ctm` returns `'accept` if `R` halts on `w` and `'reject` otherwise.
- Consider the following language:

```
H = {M | M is a tm encoding that halts when given itself as input}
```

- The complement of `H` is:

```
Ĥ = {w|w not a tm encoding ∨ w does not halt when given itself as input}
```

- `Ĥ` is not decidable. Let `P` be a `tm` encoding that semidecides `Ĥ`
- Is `P` is in `Ĥ`?

Church-Turing Thesis and Undecidability

Theorem

The halting problem is undecidable.

Proof.

- Assume L is decidable. This means that there is `M`, that decides `L`
- Consider a `tm`, `N`, that decides `L(N)`: PRE: tape = `(,LM BLANK w) ∧ head position = 1
- Easily edit `N` to create `R` that semidecides `L(N)`
- We can build a `ctm` that decides `L(N)` using `M` as follows:

```
1. Transform the input tape from `(,LM BLANK w) to `(,LM R ,BLANK w)
2. Run M on the transformed input tape
```

- By assumption, this `ctm` returns `'accept` if `R` halts on `w` and `'reject` otherwise.
- Consider the following language:

```
H = {M | M is a tm encoding that halts when given itself as input}
```

- The complement of `H` is:

```
Ĥ = {w|w not a tm encoding ∨ w does not halt when given itself as input}
```

- `Ĥ` is not decidable. Let `P` be a `tm` encoding that semidecides `Ĥ`
- Is `P` is in `Ĥ`?
- `P∈Ĥ` if and only if `P` halts on itself: `P` does not halt on itself if and only if `P` halts on itself

Church-Turing Thesis and Undecidability

Theorem

The halting problem is undecidable.

Proof.

- Assume L is decidable. This means that there is `M`, that decides `L`
- Consider a `tm`, `N`, that decides `L(N)`: PRE: tape = `(,LM BLANK w) ∧ head position = 1
- Easily edit `N` to create `R` that semidecides `L(N)`
- We can build a `ctm` that decides `L(N)` using `M` as follows:

```
1. Transform the input tape from `(,LM BLANK w) to `(,LM R ,BLANK w)
2. Run M on the transformed input tape
```

- By assumption, this `ctm` returns `'accept` if `R` halts on `w` and `'reject` otherwise.
- Consider the following language:

```
H = {M | M is a tm encoding that halts when given itself as input}
```

- The complement of `H` is:

```
Ĥ = {w|w not a tm encoding ∨ w does not halt when given itself as input}
```

- `Ĥ` is not decidable. Let `P` be a `tm` encoding that semidecides `Ĥ`
- Is `P` is in `Ĥ`?
- `P`∈`Ĥ` if and only if `P` halts on itself: `P` does not halt on itself if and only if `P` halts on itself
- If `Ĥ` is not semi-decidable then it is also not decidable
- Thus, our assumption that `L` is decidable is wrong and `L` must be undecidable.

Church-Turing Thesis and Undecidability

- HOMEWORK: 1–3

Church-Turing Thesis and Undecidability

- From the undecidability of the halting problem the undecidability of many other problems follow

Church-Turing Thesis and Undecidability

- From the undecidability of the halting problem the undecidability of many other problems follow
- A *reduction proof* establishes that an undecidable language/problem, `L`, may be decided if some different language/problem, `L1`, is decidable
- That is, the `tm` that decides `L1` may be used to build a `tm` that decides `L`, which is a contradiction

Church-Turing Thesis and Undecidability

- From the undecidability of the halting problem the undecidability of many other problems follow
- A *reduction proof* establishes that an undecidable language/problem, `L`, may be decided if some different language/problem, `L1`, is decidable
- That is, the `tm` that decides `L1` may be used to build a `tm` that decides `L`, which is a contradiction
- It is important to note the direction of the reduction

Church-Turing Thesis and Undecidability

- From the undecidability of the halting problem the undecidability of many other problems follow
- A *reduction proof* establishes that an undecidable language/problem, `L`, may be decided if some different language/problem, `L1`, is decidable
- That is, the `tm` that decides `L1` may be used to build a `tm` that decides `L`, which is a contradiction
- It is important to note the direction of the reduction

Church-Turing Thesis and Undecidability

- From the undecidability of the halting problem the undecidability of many other problems follow
- A *reduction proof* establishes that an undecidable language/problem, `L`, may be decided if some different language/problem, `L1`, is decidable
- That is, the `tm` that decides `L1` may be used to build a `tm` that decides `L`, which is a contradiction
- It is important to note the direction of the reduction

Church-Turing Thesis and Undecidability

- More formally, let R and S be two problems. A reduction from R to S is a transformation function, τ, such that $x \in R \Leftrightarrow \tau(x) \in S$ and $x \notin R \Leftrightarrow \tau(x) \notin S$

Church-Turing Thesis and Undecidability

- More formally, let R and S be two problems. A reduction from R to S is a transformation function, τ, such that $x \in R \Leftrightarrow \tau(x) \in S$ and $x \notin R \Leftrightarrow \tau(x) \notin S$
- You may think of τ as a function that is computable by a tm that transforms a subset of inputs for R to inputs for S in such a manner that the answer returned by S means that we know the answer that R ought to return

Church-Turing Thesis and Undecidability

Theorem

R is undecidable $\wedge \exists\ \tau$ = a reduction function from R to $S \Rightarrow S$ is undecidable.

-

Church-Turing Thesis and Undecidability

Theorem

R is undecidable $\wedge \exists\ \tau =$ *a reduction function from R to S* $\Rightarrow$ *S is undecidable.*

-

Proof.

Assume M_S decides S and T computes τ.
The `ctm` T M_S decides R. This is a contradiction. Therefore, S is undecidable.

□

-

Church-Turing Thesis and Undecidability

Theorem

R is undecidable $\wedge \exists\ \tau =$ *a reduction function from R to S* $\Rightarrow$ *S is undecidable.*

-

Proof.

Assume M_S decides S and T computes τ.
The `ctm` T M_S decides R. This is a contradiction. Therefore, S is undecidable. □

-
- Alan Turing penned one of the great intellectual achievements of the $20^{\mathtt{th}}$ when in 1936 he proved that the halting problem was undecidable
- It paved the way to prove that other problems are undecidable using reduction proofs
- We shall explore undecidable problems about Turing machines

Church-Turing Thesis and Undecidability

Theorem

Given a `tm`*, M, determining if M halts on* `EMP` *is undecidable.*

-

Church-Turing Thesis and Undecidability

Theorem

Given a tm, M, determining if M halts on EMP is undecidable.

Proof.

Assume S decides if M halts on EMP. That is, S decides the following language:

```
L = {N | N is a tm or ctm that halts on EMP}
```

We shall use S to build a ctm, R, that decides the halting problem.
The inputs for R are M and a word w. The reduction machine for τ builds a tm, M', that operates as follows:

```
1. M''s starting configuration is: (S 1 `(,LM ,BLANK))
2. M' writes w on the tape
3. Moves the head to the first blank to the left (i.e., the
   first blank on the tape)
4. Simulates M
```

Stated using the graphical ctm notation, for $w = a_1 a_2 \ldots a_n$ M' is:

R a_1 R a_2...R a_n FBL M

M' is given as input to S. If S accepts then M halts on w and the R's output is accept. Otherwise, if S rejects then M does not halt on w and the R's output is reject. Given that the halting problem is undecidable, we have a contradiction and, therefore, S cannot exist. □

Church-Turing Thesis and Undecidability

Theorem

Given a `tm`*, M, determining if their exists a word for which* `M` *halts is undecidable.*

-

Church-Turing Thesis and Undecidability

Theorem

Given a `tm`, M, determining if their exists a word for which `M` halts is undecidable.

Proof.

Assume `S` decides if their exist any word for which a given `tm` halts. That is, S decides the following language:

```
L = {N | N is a tm or ctm such that their exists a word
         for which N halts}
```

We shall use `S` to build a `ctm`, `R`, that decides if M halts on `EMP`. This language is proven undecidable in **??**. The input to `R` is, M, an arbitrary `tm`. The reduction machine for τ builds a `tm`, `M'`, that operates as follows:

```
1. Erases its input
2. Simulates M
```

`M'` is given as input to `S`. If `S` accepts this means that `M'` halts on some word if and only if it halts on all words. Therefore, M halts on `EMP`. Thus, `R` ought to accept. Otherwise, if `S` rejects this means that `M'` does not halt on any word if and only if it does not halt on all words. Therefore, M does not halt on `EMP`. Thus, `R` ought to reject. Given that determining if a `tm` halts on `EMP` is undecidable, we have a contradiction and, therefore, `S` cannot exist. □

Church-Turing Thesis and Undecidability

- HOMEWORK: 6–8

Church-Turing Thesis and Undecidability

Theorem

Given a csg, G, determining if a word, w, is in the language generated by G is undecidable.

-

Church-Turing Thesis and Undecidability

Theorem

Given a `csg`*, G, determining if a word, w, is in the language generated by G is undecidable.*

Proof.

Assume `S` decides if w∈L(G). That is, S decides the following language of doubles:

`L = {(G w) | G is a csg and w∈L(G)}`

We shall use `S` to build a `ctm`, `R`, that decides the halting problem.
The input to `R` is, M, an arbitrary `tm`, and, w, an arbitrary word. The reduction machine for τ builds a `csg`, `G'`, using M and the construction algorithm sketched in **??**. `L(G')` is the language semidecided by M.
`G'` and w are given as input to `S`. If `S` accepts then we know that `G'` generates w. This means M halts and accepts w. Therefore, `R` ought to accept. If `S` rejects then we know that `G'` does not generate w. This means M does not halt on w. Therefore, `R` ought to reject. Given that the halting problem is undecidable, we have a contradiction and, therefore, `S` cannot exist. □

Church-Turing Thesis and Undecidability

Theorem

Given a csg, G, determining if L(G)=∅ is undecidable.

-

Church-Turing Thesis and Undecidability

Theorem

Given a `csg`*, G, determining if L(G)=∅ is undecidable.*

•

Proof.

Assume `S` decides if L(G)=∅. That is, S decides the following language of doubles:

• `L = {G | G is a csg and L(G)=∅}`

We shall use `S` to build a `ctm`, `R`, that decides if there is any word for which a `tm` halts. This `tm` problem is proven undecidable in **??**.

The input to `R` is, M, an arbitrary `tm`. The reduction machine for τ builds a `csg`, `G'`, for M using the construction algorithm sketched in **??**. `L(G)'` is the language semidecided by M.

`G'` is given as input to `S`. If `S` accepts then we know that `L(G)'`=∅. This means L(M) is empty. Therefore, `R` ought to reject. If `S` rejects then we know that `L(G)'`≠∅. This means L(M) is not empty. Therefore, `R` ought to accept. Given that determining if there is any word for which a `tm` halts is undecidable, we have a contradiction and, therefore, `S` cannot exist. □

Church-Turing Thesis and Undecidability

- HOMEWORK: 11–12

Complexity

- One of the principal threads in this book is nondeterminism

Complexity

- One of the principal threads in this book is nondeterminism
- Are deterministic and nondeterministic Turing machines are equivalent?

Complexity

- A nondeterministic `tm` may be simulated by a deterministic `tm`

Complexity

- A nondeterministic `tm` may be simulated by a deterministic `tm`
- We shall develop an algorithm for a deterministic `tm` to systematically simulate all the possible steps that a nondeterministic `tm` may take–even those that do not lead to the machine halting or accepting
- Exhaustive search of all the possible transitions the nondeterministic `tm` can perform

Complexity

- Each node represents a configuration for `N`
- The deterministic machine must simulate all paths
- We need a systematic way to represent a computation
- Number branches and encode branch chosen at each step
- `(4 1)` $=$ `S` $\vdash$ `D` $\vdash$ `M`

Complexity

- The deterministic machine can perform a breadth-first search of the computation tree exploring shorter paths before longer paths

Complexity

- The deterministic machine can perform a breadth-first search of the computation tree exploring shorter paths before longer paths
- Key to making this process systematic is to define an increment function for a word (of encoded numbers) on the tape
- The increment function may be described as follows:
  1. If the rightmost number is less than `r` add 1 to it.
  2. If the rightmost number is `r` make it 1 and propagate a 1 to the left.
  3. During left propagation if a number is less than `r` add 1 to it. If the number is `r` make it a 1 and propagate left. If the blank is read then make the blank a 1 and shift the number one space to the right.

Complexity

- `(1) represents the computation S ⊢ A`
  `(2) represents the computation S ⊢ B`
  `(3) represents the computation S ⊢ C`
  `(4) represents the computation S ⊢ D`

Complexity

- ```
  (1) represents the computation S ⊢ A
  (2) represents the computation S ⊢ B
  (3) represents the computation S ⊢ C
  (4) represents the computation S ⊢ D
  ```
- At this point, the number equals r and the increment must propagate left:

  ```
  (1 1) represents the computation S ⊢ A ⊢ E
  (1 2) represents the computation S ⊢ A ⊢ F
  (1 3) represents the computation S ⊢ A ⊢ G
  (1 4) represents a computation that does not exist
  ```
  ```

Complexity

- ```
  (1) represents the computation S ⊢ A
  (2) represents the computation S ⊢ B
  (3) represents the computation S ⊢ C
  (4) represents the computation S ⊢ D
  ```
- At this point, the number equals r and the increment must propagate left:

  ```
  (1 1) represents the computation S ⊢ A ⊢ E
  (1 2) represents the computation S ⊢ A ⊢ F
  (1 3) represents the computation S ⊢ A ⊢ G
  (1 4) represents a computation that does not exist
  ```
- Once again, the number equals r and the increment must propagate left:

  ```
  (2 1) represents the computation S ⊢ B ⊢ H
  (2 2) represents the computation S ⊢ B ⊢ I
  (2 3) represents a computation that does not exist
  (2 4) represents a computation that does not exist
  ```
  ```

Complexity

- We can now outline an algorithm for a `tm` to systematically search a computation tree
- It shall use an iterative deepening strategy
- 3 main values (more values are needed in an actual implementation): the input word `w`, `N`'s current configuration `C`, and the encoded number, `I`, representing the current computation performed

Complexity

- We can now outline an algorithm for a `tm` to systematically search a computation tree
- It shall use an iterative deepening strategy
- 3 main values (more values are needed in an actual implementation): the input word `w`, `N`'s current configuration `C`, and the encoded number, `I`, representing the current computation performed
- The machine's operation is outlined as follows:

```
1. If N's starting state is a final state then halt.
2. Start with N's starting configuration and I = (1).
3. Perform the computation encoded by I if possible.
   If not possible ignore the rest of I and return C.
4. Check the configuration returned by Step 3. If it is
   in one of N's halting  states then halt. Otherwise,
   increment I, restore N's starting configuration and
   go to step 3.
```

Complexity

Theorem

A nondeterministic `tm` *semidecides* $L \Rightarrow \exists$ *a deterministic* `tm` *that semidecides* L

•

Complexity

Theorem

A nondeterministic `tm` *semidecides L* $\Rightarrow \exists$ *a deterministic* `tm` *that semidecides L*

•

Proof.

(Sketch)
Assume N is a nondeterministic `tm` that semidecides L.

Complexity

Theorem

A nondeterministic `tm` *semidecides L* $\Rightarrow \exists$ *a deterministic* `tm` *that semidecides L*

Proof.

(Sketch)

Assume N is a nondeterministic `tm` that semidecides L.

- An `mttm` with 5 main tapes shall simulate N. The purpose of each tape is described as follows:

```
Tape 0: Contains the input word, w, and is never mutated
Tape 1: Used to simulate N by recording N's configuration
Tape 2: Stores, I, the next computation number
Tape 3: Stores N's rules
Tape 4: Stores the rules applicable to the current
        configuration on Tape 1
```

More tapes may be needed to manage low-level details, but we shall not concern ourselves with these details at this time. Clearly, an `mttm` may have as many tapes as needed.

Complexity

Theorem

A nondeterministic tm semidecides L $\Rightarrow \exists$ a deterministic tm that semidecides L

Proof.

(Sketch)

Assume N is a nondeterministic tm that semidecides L.

- An mttm with 5 main tapes shall simulate N. The purpose of each tape is described as follows:

```
Tape 0: Contains the input word, w, and is never mutated
Tape 1: Used to simulate N by recording N's configuration
Tape 2: Stores, I, the next computation number
Tape 3: Stores N's rules
Tape 4: Stores the rules applicable to the current
        configuration on Tape 1
```

  More tapes may be needed to manage low-level details, but we shall not concern ourselves with these details at this time. Clearly, an mttm may have as many tapes as needed.
- The mttm starts with the input word, w, on tape 0 and operates as follows:

```
1. Write the encoding for the computation number '(1) onto tape 3
2. Write N's starting configuration on tape 2 using w on tape 1
3. Read the rightmost unprocessed encoded element, k, in the computation
   number on tape 3
4. Extract the rules from tape 3 that may be used on the configuration on
   tape 1 and copy them to tape 4
5. Apply the k^th rule on tape 4 to configuration on tape 1 and mutate tape 1
   to store this new configuration. If no such rule exists go to the next step.
6. Check if tape 1 contains a halting configuration
   a. If so, halt
   b. Otherwise, increment the computation number on tape 3 and goto step 2
```

Complexity

- HOMEWORK: 1

Complexity

- Is the solution to a solvable problem practical?

Complexity

- Is the solution to a solvable problem practical?
- To explore whether or not solvable implies a practical solution, we shall discuss the traveling salesman problem (`TSP`)
- Given `k` cities and the distances between each pair of them, find the shortest itinerary that returns to its starting point and visits each other city exactly once along the way

Complexity

- Is the solution to a solvable problem practical?
- To explore whether or not solvable implies a practical solution, we shall discuss the traveling salesman problem (`TSP`)
- Given `k` cities and the distances between each pair of them, find the shortest itinerary that returns to its starting point and visits each other city exactly once along the way
- This problem is clearly solvable:

```
1. List all possible itineraries that satisfy the
   constraint of visiting each city once
2. Return the itinerary with the shortest distance
```

Complexity

- Is the solution to a solvable problem practical?
- To explore whether or not solvable implies a practical solution, we shall discuss the traveling salesman problem (TSP)
- Given k cities and the distances between each pair of them, find the shortest itinerary that returns to its starting point and visits each other city exactly once along the way
- This problem is clearly solvable:

```
1. List all possible itineraries that satisfy the
   constraint of visiting each city once
2. Return the itinerary with the shortest distance
```

- How many times must a sum be computed?

```
                City: A ___ ___ ___ ...  ___  ___ A
Itinerary position: 0  1   2   3      k-2  k-1 k
```

Complexity

- Is the solution to a solvable problem practical?
- To explore whether or not solvable implies a practical solution, we shall discuss the traveling salesman problem (TSP)
- Given k cities and the distances between each pair of them, find the shortest itinerary that returns to its starting point and visits each other city exactly once along the way
- This problem is clearly solvable:

```
1. List all possible itineraries that satisfy the
   constraint of visiting each city once
2. Return the itinerary with the shortest distance
```

- How many times must a sum be computed?

```
                City: A ___ ___ ___ ...  ___  ___ A
Itinerary position: 0  1   2   3      k-2  k-1 k
```

- The number of different itineraries is equal to (k - 1)!

Complexity

- Is the solution to a solvable problem practical?
- To explore whether or not solvable implies a practical solution, we shall discuss the traveling salesman problem (TSP)
- Given k cities and the distances between each pair of them, find the shortest itinerary that returns to its starting point and visits each other city exactly once along the way
- This problem is clearly solvable:

```
1. List all possible itineraries that satisfy the
   constraint of visiting each city once
2. Return the itinerary with the shortest distance
```

- How many times must a sum be computed?

```
                City: A ___ ___ ___ ...  ___  ___ A
Itinerary position: 0  1   2   3      k-2  k-1 k
```

- The number of different itineraries is equal to (k - 1)!
- What if our hypothetical salesman must travel to 50 cities?

Complexity

- Is the solution to a solvable problem practical?
- To explore whether or not solvable implies a practical solution, we shall discuss the traveling salesman problem (TSP)
- Given k cities and the distances between each pair of them, find the shortest itinerary that returns to its starting point and visits each other city exactly once along the way
- This problem is clearly solvable:

```
1. List all possible itineraries that satisfy the
   constraint of visiting each city once
2. Return the itinerary with the shortest distance
```

- How many times must a sum be computed?

```
                City: A ___ ___ ___ ...  ___  ___ A
Itinerary position: 0  1   2   3      k-2  k-1 k
```

- The number of different itineraries is equal to (k - 1)!
- What if our hypothetical salesman must travel to 50 cities?
- 49! = 608281864034267560872252163321295376887552831379210240000000000
- If 10 billion itineraries could be processed per second, faster than any computer in the foreseeable future, then it would take about 19288491376023197643082577477210025903334374 40 years to find the shortest itinerary

Complexity

- Is the solution to a solvable problem practical?
- To explore whether or not solvable implies a practical solution, we shall discuss the traveling salesman problem (TSP)
- Given k cities and the distances between each pair of them, find the shortest itinerary that returns to its starting point and visits each other city exactly once along the way
- This problem is clearly solvable:

```
1. List all possible itineraries that satisfy the
   constraint of visiting each city once
2. Return the itinerary with the shortest distance
```

- How many times must a sum be computed?

```
                City: A ___ ___ ___ ...  ___  ___ A
Itinerary position: 0  1   2   3      k-2  k-1 k
```

- The number of different itineraries is equal to (k - 1)!
- What if our hypothetical salesman must travel to 50 cities?
- 49! = 608281864034267560872252163321295376887552831379210240000000000
- If 10 billion itineraries could be processed per second, faster than any computer in the foreseeable future, then it would take about 19288491376023197643082577477210025903334374 40 years to find the shortest itinerary
- Grows faster than 2^k
- The growth function is an exponential function and renders the solution impractical

Complexity

- We need a way to characterize solutions that are practical (i.e., polynomial growth)
- This eliminates deterministic `tms` that simulate nondeterministic `tms`

Complexity

- We need a way to characterize solutions that are practical (i.e., polynomial growth)
- This eliminates deterministic `tms` that simulate nondeterministic `tms`
- We define the set of Turing machines (i.e., algorithms) that are practical as:

  $\mathcal{P}$ = {`M` | `M is a deterministic Turing machine that decides a language or computes a function in a number of steps proportional to` $\mathtt{n}^k$}

- We say that any problem solvable by a `tm`$\in\mathcal{P}$ is *tractable*.

Complexity

- We need a way to characterize solutions that are practical (i.e., polynomial growth)
- This eliminates deterministic `tms` that simulate nondeterministic `tms`
- We define the set of Turing machines (i.e., algorithms) that are practical as:

  $\mathcal{P}$ = {`M` | `M is a deterministic Turing machine that decides a language or computes a function in a number of steps proportional to` $\mathtt{n}^k$}
- We say that any problem solvable by a `tm`$\in\mathcal{P}$ is *tractable*.
- Not every practical algorithm (i.e., in $\mathcal{P}$) means it is suitable for use in every day computing
- Consider a deterministic `tm` whose number of steps is proportional to $\mathtt{n}^{10}$
- For a modest input size of 100, the number of operations is proportional to 10^{100}
- It is difficult to see how such an algorithm can be practical for other than the smallest instances of the problem solved
- Thankfully, most algorithms of interest to industry programmers are solved in a number of steps proportional to $\mathtt{n}^3$ or less.

Complexity

- We can prove properties of $\mathcal{P}$ just like we proved properties for other classes of languages

Complexity

- We can prove properties of $\mathcal{P}$ just like we proved properties for other classes of languages
- For instance, $\mathcal{P}$ is closed under complement.

Complexity

- We can prove properties of $\mathcal{P}$ just like we proved properties for other classes of languages
- For instance, $\mathcal{P}$ is closed under complement.

Theorem

$\mathcal{P}$ is closed under complement.

-

Complexity

- We can prove properties of $\mathcal{P}$ just like we proved properties for other classes of languages
- For instance, $\mathcal{P}$ is closed under complement.

Theorem

$\mathcal{P}$ is closed under complement.

-

Proof.

Let M = (make-tm K Σ R S '(Y N) 'Y) be deterministic and decide a language L in polynomial time. This means M$\in\mathcal{P}$. The interpretation of the final states is as you may expect: Y is for accept and N is for reject.
From M, we can build a deterministic Turing machine, $\bar{M}$, to decide L's complement in polynomial time. Every word accepted by M ought to be rejected by $\bar{M}$ and every word rejected by M ought to be accepted by $\bar{M}$. To achieve this, $\bar{M}$ is the same as M except that the roles of Y and N are swapped. In this manner, $\bar{M}$ only accepts if M would have rejected and viceversa. □

-

Complexity

- ```
  ;; tm-language-recognizer → tm-language-recognizer
  ;; Purpose: Build a deterministic tm-language-recognizer for
  ;;          the complement of the language decided by the
  ;;          given deterministic tm-language-recognizer
  ;; Assumption: Given tm's final states are Y and N
  ;;             and Y is the accepting state
  (define (dtm4L->dtm4notL M)
    (make-tm (sm-states M)
             (sm-sigma M)
             (sm-rules M)
             (sm-start M)
             (sm-finals M)
             'N))
  ```
  ```

Complexity

- A classical problem discussed to illustrate if a problem is or is not in $\mathcal{P}$ is the Boolean satisfiability problem
- The Boolean satisfiability problem takes as input a Boolean formula in conjunctive normal form and determines if the formula is satisfiable

Complexity

- A classical problem discussed to illustrate if a problem is or is not in $\mathcal{P}$ is the Boolean satisfiability problem
- The Boolean satisfiability problem takes as input a Boolean formula in conjunctive normal form and determines if the formula is satisfiable
- Consider:

```
(and (or x y) (not x) (or y z))
```

Complexity

- A classical problem discussed to illustrate if a problem is or is not in $\mathcal{P}$ is the Boolean satisfiability problem
- The Boolean satisfiability problem takes as input a Boolean formula in conjunctive normal form and determines if the formula is satisfiable
- Consider:

```
(and (or x y) (not x) (or y z))
```

- This formula is satisfiable. Make `x = #f`:

```
(and (or #f (not y)) #t (y ∨ z))
(and (not y) (or y z)
```

Complexity

- A classical problem discussed to illustrate if a problem is or is not in $\mathcal{P}$ is the Boolean satisfiability problem
- The Boolean satisfiability problem takes as input a Boolean formula in conjunctive normal form and determines if the formula is satisfiable
- Consider:

```
(and (or x y) (not x) (or y z))
```

- This formula is satisfiable. Make `x = #f`:

```
(and (or #f (not y)) #t (y ∨ z))
(and (not y) (or y z)
```

- Make `y = #f`:

```
(and #t (or #f z)
z
```

Complexity

- A classical problem discussed to illustrate if a problem is or is not in $\mathcal{P}$ is the Boolean satisfiability problem
- The Boolean satisfiability problem takes as input a Boolean formula in conjunctive normal form and determines if the formula is satisfiable
- Consider:

```
(and (or x y) (not x) (or y z))
```

- This formula is satisfiable. Make `x = #f`:

```
(and (or #f (not y)) #t (y ∨ z))
(and (not y) (or y z)
```

- Make `y = #f`:

```
(and #t (or #f z)
z
```

- Making `z = #t` establishes the formula is satisfiable

Complexity

- A classical problem discussed to illustrate if a problem is or is not in $\mathcal{P}$ is the Boolean satisfiability problem
- The Boolean satisfiability problem takes as input a Boolean formula in conjunctive normal form and determines if the formula is satisfiable
- Consider:

```
(and (or x y) (not x) (or y z))
```

- This formula is satisfiable. Make `x = #f`:

```
(and (or #f (not y)) #t (y ∨ z))
(and (not y) (or y z)
```

- Make `y = #f`:

```
(and #t (or #f z)
z
```

- Making `z = #t` establishes the formula is satisfiable
- In contrast, the following formula is not satisfiable:

```
(and (or x y z)
     (or (not x) (not y) (not z))
     (or x (not y))
     (or y (not z))
     (or (not x) z))
```

- `(or x y z)`, is satisfied if any variable is true
- `(or (not x) (not y) (not z))` is satisfied if any variable is false
- At least one variable must be true and at least one variable must be false

Complexity

- A classical problem discussed to illustrate if a problem is or is not in $\mathcal{P}$ is the Boolean satisfiability problem
- The Boolean satisfiability problem takes as input a Boolean formula in conjunctive normal form and determines if the formula is satisfiable
- Consider:

```
(and (or x y) (not x) (or y z))
```

- This formula is satisfiable. Make `x = #f`:

```
(and (or #f (not y)) #t (y ∨ z))
(and (not y) (or y z)
```

- Make `y = #f`:

```
(and #t (or #f z)
z
```

- Making `z = #t` establishes the formula is satisfiable
- In contrast, the following formula is not satisfiable:

```
(and (or x y z)
     (or (not x) (not y) (not z))
     (or x (not y))
     (or y (not z))
     (or (not x) z))
```

- `(or x y z)`, is satisfied if any variable is true
- `(or (not x) (not y) (not z))` is satisfied if any variable is false
- At least one variable must be true and at least one variable must be false
- The remaining conjunction:

```
(and (or x (not y)) (or y (not z)) (or (not x) z))
```

- Only satisfiable if the value of all three variables is the same

Complexity

- We shall now explore how to solve a simplified version of the Boolean satisfiability problem called the 2-satisfiability problem
- In this simplified version every clause of a formula in disjunctive normal form is either a singleton or a 2-disjunction

Complexity

- We shall now explore how to solve a simplified version of the Boolean satisfiability problem called the 2-satisfiability problem
- In this simplified version every clause of a formula in disjunctive normal form is either a singleton or a 2-disjunction
- The solution presented shall draw upon our knowledge of grammars to represent input formulae given by the user
- In addition, the grammar for input formulae is used to develop a formula parser
- As you may have learned in a Programming Languages course, parsers produce parse trees that eliminate the need for superfluous information like parentheses and keywords (e.g., `not` and `or`)

Complexity

- Context-Free Grammar for the 2-Satisfiability Problem Formulae

```
iformula → '(and iclauses)

iclauses → ()
         → (cons iclause iclauses)

 iclause → isingleton
         → i2disjunction

isingleton → symbol
           → (not symbol)

i2disjunction → (or isingleton isingleton)
```

Complexity

- Context-Free Grammar for the 2-Satisfiability Problem Formulae

```
iformula → '(and iclauses)

iclauses → ()
         → (cons iclause iclauses)

 iclause → isingleton
         → i2disjunction

isingleton → symbol
           → (not symbol)

i2disjunction → (or isingleton isingleton)
```

- Representation for the abstract
- We define formula as follows:

```
;; A formula → '()
;;           → (cons clause formula)
```

Complexity

- Context-Free Grammar for the 2-Satisfiability Problem Formulae

```
iformula → '(and iclauses)

iclauses → ()
         → (cons iclause iclauses)

 iclause → isingleton
         → i2disjunction

isingleton → symbol
           → (not symbol)

i2disjunction → (or isingleton isingleton)
```

- Representation for the abstract
- We define formula as follows:

```
;; A formula → '()
;;           → (cons clause formula)
```

- A clause is defined as follows:

```
;; clause → singleton
;;        → 2disjunction
```

Complexity

- Context-Free Grammar for the 2-Satisfiability Problem Formulae

```
iformula → '(and iclauses)

iclauses → ()
         → (cons iclause iclauses)

 iclause → isingleton
         → i2disjunction

isingleton → symbol
           → (not symbol)

i2disjunction → (or isingleton isingleton)
```

- Representation for the abstract
- We define formula as follows:

```
;; A formula → '()
;;           → (cons clause formula)
```

- A clause is defined as follows:

```
;; clause → singleton
;;        → 2disjunction
```

- A singleton is defined as follows:

```
;; singleton → (var symbol)
;;           → (notvar symbol)

(struct var (symb) #:transparent)
(struct notvar (symb) #:transparent)
```

Complexity

- Context-Free Grammar for the 2-Satisfiability Problem Formulae

```
iformula → '(and iclauses)

iclauses → ()
         → (cons iclause iclauses)

 iclause → isingleton
         → i2disjunction

isingleton → symbol
           → (not symbol)

i2disjunction → (or isingleton isingleton)
```

- Representation for the abstract
- We define formula as follows:

```
;; A formula → '()
;;           → (cons clause formula)
```

- A clause is defined as follows:

```
;; clause → singleton
;;        → 2disjunction
```

- A singleton is defined as follows:

```
;; singleton → (var symbol)
;;           → (notvar symbol)

(struct var (symb) #:transparent)
(struct notvar (symb) #:transparent)
```

- A disjunction is represented as follows:

```
;; disjunction → (2disjunction singleton singleton)

(struct 2disjunction (s1 s2) #:transparent)
```

Complexity

- The parser

```
;; iformula → formula
;; Purpose: Parse the given iformula
(define (parse-iformula an-iformula)
```

- `(map parse-iclause (rest an-iformula)))`

Complexity

- The parser

```
;; iformula → formula
;; Purpose: Parse the given iformula
(define (parse-iformula an-iformula)
```

- ```
  ;; icl → clause
  ;; Purpose: Parse the given icl
  (define (parse-iclause an-icl)
  ```

- ```
  (if (isingleton? an-icl)
      (parse-isingleton an-icl)
      (parse-i2disjunction (rest an-icl))))
  ```

- ```
  (map parse-iclause (rest an-iformula)))
  ```
  ```

Complexity

- The parser

```
;; iformula → formula
;; Purpose: Parse the given iformula
(define (parse-iformula an-iformula)
```

- ```
  ;; icl → clause
  ;; Purpose: Parse the given icl
  (define (parse-iclause an-icl)
  ```

- ```
  ;; iclause → Boolean
  ;; Purpose: Determine if the given iclause is an isingleton
  (define (isingleton? an-icl)
    (or (symbol? an-icl)
        (eq? (first an-icl) 'not)))
  ```

- ```
  (if (isingleton? an-icl)
      (parse-isingleton an-icl)
      (parse-i2disjunction (rest an-icl))))
  ```

- ```
  (map parse-iclause (rest an-iformula)))
  ```

Complexity

- The parser

```
;; iformula → formula
;; Purpose: Parse the given iformula
(define (parse-iformula an-iformula)
```

- ```
  ;; icl → clause
  ;; Purpose: Parse the given icl
  (define (parse-iclause an-icl)
  ```

- ```
    ;; isingleton → singleton
    ;; Purpose: Parse the given isingleton
    (define (parse-isingleton an-is)
      (if (symbol? an-is)
          (var an-is)
          (notvar (second an-is))))
  ```

- ```
    ;; iclause → Boolean
    ;; Purpose: Determine if the given iclause is an isingleton
    (define (isingleton? an-icl)
      (or (symbol? an-icl)
          (eq? (first an-icl) 'not)))
  ```

- ```
  (if (isingleton? an-icl)
      (parse-isingleton an-icl)
      (parse-i2disjunction (rest an-icl))))
  ```

- ```
  (map parse-iclause (rest an-iformula)))
  ```
  ```

Complexity

- The parser

```
;; iformula → formula
;; Purpose: Parse the given iformula
(define (parse-iformula an-iformula)
```

- ```
  ;; icl → clause
  ;; Purpose: Parse the given icl
  (define (parse-iclause an-icl)
  ```

- ```
    ;; isingleton → singleton
    ;; Purpose: Parse the given isingleton
    (define (parse-isingleton an-is)
      (if (symbol? an-is)
          (var an-is)
          (notvar (second an-is))))
    ```

- ```
    ;; idisjunction → disjunction
    ;; Parse: Parse the given idisjunction
    (define (parse-i2disjunction an-id)
      (2disjunction (parse-isingleton (first an-id))
                    (parse-isingleton (second an-id))))
    ```

- ```
    ;; iclause → Boolean
    ;; Purpose: Determine if the given iclause is an isingleton
    (define (isingleton? an-icl)
      (or (symbol? an-icl)
          (eq? (first an-icl) 'not)))
    ```

- ```
  (if (isingleton? an-icl)
      (parse-isingleton an-icl)
      (parse-i2disjunction (rest an-icl))))
  ```

- ```
  (map parse-iclause (rest an-iformula)))
  ```

Complexity

- The 2-satisfiability solver searches for a set of assignments to satisfy a given `formula`
- Returns `'accept` if variables may be assigned values that satisfy the formula. Otherwise, `'reject` is returned.

Complexity

- The 2-satisfiability solver searches for a set of assignments to satisfy a given `formula`
- Returns `'accept` if variables may be assigned values that satisfy the formula. Otherwise, `'reject` is returned.
- An accumulator to store the current variable assignments in a list is used
- The accumulator invariant is that the accumulator contains the variable bindings made so far to satisfy the formula
- The function first checks if any variable assignment is possible
- If the given formula is empty then all needed variable assignments have been made and the accumulator is returned
- If the formula has no solution then the empty list is returned
- A formula has no solution when a variable and its complement are both clauses in the formula

Complexity

- The 2-satisfiability solver searches for a set of assignments to satisfy a given `formula`
- Returns `'accept` if variables may be assigned values that satisfy the formula. Otherwise, `'reject` is returned.
- An accumulator to store the current variable assignments in a list is used
- The accumulator invariant is that the accumulator contains the variable bindings made so far to satisfy the formula
- The function first checks if any variable assignment is possible
- If the given formula is empty then all needed variable assignments have been made and the accumulator is returned
- If the formula has no solution then the empty list is returned
- A formula has no solution when a variable and its complement are both clauses in the formula
- If the formula contains one or more singletons the function assigns a value to a single variable, simplifies the formula based on the assignment made, and recursively searches for assignments using the simplified formula

Complexity

- The 2-satisfiability solver searches for a set of assignments to satisfy a given `formula`
- Returns `'accept` if variables may be assigned values that satisfy the formula. Otherwise, `'reject` is returned.
- An accumulator to store the current variable assignments in a list is used
- The accumulator invariant is that the accumulator contains the variable bindings made so far to satisfy the formula
- The function first checks if any variable assignment is possible
- If the given formula is empty then all needed variable assignments have been made and the accumulator is returned
- If the formula has no solution then the empty list is returned
- A formula has no solution when a variable and its complement are both clauses in the formula
- If the formula contains one or more singletons the function assigns a value to a single variable, simplifies the formula based on the assignment made, and recursively searches for assignments using the simplified formula
- If the formula does not contains any singleton clauses then it must only contain disjunctions
- In this case, a backtracking strategy is used
- The first clause's first singleton is assigned a value, the formula is simplified based on the assignment, and the simplified formula is solved
- If this search is successful then the found list of bindings is returned
- Otherwise, the first clause's first singleton is assigned the complement of its first assignment, the formula is simplified based on this new assignment, and the simplified formula is solved
- The result of this second search for assignments is returned.

Complexity

- The solver must distinguish between four properties the given formula may have:
  1. The formula is empty.
  2. The formula has no solution because it contains both a variable and its complement as clauses.
  3. The formula contains a singleton.
  4. The formula only contains disjunctions.

Complexity

- The 2-Satisfiability Solver

```
;; formula → Boolean     Purpose: Determine if given formula is satisfiable
(define (2-satisfiability a-formula)
  ...
```

- `(if (not (null? (solve a-formula '()))) 'accept 'reject))`

Complexity

- The 2-Satisfiability Solver

```
;; formula → Boolean     Purpose: Determine if given formula is satisfiable
(define (2-satisfiability a-formula)
  ...
```

- `(define (solve a-formula acc)`

- `(if (not (null? (solve a-formula '()))) 'accept 'reject))`

Complexity

- The 2-Satisfiability Solver

```
;; formula → Boolean     Purpose: Determine if given formula is satisfiable
(define (2-satisfiability a-formula)
  ...
```

- `  (define (solve a-formula acc)`
- `    (cond [(empty? a-formula) acc]`
- `  (if (not (null? (solve a-formula '()))) 'accept 'reject))`

Complexity

- The 2-Satisfiability Solver

```
;; formula → Boolean     Purpose: Determine if given formula is satisfiable
(define (2-satisfiability a-formula)
  ...
  (define (solve a-formula acc)
    (cond [(empty? a-formula) acc]
          [(has-no-solution? (filter (λ (c) (not (2disjunction? c))) a-formula))
           '()]
```

```
  (if (not (null? (solve a-formula '()))) 'accept 'reject))
```

Complexity

- The 2-Satisfiability Solver

```
;; formula → Boolean     Purpose: Determine if given formula is satisfiable
(define (2-satisfiability a-formula)
  ...
  (define (solve a-formula acc)
    (cond [(empty? a-formula) acc]
          [(has-no-solution? (filter (λ (c) (not (2disjunction? c))) a-formula))
           '()]
          [(ormap (λ (c) (or (var? c) (notvar? c))) a-formula)
           (let [(form-vars (filter var? a-formula))
                 (form-notvars (filter notvar? a-formula))]
             (solve (remove-duplicates
                     (simplify-formula a-formula
                                       (if (null? form-notvars)
                                           (first form-vars)
                                           (first form-notvars))))
                    (if (null? form-notvars)
                        (cons (list (var-symb (first form-vars)) #t) acc)
                        (cons (list (notvar-symb (first form-notvars)) #f) acc))))]
```

```
  (if (not (null? (solve a-formula '()))) 'accept 'reject))
```

Complexity

- The 2-Satisfiability Solver

```
;; formula → Boolean     Purpose: Determine if given formula is satisfiable
(define (2-satisfiability a-formula)
  ...
  (define (solve a-formula acc)
    (cond [(empty? a-formula) acc]
          [(has-no-solution? (filter (λ (c) (not (2disjunction? c))) a-formula))
           '()]
          [(ormap (λ (c) (or (var? c) (notvar? c))) a-formula)
           (let [(form-vars (filter var? a-formula))
                 (form-notvars (filter notvar? a-formula))]
             (solve (remove-duplicates
                     (simplify-formula a-formula
                                       (if (null? form-notvars)
                                           (first form-vars)
                                           (first form-notvars))))
                    (if (null? form-notvars)
                        (cons (list (var-symb (first form-vars)) #t) acc)
                        (cons (list (notvar-symb (first form-notvars)) #f) acc))))]
          [else ;; a-formula only has 2disjunctions
           (let* [(fsingleton (2disjunction-s1 (first a-formula)))
                  (fvar (if (var? fsingleton)
                            (var-symb fsingleton)
                            (notvar-symb fsingleton)))
                  (not-fsingleton (complement-singleton fsingleton))
                  (sol1 (solve (simplify-formula a-formula fsingleton)
                               (cons (list fvar (var? fsingleton)) acc)))]
             (if (not (null? sol1))
                 sol1
                 (solve (simplify-formula a-formula not-fsingleton)
                        (cons (list fvar (notvar? fsingleton)) acc))))]))
  (if (not (null? (solve a-formula '()))) 'accept 'reject))
```

Complexity

- Is the 2-Satisfiability Problem is in $\mathcal{P}$?

Complexity

- Is the 2-Satisfiability Problem is in $\mathcal{P}$?
- Number of variables in the given formula `n`

Complexity

- Is the 2-Satisfiability Problem is in $\mathcal{P}$?
- Number of variables in the given formula `n`
- `complement-singleton` performs a constant number of operations

Complexity

- Is the 2-Satisfiability Problem is in $\mathcal{P}$?
- Number of variables in the given formula `n`
- `complement-singleton` performs a constant number of operations
- For `has-no-solution?` the number of operations performed is proportional to $\mathtt{n}^2$

Complexity

- Is the 2-Satisfiability Problem is in $\mathcal{P}$?
- Number of variables in the given formula `n`
- `complement-singleton` performs a constant number of operations
- For `has-no-solution?` the number of operations performed is proportional to $\mathtt{n}^2$
- For `simplify` the number of operations is proportional to `n`

Complexity

- Is the 2-Satisfiability Problem is in $\mathcal{P}$?
- Number of variables in the given formula `n`
- `complement-singleton` performs a constant number of operations
- For `has-no-solution?` the number of operations performed is proportional to $\mathtt{n}^2$
- For `simplify` the number of operations is proportional to `n`
- For `simplify-formula`, that uses `simplify`, the number of operations performed is proportional to $\mathtt{n}^2$

Complexity

- Is the 2-Satisfiability Problem is in $\mathcal{P}$?
- Number of variables in the given formula `n`
- `complement-singleton` performs a constant number of operations
- For `has-no-solution?` the number of operations performed is proportional to $\mathtt{n}^2$
- For `simplify` the number of operations is proportional to `n`
- For `simplify-formula`, that uses `simplify`, the number of operations performed is proportional to $\mathtt{n}^2$
- For `solve`, in the worst case 2`n` recursive calls are made and for each $\mathtt{n}^2$ operations are done.

Complexity

- Is the 2-Satisfiability Problem is in $\mathcal{P}$?
- Number of variables in the given formula `n`
- `complement-singleton` performs a constant number of operations
- For `has-no-solution?` the number of operations performed is proportional to $\mathtt{n}^2$
- For `simplify` the number of operations is proportional to `n`
- For `simplify-formula`, that uses `simplify`, the number of operations performed is proportional to $\mathtt{n}^2$
- For `solve`, in the worst case $2\mathtt{n}$ recursive calls are made and for each $\mathtt{n}^2$ operations are done.
- In the worst case, the total number of operations is proportional to $\mathtt{n}^3$
- The 2-satisfiability problem is in $\mathcal{P}$.

Complexity

- A primary goal of complexity theory is to discover mathematical approaches to establish that solutions to practical problems are not in P
- One such problem we have discussed that appears not to be in P is the traveling salesman problem
- We use the word *appears*, because the best known algorithm implemented as a deterministic Turing machine takes an exponential number of steps to find the solution
- A nondeterministic Turing machine finds the solution in a polynomial number of steps

Complexity

- A primary goal of complexity theory is to discover mathematical approaches to establish that solutions to practical problems are not in P
- One such problem we have discussed that appears not to be in P is the traveling salesman problem
- We use the word *appears*, because the best known algorithm implemented as a deterministic Turing machine takes an exponential number of steps to find the solution
- A nondeterministic Turing machine finds the solution in a polynomial number of steps
- Separating nondeterminism and determinism in terms of polynomial running time is one of the biggest and most profound problems in Computer Science
- It is an open research question and remains unanswered

Complexity

- A primary goal of complexity theory is to discover mathematical approaches to establish that solutions to practical problems are not in `P`
- One such problem we have discussed that appears not to be in `P` is the traveling salesman problem
- We use the word *appears*, because the best known algorithm implemented as a deterministic Turing machine takes an exponential number of steps to find the solution
- A nondeterministic Turing machine finds the solution in a polynomial number of steps
- Separating nondeterminism and determinism in terms of polynomial running time is one of the biggest and most profound problems in Computer Science
- It is an open research question and remains unanswered
- To explore problems not in $\mathcal{P}$, it helps to formally define what it means that a computation by a nondeterministic Turing machine is bounded by a polynomial number of steps
- A nondeterministic Turing machine, `M`, is bounded by a polynomial, `p(n)`, if there is no (possible) computation that takes more than `p(n)` steps

Complexity

- A primary goal of complexity theory is to discover mathematical approaches to establish that solutions to practical problems are not in `P`
- One such problem we have discussed that appears not to be in `P` is the traveling salesman problem
- We use the word *appears*, because the best known algorithm implemented as a deterministic Turing machine takes an exponential number of steps to find the solution
- A nondeterministic Turing machine finds the solution in a polynomial number of steps
- Separating nondeterminism and determinism in terms of polynomial running time is one of the biggest and most profound problems in Computer Science
- It is an open research question and remains unanswered
- To explore problems not in $\mathcal{P}$, it helps to formally define what it means that a computation by a nondeterministic Turing machine is bounded by a polynomial number of steps
- A nondeterministic Turing machine, `M`, is bounded by a polynomial, `p(n)`, if there is no (possible) computation that takes more than `p(n)` steps
- We define the nondeterministic polynomial class of languages, `NP`, as those languages decided by a polynomially bounded nondeterministic Turing machine

Complexity

- Computation tree for a nondeterministic Turing machine

- Recall that a deterministic Turing machine may simulate a nondeterministic Turing machine by performing a breadth-first traversal
- Clear why nondeterminism in Turing machines is so powerful
- Only the configurations on a single path to accept need to be visited

Complexity

- Most computer scientists today believe that the Boolean satisfiability problem is not in $\mathcal{P}$
- We shall demonstrate that it is $\mathcal{NP}$
- Specifically, we shall describe a nondeterministic 2-tape Turing machine that decides the Boolean satisfiability problem

Complexity

- Most computer scientists today believe that the Boolean satisfiability problem is not in $\mathcal{P}$
- We shall demonstrate that it is $\mathcal{NP}$
- Specifically, we shall describe a nondeterministic 2-tape Turing machine that decides the Boolean satisfiability problem
- Represent true as `T` and false as `F`
- `n` denotes the number of (distinct) variables and `m` denotes the number of clauses

Complexity

- Most computer scientists today believe that the Boolean satisfiability problem is not in $\mathcal{P}$
- We shall demonstrate that it is $\mathcal{NP}$
- Specifically, we shall describe a nondeterministic 2-tape Turing machine that decides the Boolean satisfiability problem
- Represent true as `T` and false as `F`
- `n` denotes the number of (distinct) variables and `m` denotes the number of clauses

- The machine operates in three stages:
  1. For every distinct variable in the formula on tape 1 write an `x` on tape 2.
  2. Nondeterministically, write an assignment on tape 2. Each `x` on tape 2 is nondeterministically substituted with a `T` or an `F`.
  3. Check that each clause in the formula on tape 1 contains a singleton that is true given the assignment on tape 2. If this is the case, the formula is satisfiable and the machine accepts. Otherwise, it rejects.

Complexity

- Most computer scientists today believe that the Boolean satisfiability problem is not in $\mathcal{P}$
- We shall demonstrate that it is $\mathcal{NP}$
- Specifically, we shall describe a nondeterministic 2-tape Turing machine that decides the Boolean satisfiability problem
- Represent true as `T` and false as `F`
- `n` denotes the number of (distinct) variables and `m` denotes the number of clauses
- The machine operates in three stages:
  1. For every distinct variable in the formula on tape 1 write an `x` on tape 2.
  2. Nondeterministically, write an assignment on tape 2. Each `x` on tape 2 is nondeterministically substituted with a `T` or an `F`.
  3. Check that each clause in the formula on tape 1 contains a singleton that is true given the assignment on tape 2. If this is the case, the formula is satisfiable and the machine accepts. Otherwise, it rejects.
- Stage 1 is accomplished deterministically in a polynomial number of steps: $\mathtt{n}^2$

Complexity

- Most computer scientists today believe that the Boolean satisfiability problem is not in $\mathcal{P}$
- We shall demonstrate that it is $\mathcal{NP}$
- Specifically, we shall describe a nondeterministic 2-tape Turing machine that decides the Boolean satisfiability problem
- Represent true as `T` and false as `F`
- `n` denotes the number of (distinct) variables and `m` denotes the number of clauses

- The machine operates in three stages:
  1. For every distinct variable in the formula on tape 1 write an `x` on tape 2.
  2. Nondeterministically, write an assignment on tape 2. Each `x` on tape 2 is nondeterministically substituted with a `T` or an `F`.
  3. Check that each clause in the formula on tape 1 contains a singleton that is true given the assignment on tape 2. If this is the case, the formula is satisfiable and the machine accepts. Otherwise, it rejects.
- Stage 1 is accomplished deterministically in a polynomial number of steps: $\mathtt{n}^2$
- Stage 2 is accomplished nondeterministically in `n` steps by traversing the `x`s on tape 2

Complexity

- Most computer scientists today believe that the Boolean satisfiability problem is not in $\mathcal{P}$
- We shall demonstrate that it is $\mathcal{NP}$
- Specifically, we shall describe a nondeterministic 2-tape Turing machine that decides the Boolean satisfiability problem
- Represent true as `T` and false as `F`
- `n` denotes the number of (distinct) variables and `m` denotes the number of clauses

- The machine operates in three stages:
  1. For every distinct variable in the formula on tape 1 write an `x` on tape 2.
  2. Nondeterministically, write an assignment on tape 2. Each `x` on tape 2 is nondeterministically substituted with a `T` or an `F`.
  3. Check that each clause in the formula on tape 1 contains a singleton that is true given the assignment on tape 2. If this is the case, the formula is satisfiable and the machine accepts. Otherwise, it rejects.
- Stage 1 is accomplished deterministically in a polynomial number of steps: $\mathtt{n}^2$
- Stage 2 is accomplished nondeterministically in `n` steps by traversing the `x`s on tape 2
- Stage 3 is done deterministically by processing the `m` clauses
- For each clause, its singletons are traversed to determine if any evaluates to true
- If all clauses are processed, then the machine accepts

Complexity

- Most computer scientists today believe that the Boolean satisfiability problem is not in $\mathcal{P}$
- We shall demonstrate that it is $\mathcal{NP}$
- Specifically, we shall describe a nondeterministic 2-tape Turing machine that decides the Boolean satisfiability problem
- Represent true as `T` and false as `F`
- `n` denotes the number of (distinct) variables and `m` denotes the number of clauses

- The machine operates in three stages:
  1. For every distinct variable in the formula on tape 1 write an `x` on tape 2.
  2. Nondeterministically, write an assignment on tape 2. Each `x` on tape 2 is nondeterministically substituted with a `T` or an `F`.
  3. Check that each clause in the formula on tape 1 contains a singleton that is true given the assignment on tape 2. If this is the case, the formula is satisfiable and the machine accepts. Otherwise, it rejects.
- Stage 1 is accomplished deterministically in a polynomial number of steps: $\mathtt{n}^2$
- Stage 2 is accomplished nondeterministically in `n` steps by traversing the `x`s on tape 2
- Stage 3 is done deterministically by processing the `m` clauses
- For each clause, its singletons are traversed to determine if any evaluates to true
- If all clauses are processed, then the machine accepts
- To process a clause, the number of singletons is bounded by `2n`
- For each plug-in its value
- If it evaluates to true, then the machine moves to the next clause
- If it evaluates to false, then it moves to the next singleton in the clause
- If none of the singletons evaluate to true then the machine moves to reject
- The number of steps for stage 3 is proportional to `m*2n`

Complexity

- Most computer scientists today believe that the Boolean satisfiability problem is not in $\mathcal{P}$
- We shall demonstrate that it is $\mathcal{NP}$
- Specifically, we shall describe a nondeterministic 2-tape Turing machine that decides the Boolean satisfiability problem
- Represent true as `T` and false as `F`
- `n` denotes the number of (distinct) variables and `m` denotes the number of clauses

- The machine operates in three stages:
  1. For every distinct variable in the formula on tape 1 write an `x` on tape 2.
  2. Nondeterministically, write an assignment on tape 2. Each `x` on tape 2 is nondeterministically substituted with a `T` or an `F`.
  3. Check that each clause in the formula on tape 1 contains a singleton that is true given the assignment on tape 2. If this is the case, the formula is satisfiable and the machine accepts. Otherwise, it rejects.
- Stage 1 is accomplished deterministically in a polynomial number of steps: $\mathtt{n}^2$
- Stage 2 is accomplished nondeterministically in `n` steps by traversing the `x`s on tape 2
- Stage 3 is done deterministically by processing the `m` clauses
- For each clause, its singletons are traversed to determine if any evaluates to true
- If all clauses are processed, then the machine accepts
- To process a clause, the number of singletons is bounded by `2n`
- For each plug-in its value
- If it evaluates to true, then the machine moves to the next clause
- If it evaluates to false, then it moves to the next singleton in the clause
- If none of the singletons evaluate to true then the machine moves to reject
- The number of steps for stage 3 is proportional to `m*2n`
- The toal number of operations is proportional to $\mathtt{n}^2 + \mathtt{n} + (\mathtt{max\ n\ m})^2$
- The Boolean satisfiability problem is in $\mathcal{NP}$

Complexity

- Demonstrating that a problem is in $\mathcal{NP}$ requires solving a problem using a nondeterministic Turing machine that is bounded by a polynomial number of steps

Complexity

- Demonstrating that a problem is in $\mathcal{NP}$ requires solving a problem using a nondeterministic Turing machine that is bounded by a polynomial number of steps
- An interesting question we may ask ourselves, is $\mathcal{NP}$ closed under complement?

Complexity

- Demonstrating that a problem is in $\mathcal{NP}$ requires solving a problem using a nondeterministic Turing machine that is bounded by a polynomial number of steps
- An interesting question we may ask ourselves, is $\mathcal{NP}$ closed under complement?
- The answer to this question is currently unknown
- Many properties of $\mathcal{NP}$ remain elusive to establish and this is an area of active research

Complexity

- Demonstrating that a problem is in $\mathcal{NP}$ requires solving a problem using a nondeterministic Turing machine that is bounded by a polynomial number of steps
- An interesting question we may ask ourselves, is $\mathcal{NP}$ closed under complement?
- The answer to this question is currently unknown
- Many properties of $\mathcal{NP}$ remain elusive to establish and this is an area of active research
- One property that is immediately obvious, however, is that $\mathcal{P} \subseteq \mathcal{NP}$
- This follows by observing that a deterministic Turing machine is a nondeterministic Turing machine whose transition relation is a function

Complexity

- Demonstrating that a problem is in $\mathcal{NP}$ requires solving a problem using a nondeterministic Turing machine that is bounded by a polynomial number of steps
- An interesting question we may ask ourselves, is $\mathcal{NP}$ closed under complement?
- The answer to this question is currently unknown
- Many properties of $\mathcal{NP}$ remain elusive to establish and this is an area of active research
- One property that is immediately obvious, however, is that $\mathcal{P} \subseteq \mathcal{NP}$
- This follows by observing that a deterministic Turing machine is a nondeterministic Turing machine whose transition relation is a function
- This observation suggests another question: Is $\mathcal{P} = \mathcal{NP}$?

Complexity

- Demonstrating that a problem is in $\mathcal{NP}$ requires solving a problem using a nondeterministic Turing machine that is bounded by a polynomial number of steps
- An interesting question we may ask ourselves, is $\mathcal{NP}$ closed under complement?
- The answer to this question is currently unknown
- Many properties of $\mathcal{NP}$ remain elusive to establish and this is an area of active research
- One property that is immediately obvious, however, is that $\mathcal{P} \subseteq \mathcal{NP}$
- This follows by observing that a deterministic Turing machine is a nondeterministic Turing machine whose transition relation is a function
- This observation suggests another question: Is $\mathcal{P} = \mathcal{NP}$?
- This question remains unanswered and is one of the biggest unsolved problems in Computer Science

Complexity

- HOMEWORK: 6

Farewell and where to go from here?

- Congratulations! You have completed your first steps into the thought-provoking world of Theoretical Computer Science
- You have learned about different models of computation: their powers and their limitations
- You have also learned about nondeterminism and the role it plays in Computer Science and in programming
- The models you have studied have varied applications across computer science. Explore them!

Farewell and where to go from here?

- Congratulations! You have completed your first steps into the thought-provoking world of Theoretical Computer Science
- You have learned about different models of computation: their powers and their limitations
- You have also learned about nondeterminism and the role it plays in Computer Science and in programming
- The models you have studied have varied applications across computer science. Explore them!
- Most importantly, you now understand how machines compute functions, how decidability problems may be solved, and the meaning of the word *algorithm*
- Understanding what an algorithm is has naturally led to thinking about what can and can not be computed and what it means for a solution to be practical
- You have become a better programmer and you have begun to explore some of humanity's limitations

Farewell and where to go from here?

- Believe it or not, you have only touched the tip of the iceberg when it comes to formal languages, automata theory, and complexity theory

Farewell and where to go from here?

- Believe it or not, you have only touched the tip of the iceberg when it comes to formal languages, automata theory, and complexity theory
- Is it possible for a `cfg` to always derive a word or state that the word is not in the language?

Farewell and where to go from here?

- Believe it or not, you have only touched the tip of the iceberg when it comes to formal languages, automata theory, and complexity theory
- Is it possible for a `cfg` to always derive a word or state that the word is not in the language?
- It turns out that the answer to this question is almost always yes for "long" words by transforming to Chomsky normal form or Greibach normal form

Farewell and where to go from here?

- Believe it or not, you have only touched the tip of the iceberg when it comes to formal languages, automata theory, and complexity theory
- Is it possible for a `cfg` to always derive a word or state that the word is not in the language?
- It turns out that the answer to this question is almost always yes for "long" words by transforming to Chomsky normal form or Greibach normal form
- Equally intriguing, there are many examples of undecidable problems you can explore
- Consider a set of dominos such that each domino has two words on the same face (e.g., one of the top and one on the bottom)
- Can dominos be placed in a row (repetitions allowed) such that the appending of top strings is the same as the appending of the bottom strings?

Farewell and where to go from here?

- Believe it or not, you have only touched the tip of the iceberg when it comes to formal languages, automata theory, and complexity theory
- Is it possible for a `cfg` to always derive a word or state that the word is not in the language?
- It turns out that the answer to this question is almost always yes for "long" words by transforming to Chomsky normal form or Greibach normal form
- Equally intriguing, there are many examples of undecidable problems you can explore
- Consider a set of dominos such that each domino has two words on the same face (e.g., one of the top and one on the bottom)
- Can dominos be placed in a row (repetitions allowed) such that the appending of top strings is the same as the appending of the bottom strings?
- This fun puzzle is known as the Post correspondence problem and it is an unsolvable problem

Farewell and where to go from here?

- There are also a myriad of topics you may explore that are not directly suggested by the topics covered in this textbook

Farewell and where to go from here?

- There are also a myriad of topics you may explore that are not directly suggested by the topics covered in this textbook
- Have you ever heard of quantum finite automata?

Farewell and where to go from here?

- There are also a myriad of topics you may explore that are not directly suggested by the topics covered in this textbook
- Have you ever heard of quantum finite automata?
- Automata theory has applications in quantum computing

Farewell and where to go from here?

- There are also a myriad of topics you may explore that are not directly suggested by the topics covered in this textbook
- Have you ever heard of quantum finite automata?
- Automata theory has applications in quantum computing
- Automata theory also has applications in program verification
- There is a special kind of invariant known as a *preserved invariant*
- A preserved invariant is one such that if it holds for some state `Q` then it also holds for any state reachable from `Q`
- This fascinating type of invariant may be used to prove that software is correct

Farewell and where to go from here?

- There are also a myriad of topics you may explore that are not directly suggested by the topics covered in this textbook
- Have you ever heard of quantum finite automata?
- Automata theory has applications in quantum computing
- Automata theory also has applications in program verification
- There is a special kind of invariant known as a *preserved invariant*
- A preserved invariant is one such that if it holds for some state `Q` then it also holds for any state reachable from `Q`
- This fascinating type of invariant may be used to prove that software is correct
- State-based machines also have alluring applications in artificial intelligence
- *neural* `tm`s are used to model artificial neural networks that exhibit temporal dynamic behavior

Farewell and where to go from here?

- You are now well-equipped to explore formal languages and automata theory applications in many fields of Computer Science as well as to deepen your understanding of Computer Science's theoretical underpinnings
- You may do so by either by taking more advanced courses or by personal inquiry
- Regardless of your approach, you bring with you actual programming experience with state-based machines, grammars, and regular expressions
- Mind what you have learned and apply it in your future intellectual endeavors

Farewell and where to go from here?

- You are now well-equipped to explore formal languages and automata theory applications in many fields of Computer Science as well as to deepen your understanding of Computer Science's theoretical underpinnings
- You may do so by either by taking more advanced courses or by personal inquiry
- Regardless of your approach, you bring with you actual programming experience with state-based machines, grammars, and regular expressions
- Mind what you have learned and apply it in your future intellectual endeavors
- Above all, enjoy the challenges that come with process of discovery and the rigor required to prove your designs correct!
- Adiós!