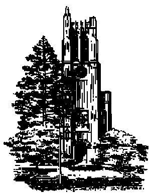
Wednesday June 29, 2016
Lecture 26

Modeling Computation
Sections 13.1 – 13.2



Notables

- Test #7 on Thursday
- Try the following problems:
  - Page 856, problems 5, and 17

Important theory

- What can a symbol processing computer do?
- What can't it do?
- How can we model sets of strings representing
  - Inputs to an algorithm or program?
  - Outputs of an algorithm or program?
  - An algorithm or program itself?
- Are some sets more difficult to recognize than others?

What are the limits on computing?

Can we build different kinds of computers to transcend these limits?

Important applications

- Models for programming language syntax.
- Models for defining program input.
- Methods for designing finite state machines, or FSAs.

a) A recognizer for C++ integers
b) A vending machine

Grammar

- A simple mechanism to describe or generate a *language* (set of strings)
  - Small set of rules can generate an infinite language.
  - Usually, base cases and recursive rules.
- Example languages:
  - Set of C++ keywords
  - Set of all C++ programs
  - Set of all well-formed PIN numbers
  - Set of all possible US telephone numbers
  - Contents of legal input files for a payroll program

Introduction

- In the English language, the grammar determines whether a combination of words is a legal sentence.
- Which of the following are *legal* sentences?
  - *The large rabbit hops quickly.*
  - *The frog writes neatly.*
  - *The swims mathematician quickly.*
- Grammars are concerned with the *syntax* (i.e., form) of a sentence, and NOT its *semantics* (i.e., meaning).

English Grammar

- ***sentence***: ***noun phrase*** ***verb phrase***.
- ***noun phrase***: ***article*** ***adjective*** ***noun*** or ***article*** ***noun***
- ***verb phrase***: ***verb adverb*** or ***verb***
- ***article***: *a* or *the*
- ***adjective***: *large* or *hungry*;
- ***noun***: *rabbit* or *mathematician* or *frog*
- ***verb***: *eats* or *hops* or *writes* or *swims*
- ***adverb***: *quickly* or *wildly* or *neatly*

Example

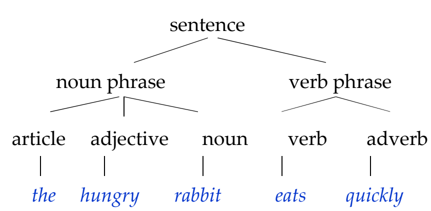
Use the grammar in generating sentences:

- Sentence
- Noun phrase verb phrase
- Article adjective noun verb phrase
- Article adjective noun verb adverb
- *the* adjective noun verb adverb
- *the large* noun verb adverb
- *the large rabbit* verb adverb
- *the large rabbit hops* adverb
- *the large rabbit hops quickly*

Also a sentence: *the large rabbit writes neatly*
(Even though probably untrue)

Example derivation tree

the hungry rabbit eats quickly



Alphabets, words & languages

- An *alphabet*, or *vocabulary*, V is a finite, nonempty set of *symbols*.
- A *word*, or *string*, over V is a finite sequence of symbols from V.
- The set of all words over V is denoted by V^*.
  - A word of length n, denoted $v_1v_2\ldots v_n \in V^*$, consists of n symbols, $n > 0$.
  - The *empty word*, consisting of 0 symbols, is denoted by $\lambda \in V^*$.
- Given words $v = v_1v_2\ldots v_n$ and $w = w_1w_2\ldots w_m$ of length n and m, the *concatenation* of v and w, denoted vw, is the word of length $n+m$ defined by: $vw = v_1v_2\ldots v_n w_1w_2\ldots w_m$
- By convention: $\lambda v = v\lambda = v$
- A *language* L over V is a subset of V^*, $L \subseteq V^*$.

Example: alphabets, words & languages

- {0, 1}* is the set of binary strings.
- Some example words over the alphabet (vocabulary) {0, 1}:
  λ 0000 1
- Some example concatenations of words from {0, 1}*:
  (0000)(1) = 00001
  λ (00) = 00
- Some example languages over the alphabet {0, 1}:
  $\{ v \mid v \in \{0, 1\}^* \text{ and } |v| < 3 \} = \{\lambda, 0, 1, 00, 01, 10, 11\}$
  $\{ 0^n1^m \mid n \le m \}_{m,\, n \in N} = \{\lambda, 1, 11, 01, 111, 011, 1111, 0111, 0011, \ldots\}$
  $\{ \lambda \}$

Grammars and derivations

- A *phrase-structure grammar*, denoted $G = (V,T,S,P)$, consists of
  - A vocabulary V
  - A *start symbol*, S, where $S \in V$
  - A set T of *terminal symbols*, where $T \subseteq V$
  - A finite set P of *production rules*:
    - $v \to w$, where v and w are strings over V
    - intuition: we can "rewrite" v as w in generating words that belong to a language

A vocabulary symbol in the set $V - T$ is called a *nonterminal symbol*

- To generate words using G
  - Start with the start symbol S
  - Using production rules, "rewrite" it until you have *derived* a string of only terminal symbols
  - The language of G, denoted $L(G)$, is the set of terminal strings that can be derived in this manner.

Example of a Grammar

- $G = (V, T, S, P)$, where
  - $V = \{a, b, A, B, S\}$
  - S is the start symbol
  - $T = \{a, b\}$ is the set of terminal symbols
  - $P = \{ S \rightarrow ABa,\ \ A \rightarrow BB,\ \ B \rightarrow ab,\ \ AB \rightarrow b,\ AB \rightarrow \lambda \}$ are the production rules
- Example *derivations*: $\boxed{S \Rightarrow^* abababa}$
  - S

Example of a Grammar

- $G = (V, T, S, P)$, where
  - $V = \{a, b, A, B, S\}$
  - S is the start symbol
  - $T = \{a, b\}$ is the set of terminal symbols
  - $P = \{ S \rightarrow ABa,\ \ A \rightarrow BB,\ \ B \rightarrow ab,\ \ AB \rightarrow b,\ AB \rightarrow \lambda \}$ are the production rules
- Example *derivations*: $\boxed{S \Rightarrow^* abababa}$
  - $S \Rightarrow ABa \Rightarrow BBBa \Rightarrow abBBa \Rightarrow ababBa \Rightarrow abababa$
  - $S \Rightarrow ABa$ $\boxed{S \Rightarrow^* ba}$
  - $S \Rightarrow ABa$ $\boxed{S \Rightarrow^* a}$

Example of a Grammar

- $G = (V, T, S, P)$, where
  - $V = \{a, b, A, B, S\}$
  - S is the start symbol
  - $T = \{a, b\}$ is the set of terminal symbols
  - $P = \{ S \rightarrow ABa,\ \ A \rightarrow BB,\ \ B \rightarrow ab,\ \ AB \rightarrow b,\ AB \rightarrow \lambda \}$ are the production rules
- Example *derivations*: $\boxed{S \Rightarrow^* abababa}$
  - $S \Rightarrow ABa \Rightarrow BBBa \Rightarrow abBBa \Rightarrow ababBa \Rightarrow abababa$
  - $S \Rightarrow ABa \Rightarrow ba$ $\boxed{S \Rightarrow^* ba}$
  - $S \Rightarrow ABa \Rightarrow \lambda\, a = a$ $\boxed{S \Rightarrow^* a}$
- Defn: The *derives relation*, denoted $\Rightarrow^*$, is the transitive closure of the *directly derives* relation, denoted $\Rightarrow$
- Defn: $L(G) = \{ w \mid w \in T^* \text{ and } S \Rightarrow^* w \}$
  $= \{ a, ba, ababa \}$

Exercise:

Let $G = (V, T, S, P)$, where
- The vocabulary is $V = \{ S, A, B, a, b \}$
- The start symbol is S
- The set of terminal symbols is $T = \{ a, b \}$
- P consists of the production rules
  - $S \rightarrow AB$
  - $A \rightarrow aAB$
  - $A \rightarrow \lambda$
  - $B \rightarrow bB$
  - $B \rightarrow b$
- Show that the following are in $L(G)$
  - b abb $aabbbb$
- How can you tell that the following are *not* in $L(G)$?
  - a ba aab
- What set of strings is $L(G)$?

Exercise

Write grammars that generate the following languages.
- All binary strings of length 5
  - Let $V = \{ S, D, 0, 1\}$, $T = \{ 0, 1\}$, start symbol be S,
    $P = \{\ S \rightarrow DDDDD,\ D \rightarrow 0,\ \ D \rightarrow 1 \}$
- All binary strings of length 5 that start with a 1.
  - Let $V = \{ S, D, 0, 1\}$, $T = \{ 0, 1\}$, start symbol be S,
    $P = \{\ S \rightarrow 1DDDD,\ D \rightarrow 0,\ \ D \rightarrow 1 \}$
- All binary strings (i.e., $\{0, 1\}^*$)
  - Let $V = \{ S, 0, 1\}$, $T = \{ 0, 1\}$, start symbol be S,
    $P = \{\ S \rightarrow \lambda,\ \ S \rightarrow 0S,\ \ S \rightarrow 1S \}$
- All binary strings of length 6 that have exactly two 1-bits in the first three bits
  - Let $V = \{ S, A, B, R, 0, 1\}$, $T = \{ 0, 1\}$, start symbol be S,
    $P = \{\ S \rightarrow AB,\ \ A \rightarrow 110,\ \ A \rightarrow 101,\ \ A \rightarrow 011,$
    $B \rightarrow RRR,\ \ R \rightarrow 0,\ \ R \rightarrow 1\}$

Types of Grammars

Grammars categorized by their production rules
- Type 0: No restriction on production rules
- Type 1 (*context-sensitive*): Every production rule has the form $x \rightarrow y$ where either $|x| \le |y|$ or $y = \lambda$
- Type 2 (*context-free*): Every production rule has the form $N \rightarrow y$ where $N \in V - T$ and $y \in V^*$
- Type 3 (*regular*): Every production rule has one of the following two forms
  - $N \rightarrow a\,M$ where $N, M \in V - T$ and $a \in T$, or
  - $N \rightarrow \lambda$ where $N \in V - T$

Observe: Type 3 $\subset$ Type 2 $\subset$ Type 1 $\subset$ Type 0

Example: Regular grammar (Type 3)

$G = (V, T, S, P)$, where
- Vocabulary $V = \{S, A, 0, 1\}$
- Terminals $T = \{0,1\}$
- Start symbol S
- Production rules:
  - $S \to 0S$
  - $S \to 1A$
  - $S \to \lambda$
  - $A \to 1A$
  - $A \to \lambda$

Type 3 (*regular*): Every production rule has one of the following two forms
1) $N \to a\,M$, or
2) $N \to \lambda$

where N, M are non-terminal symbols and a is a terminal symbol

- Exercise: What is $L(G)$?
  $L(G) = \{0^m1^n \mid m \text{ and } n \text{ are nonnegative integers}\}$

Example: Context Free (Type 2)

$G = (V,T,S,P)$, where
- Alphabet $V = \{S, 0, 1\}$
- Terminals $T = \{0,1\}$
- Start symbol S
- Production P:
  - $S \to 0S1$
  - $S \to \lambda$

Every production rule has the form
$N \to y$
where N is a non-terminal symbol and is y any string over the vocabulary

Important result of formal language theory: No regular grammar generates this language. (CSE 460)

- Exercise: What is $L(G)$?
  $L(G) = \{0^n1^n \mid n \text{ is nonnegative integer}\}$

Type 3 $\subset$ Type 2

Example: Context Sensitive (Type 1)

- $G = (V,T,S,P)$, where
  - Alphabet $V = \{S, A, B, 0, 1, 2\}$
  - Terminals $T = \{0,1, 2\}$
  - Start symbol S
  - Production P:
    - $S \to 0SAB$
    - $S \to \lambda$
    - $BA \to AB$
    - $0A \to 01$
    - $1A \to 11$
    - $1B \to 12$
    - $2B \to 22$
  - Exercise: What is $L(G)$?
    $L(G) = \{0^n1^n2^n \mid n \text{ is nonnegative integer}\}$

Type 1 (*context-sensitive*): Every production rule has the form
$x \to y$
where either $|x| \le |y|$ or $y = \lambda$

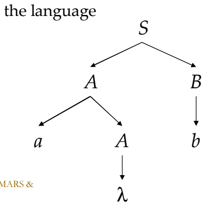
Important result of formal language theory: No context free grammar generates this language. (CSE 460)

Type 2 $\subset$ Type1

Parse Trees/Derivation Trees

- A derivation produced using a context-free grammar can be represented graphically by a *parse tree*, also called a *derivation tree*.
  - Root is labeled with the start symbol
  - Represent application of a production $N \to y$ by a node labeled N with children labeled by the symbols in y (in order)
  - All leaf nodes are labeled with a terminal or λ; when concatenated, they yield a word in the language

$\underline{S} \Rightarrow \underline{A}B \Rightarrow a\underline{A}B \Rightarrow a\lambda\underline{B} \Rightarrow a\lambda b = ab$



Application: Programming Languages (PLs)

- Backus-Naur form for describing syntax of a PL
  - Enclose nonterminals in angle brackets, e.g., <*e*>
  - Start symbol is on the left-hand-side of the first production rule
  - Use ::= in place of →
  - Abbreviate listing of productions for the same nonterminal:
    <*e*> ::= *w* | *x* | … | *z*
    is short for multiple productions
    <*e*> ::= *w* <*e*> ::= *x* … <*e*> ::= *z*
- Example: Backus-Naur form for identifiers (Algol 60)

```
<ident>              ::= <letter> <letters_or_digits>
<letters_or_digits>  ::= <letter> <letters_or_digits>
                       | <digit> <letters_or_digits>
                       | λ
<letter>             ::= a | b | c | … | z
<digit>              ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```


Application: Programming languages

- Regular grammars can describe the *tokens* (keywords, literals, operators, punctuation, etc.) of a programming language
- Example: The previous grammar for identifiers can be "rewritten" as a regular grammar

```
<ident>              ::= a <letters_or_digits> | b <letters_or_digits> | …
                       | z <letters_or_digits>
<letters_or_digits>  ::= a <letters_or_digits> | b <letters_or_digits> | …
                       | z <letters_or_digits>
                       | 0 <letters_or_digits> | 1 <letters_or_digits> | …
                       | 9 <letters_or_digits>
                       | λ
```

Application: Programming languages

- Context-free grammar can express much of the syntax of a programming language
- Example: Backus-Naur form for numeric expressions
  - <*e*> ::= <*e*> + <*t*> | <*e*> – <*t*> | <*t*>
  - <*t*> ::= <*t*> * <*f*> | <*t*> / <*f*> | <*f*>
  - <*f*> ::= **id** | **num** | **(** <*e*> **)**

Application: Compiler

Given a source program (string of ascii characters), a compiler:

- Tokenizes it using a finite state automaton (coming soon) constructed from a regular grammar
- Constructs a parse tree for the tokenized program using a push down automaton (CSE 460/450) constructed from a context free grammar
- Traverses the parse tree (an abstract version of it, called an *abstract syntax tree*) to generate object code

Parsing

- Given a grammar G with terminal alphabet T and a word w over T, either construct a parse tree generating w or, if none exists, *rejects* w
  - *Top-down parser*: start at the root (start symbol, S) and work down to the leaves; systematically try productions
    - Applying a production *expands* the nonterminal
    - So it attempts to expand nonterminals to rewrite S into w
  - *Bottom-up parser*: start at the leaves (w) and work back to the root; systematically try productions *in reverse*
    - Application of a production in reverse is called a *reduction*
    - So it attempts to reduce w to the start symbol

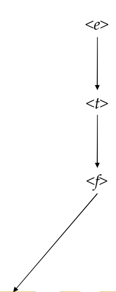
Example: Top down parsing

- Start at the root (start symbol) and work down to the leaves; systematically try productions.
- Example: Backus-Naur form for expression grammar
  - <*e*> ::= <*t*> | <*e*> + <*t*>
  - <*t*> ::= <*f*> | <*t*> * <*f*>
  - <*f*> ::= **id** | **num** | **(** <*e*> **)**

Example: Top down parsing

Grammar:
<*e*> ::= <*t*> | <*e*> + <*t*>
<*t*> ::= <*f*> | <*t*> * <*f*>
<*f*> ::= **id** | **num** | **(** <*e*> **)**

Try <*e*> ::= <*t*>
Try <*t*> ::= <*f*>
Try <*f*> ::= **id**



Fails: no non-terminals are left to expand.

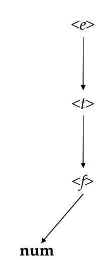
So try again.

Input string: **id** **+** **num** ***** **id**

Example: Top down parsing

Grammar:
<*e*> ::= <*t*> | <*e*> + <*t*>
<*t*> ::= <*f*> | <*t*> * <*f*>
<*f*> ::= **id** | **num** | **(** <*e*> **)**

Try <*e*> ::= <*t*>
Try <*t*> ::= <*f*>
~~Try <*f*> ::= **id**~~
Try <*f*> ::= **num**



Fails: terminals are incompatible.

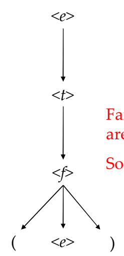
So try again.

Input string: **id** **+** **num** ***** **id**

Example: Top down parsing

Grammar:
<e> ::= <t> | <e> + <t>
<t> ::= <f> | <t> * <f>
<f> ::= **id** | **num** | (<e>)

Try <e> ::= <t>
Try <t> ::= <f>
~~Try <f> ::= **id**~~
~~Try <f> ::= **num**~~
Try <f> ::= (< e >)



Fails: terminals are incompatible.

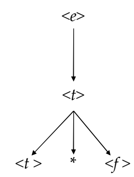
So try again.

Input string: **id** **+** **num** ***** **id**

Example: Top down parsing

Grammar:
<e> ::= <t> | <e> + <t>
<t> ::= <f> | <t> * <f>
<f> ::= **id** | **num** | (<e>)

Try <e> ::= <t>
~~Try <t> ::= <f>~~
~~Try <f> ::= **id**~~
~~Try <f> ::= **num**~~
~~Try <f> ::= (< e >)~~
Try <t> ::= <t> * <f>
And so on …

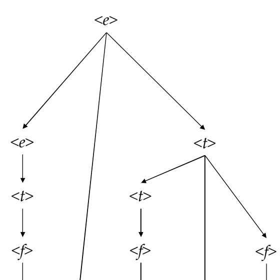


Input string: **id** **+** **num** ***** **id**

Example: Top down parse

Grammar:
<e> ::= <t> | <e> + <t>
<t> ::= <f> | <t> * <f>
<f> ::= **id** | **num** | (<e>)

Analysis of an "appropriate" grammar (CSE 450) produces a table that drives the expansion process based on the non-terminal to expand and the next input symbol. (No backing up.)



Input string: **id** **+** **num** ***** **id**

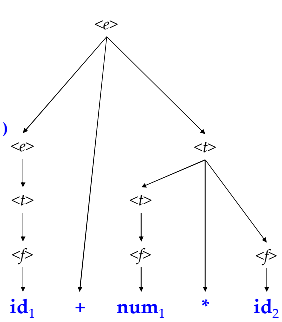
Example: Bottom up parsing

- Start at the leaves (input string) and work back to the root (start symbol); systematically try productions in reverse (reductions).
- Example: Backus-Naur form for expression grammar
  - <e> ::= <t> | <e> + <t>
  - <t> ::= <f> | <t> * <f>
  - <f> ::= **id** | **num** | (<e>)

Example: Bottom up parsing

- Example: Backus-Naur form for expression grammar
  - <e> ::= <t> | <e> + <t>
  - <t> ::= <f> | <t> * <f>
  - <f> ::= **id** | **num** | (<e>)
- Bottom up parse of **id**$_1$ **+ num**$_1$ *** id**$_2$

Analysis of an "appropriate" grammar (CSE 450) produces a FSA (coming soon) that drives the reduction process. Less intuitive, but more "powerful" than top-down parsing.



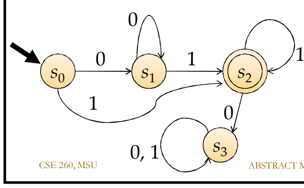
Grammars and Machines

- Each type of grammar corresponds to a *machine*, or *automaton*, that can parse the language of the grammar
- *Parsing* is a kind of computation
- Think of these machines as abstract implementations of the algorithms that perform the computations
- Machine types:
  - Type 0: Turing Machine
  - Type 1: Linear Bounded Automaton
  - Type 2: Push Down Automaton (PDA)
  - Type 3: Finite State Automaton (FSA)

Finite State Automaton (FSA)

An abstract machine that *recognizes* languages.

- An alphabet – input symbols
- A finite set of *states* – the FSA's memory
  - *Control* is in exactly one state, called the *current state.*
  - Has a *start state* and one or more *final states.*
- A finite set of *transitions* – indicate how *reading* an input symbol changes the FSA's current state.

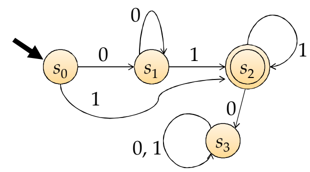


Input symbols: {0, 1}
States: $\{s_0, s_1, s_2, s_3\}$
Start state: s_0 Final state(s): s_2
Transitions:
$\{ (s_0, 0) \rightarrow s_1, (s_0, 1) \rightarrow s_2,$
$(s_1, 0) \rightarrow s_1, (s_1, 1) \rightarrow s_2,$
$\ldots (s_3, 1) \rightarrow s_3 \}$

Finite State Automaton (FSA)

How does an FSA recognize (accept) a string w ?

- Start with "control" in the start state, s_0
- "Read" symbols in w: follow the path through the FSA that spells out w
- If reading w leaves control in a final state, accept w; otherwise, reject w.

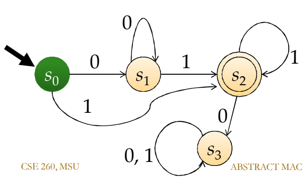


Finite State Automaton (FSA)

How does an FSA recognize (accept) a string w ?

- Start with "control" in the start state, s_0
- "Read" symbols in w: follow the path through the FSA that spells out w
- If reading w leaves control in a final state, accept w; otherwise, reject w.

Example: 00111



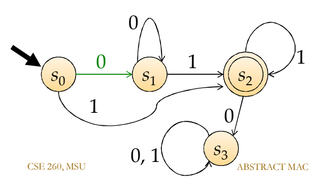
Example: 0011101

Finite State Automaton (FSA)

How does an FSA recognize (accept) a string w ?

- Start with "control" in the start state, s_0
- "Read" symbols in w: follow the path through the FSA that spells out w
- If reading w leaves control in a final state, accept w; otherwise, reject w.

Example: 00111



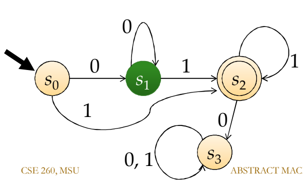
Example: 0011101

Finite State Automaton (FSA)

How does an FSA recognize (accept) a string w ?

- Start with "control" in the start state, s_0
- "Read" symbols in w: follow the path through the FSA that spells out w
- If reading w leaves control in a final state, accept w; otherwise, reject w.

Example: 00111



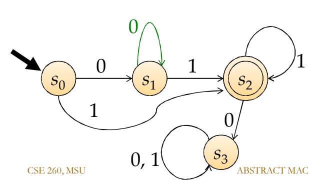
Example: 0011101

Finite State Automaton (FSA)

How does an FSA recognize (accept) a string w ?

- Start with "control" in the start state, s_0
- "Read" symbols in w: follow the path through the FSA that spells out w
- If reading w leaves control in a final state, accept w; otherwise, reject w.

Example: 00111



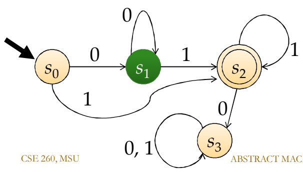
Example: 0011101

Finite State Automaton (FSA)

How does an FSA recognize (accept) a string w ?

- Start with "control" in the start state, s_0
- "Read" symbols in w: follow the path through the FSA that spells out w
- If reading w leaves control in a final state, accept w; otherwise, reject w.

Example: 00111



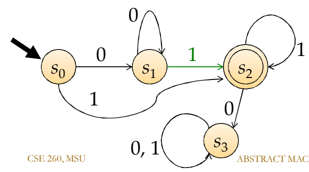
Example: 0011101

Finite State Automaton (FSA)

How does an FSA recognize (accept) a string w ?

- Start with "control" in the start state, s_0
- "Read" symbols in w: follow the path through the FSA that spells out w
- If reading w leaves control in a final state, accept w; otherwise, reject w.

Example: 00111



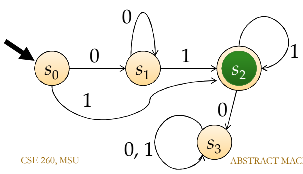
Example: 0011101

Finite State Automaton (FSA)

How does an FSA recognize (accept) a string w ?

- Start with "control" in the start state, s_0
- "Read" symbols in w: follow the path through the FSA that spells out w
- If reading w leaves control in a final state, accept w; otherwise, reject w.

Example: 00111



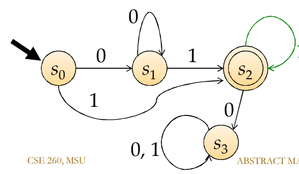
Example: 0011101

Finite State Automaton (FSA)

How does an FSA recognize (accept) a string w ?

- Start with "control" in the start state, s_0
- "Read" symbols in w: follow the path through the FSA that spells out w
- If reading w leaves control in a final state, accept w; otherwise, reject w.

Example: 00111



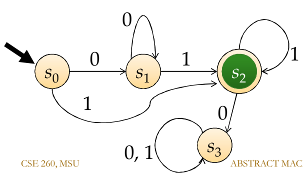
Example: 0011101

Finite State Automaton (FSA)

How does an FSA recognize (accept) a string w ?

- Start with "control" in the start state, s_0
- "Read" symbols in w: follow the path through the FSA that spells out w
- If reading w leaves control in a final state, accept w; otherwise, reject w.

Example: 00111



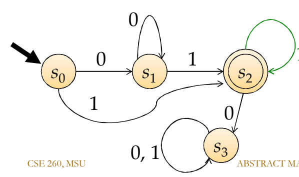
Example: 0011101

Finite State Automaton (FSA)

How does an FSA recognize (accept) a string w ?

- Start with "control" in the start state, s_0
- "Read" symbols in w: follow the path through the FSA that spells out w
- If reading w leaves control in a final state, accept w; otherwise, reject w.

Example: 00111



Example: 0011101

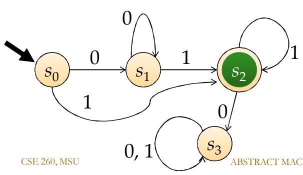
Finite State Automaton (FSA)

How does an FSA recognize (accept) a string w ?

- Start with "control" in the start state, s_0
- "Read" symbols in w: follow the path through the FSA that spells out w
- If reading w leaves control in a final state, accept w; otherwise, reject w.

Example: 00111

Accept!



Example: 0011101

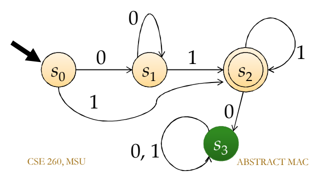
Finite State Automaton (FSA)

How does an FSA recognize (accept) a string w ?

- Start with "control" in the start state, s_0
- "Read" symbols in w: follow the path through the FSA that spells out w
- If reading w leaves control in a final state, accept w; otherwise, reject w.

Example: 00111

Accept!

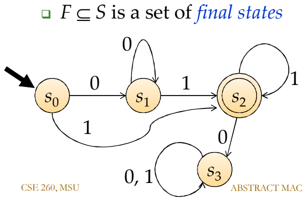


Example: 0011101

Reject!

Finite State Automaton

- A *finite-state automaton* $M = (S, I, f, s_0, F)$ where
  - S is a finite set of *states*
  - I is a finite *input alphabet*
  - f is a *transition function*, $f: S \times I \rightarrow S$, which assigns to each (state, input)-pair a next state
  - s_0 is an *initial (start) state*
  - $F \subseteq S$ is a set of *final states*

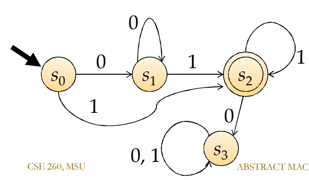


- $S = \{ s_0 , s_1 , s_2 , s_3 \}$
- $I = \{ 0, 1 \}$
- $f(s_0, 0) = s_1$ $f(s_0, 1) = s_2$
  $f(s_1, 0) = s_1$ $f(s_1, 1) = s_2$
  $f(s_2, 0) = s_3$ $f(s_2, 1) = s_2$
  $f(s_3, 0) = s_3$ $f(s_3, 1) = s_3$
- $F = \{s_2\}$

State transition table

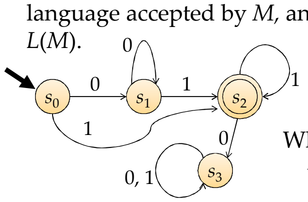
- Graphical representation of an FSA is convenient for us to read if the FSA is not too complex
- For computing, use a *state table*

S / I	0	1
s_0	s_1	s_2
s_1	s_1	s_2
s_2	s_3	s_2
s_3	s_3	s_3



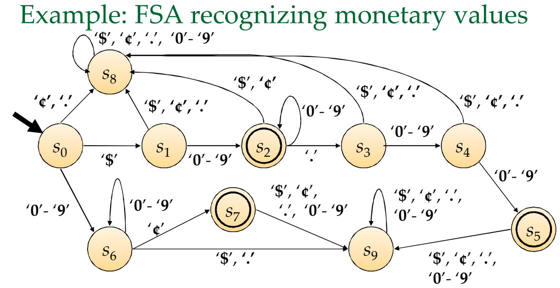
FSA Behavior

- A word w is said to be *recognized* (or *accepted*) by the FSA M if, starting at the start state, it reads w one symbol at a time, from left to right, and ends at some final state.
- The set of all words accepted by M is the language accepted by M, and is denoted by $L(M)$.



What is $L(M)$?
$\{ 0^n1^m \mid n \geq 1, m \geq 1 \}$

Example: FSA recognizing monetary values



- States: $s_0 , s_1 , \ldots , s_7$
- Alphabet: '\$', '¢', '.', '0'- '9'
- Start state: s_0
- Final states: $\{ s_2 , s_5 , s_7 \}$

$f(s_0, \text{'\$'}) = s_1, f(s_0, \text{'0'}) = s_6,$
$f(s_0, \text{'1'}) = s_6, f(s_0, \text{'2'}) = s_6, \ldots$
$f(s_1, \text{'0'}) = s_2, f(s_1, \text{'1'}) = s_2,$
$f(s_1, \text{'2'}) = s_2, \ldots$

State transition table

- Graphical representation of an FSA is convenient for us to read if the FSA is not too complex
- For computing, use a *state table*

State/ Input	'\$'	'¢'	'.'	'0'- '9'
s_0	s_1	s_8	s_8	s_6
s_1	s_8	s_8	s_8	s_2
s_2	s_8	s_8	s_3	s_2
…	…	…	…	…
s_8	s_8	s_8	s_8	s_8
…	…	…	…	…

Exercise: C++ identifiers

- In C++, an identifier should begin with a letter, which may be followed by any number of letters, digits or underscores ('_'). Draw an FSA that recognizes C++ identifiers.
- What is the input alphabet of your FSA?
- What are the final states of you FSA?
- What is the start state of your FSA?
- Create a state transition table for your FSA.

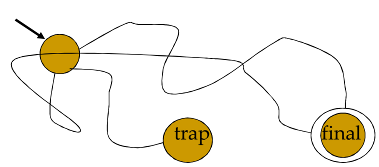
Nondeterministic FSA

- The FSA as described in your textbook is *deterministic*, since for each pair (state, input) there is a unique next state.
- A *nondeterministic FSA (NFA)* has a transition function that assigns a set of states to a (state, input) pair rather than just one
- Thus, an entry of the state table may list many or no next states

Nondeterministic FSA

- Also, an NFA may also have multiple start states.
- Nondeterminism in effect means that it may be necessary to follow multiple paths to determine if a word is accepted. (But can often use smaller automaton – space time tradeoff)

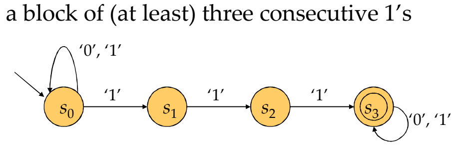
NFA M recognizes string w iff, when started in some start state, w drives the machine to some final state by some path.



There may be multiple paths that succeed and multiple paths that fail!

NFA: Example

- An NFA that recognizes binary strings containing a block of (at least) three consecutive 1's



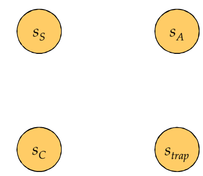
State/Input	'0'	'1'
s_0	s_0	s_0, s_1
s_1		s_2
s_2		s_3
s_3	s_3	s_3

NFAs recognize regular languages

- From a regular grammar, G, we can construct an NFA M that accepts exactly the words generated by the grammar – i.e., such that $L(G) = L(M)$
- Construction:
  - For each nonterminal N of G, introduce a state s_N; in addition, introduce a special *trap state* s_{trap}
  - For each nonterminal, N, and terminal, **a**,
    - for each production of the form $N \to \mathbf{a}M$, introduce a transition from s_N to s_M on input **a**;
    - if there is no transition $N \to \mathbf{a}M$, introduce a transition from s_N to s_{trap} on input **a**.
  - For each terminal , **a**,
    - introduce a transition from s_{trap} to s_{trap} on input **a**.
  - If S_0 is the start symbol of G, then make s_{S0}, the start state of M
  - Make s_N a final state iff there is a production of the form $N \to \lambda$

Example:

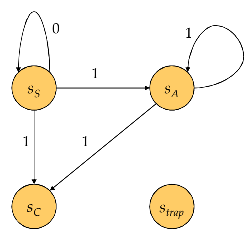
$G = (V,T,S,P)$, where
- $V = \{S, A, C, 0, 1\}$
- $T = \{0,1\}$
- P contains the rules:
  - $S \to 0S$
  - $S \to 1A$
  - $S \to 1C$
  - $A \to 1A$
  - $A \to 1C$
  - $S \to \lambda$
  - $C \to \lambda$



For each nonterminal N of G, introduce a state s_N; in addition, introduce a special *trap state* s_{trap}

Example:

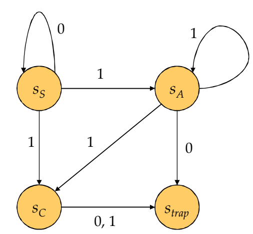
$G = (V,T,S,P)$, where
- $V = \{S, A, C, 0, 1\}$
- $T = \{0,1\}$
- P contains the rules:
  - $S \to 0S$
  - $S \to 1A$
  - $S \to 1C$
  - $A \to 1A$
  - $A \to 1C$
  - $S \to \lambda$
  - $C \to \lambda$



for each production of the form $N \to \mathbf{a}M$, introduce a transition from s_N to s_M on input **a**

Example:

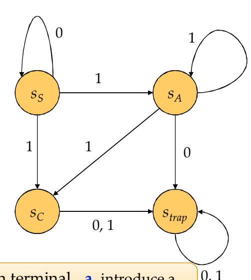
$G = (V,T,S,P)$, where
- $V = \{S, A, C, 0, 1\}$
- $T = \{0,1\}$
- P contains the rules:
  - $S \to 0S$
  - $S \to 1A$
  - $S \to 1C$
  - $A \to 1A$
  - $A \to 1C$
  - $S \to \lambda$
  - $C \to \lambda$



if there is no transition $N \to \mathbf{a}M$, introduce a transition from s_N to s_{trap} on input **a**.

Example:

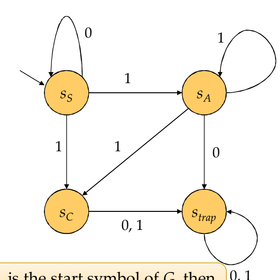
$G = (V,T,S,P)$, where
- $V = \{S, A, C, 0, 1\}$
- $T = \{0,1\}$
- P contains the rules:
  - $S \to 0S$
  - $S \to 1A$
  - $S \to 1C$
  - $A \to 1A$
  - $A \to 1C$
  - $S \to \lambda$
  - $C \to \lambda$



For each terminal , **a**, introduce a transition from s_{trap} to s_{trap} on input **a**

Example:

$G = (V,T,S,P)$, where
- $V = \{S, A, C, 0, 1\}$
- $T = \{0,1\}$
- P contains the rules:
  - $S \to 0S$
  - $S \to 1A$
  - $S \to 1C$
  - $A \to 1A$
  - $A \to 1C$
  - $S \to \lambda$
  - $C \to \lambda$

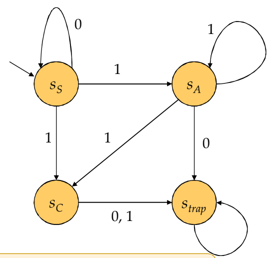


If S_0 is the start symbol of G, then make s_{S0}, the start state of M

Example:

$G = (V,T,S,P)$, where

- $V = \{S, A, C, 0, 1\}$
- $T = \{0,1\}$
- P contains the rules:
  - $S \to 0S$
  - $S \to 1A$
  - $S \to 1C$
  - $A \to 1A$
  - $A \to 1C$
  - $S \to \lambda$
  - $C \to \lambda$

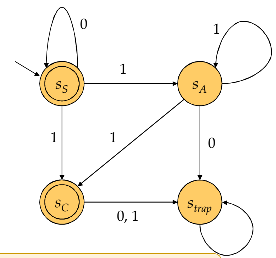


Make s_N a final state iff there is a production of the form $N \to \lambda$

Example:

$G = (V,T,S,P)$, where

- $V = \{S, A, C, 0, 1\}$
- $T = \{0,1\}$
- P contains the rules:
  - $S \to 0S$
  - $S \to 1A$
  - $S \to 1C$
  - $A \to 1A$
  - $A \to 1C$
  - $S \to \lambda$
  - $C \to \lambda$
- $L(G) = \{0^m 1^n \mid m, n \geq 0\}$



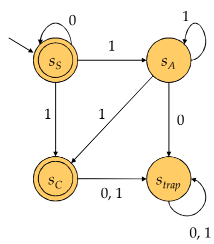
Make s_N a final state iff there is a production of the form $N \to \lambda$

Equivalence of NFAs & FSAs

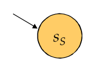
- Every NFA is *equivalent* to a (deterministic) FSA, in the sense that the NFA and the FSA recognize the same language.
- The technique for *determinizing* a NFA is an important computational technique.
- Idea:
  - Sets of states of the NFA become the states of the FSA. (An FSA state "remembers" what states of the NFA could be the current state.)
  - Group states of the NFA, as needed, to make the automaton deterministic.

Example: Determinizing a NFA

Given NFA:



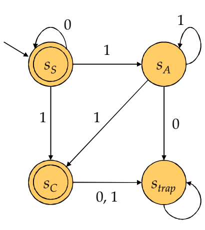
To grow the FSA, start off with a copy of the NFA's initial state to serve as the initial state of the FSA.



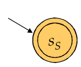
Because s_S is a final state of the NFA, make the copy of s_S a final state of the FSA.

Example: Determinizing a NFA

Given NFA:



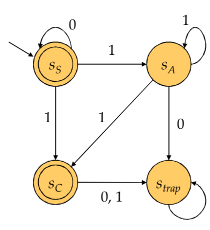
To grow the FSA, start off with a copy of the NFA's initial state to serve as the initial state of the FSA.



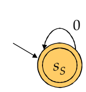
Because s_S is a final state of the NFA, make the copy of s_S a final state of the FSA.

Example: Determinizing a NFA

Given NFA:

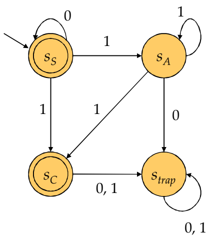


Reading 0 takes the NFA from s_S to (only) s_S, so add a transition in the FSA from the start state to itself on input 0.

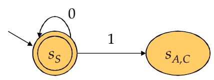


Example: Determinizing a NFA

Given ND FSA:



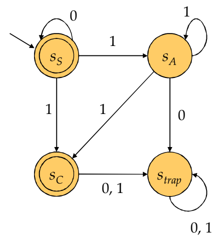
Reading 1 takes the NFA from s_S to either s_A *or* s_C so add a state $s_{A,C}$ to the FSA, which "remembers" that the current state of the NFA could be either s_A *or* s_C, and a transition from s_S to $s_{A,C}$ on 1



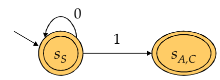
Because s_C is a final state of the NFA, $s_{A,C}$ is a final state of the FSA.

Example: Determinizing a NFA

Given ND FSA:



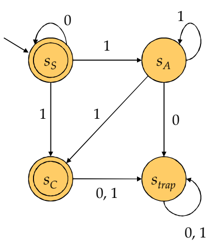
Reading 1 takes the NFAfrom s_S to either s_A *or* s_C so add a state $s_{A,C}$ to the FSA, which "remembers" that the current state of the NFA could be either s_A *or* s_C, and a transition from s_S to $s_{A,C}$ on 1



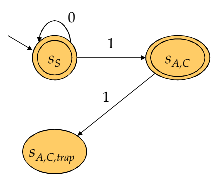
Because s_C is a final state of the NFA, $s_{A,C}$ is a final state of the FSA.

Example: Determinizing a NFA

Given ND FSA:



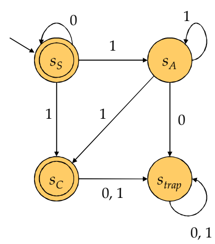
Reading 1 from s_A *or* s_C takes the NFA to s_A *or* s_C *or* s_{trap}, so add a state $s_{A,C,trap}$ to the FSA and a transition from $s_{A,C}$ to $s_{A,C,trap}$ on 1.



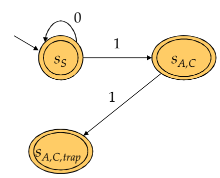
Because s_C is a final state of the NFA, $s_{A,C,trap}$ is a final state of the FSA.

Example: Determinizing a NFA

Given ND FSA:



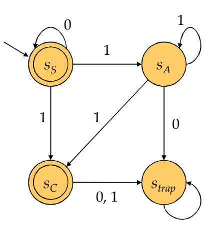
Reading 1 from s_A *or* s_C takes the NFA to s_A *or* s_C *or* s_{trap}, so add a state $s_{A,C,trap}$ to the FSA and a transition from $s_{A,C}$ to $s_{A,C,trap}$ on 1.



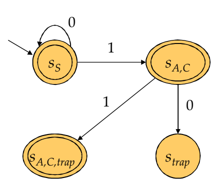
Because s_C is a final state of the NFA, $s_{A,C,trap}$ is a final state of the FSA.

Example: Determinizing a NFA

Given ND FSA:



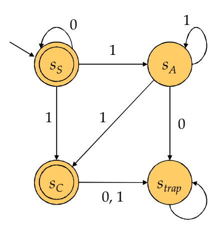
Reading 0 from s_A *or* s_C takes the NFA to (only) s_{trap}, so add a state s_{trap} to the FSA and a transition from $s_{A,C}$ to $s_{,trap}$ on 0.



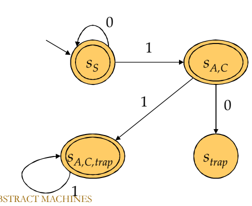
s_{trap} is not a final state of the FSA since s_{trap} is not a final state of the NFA

Example: Determinizing a NFA

Given NFA:

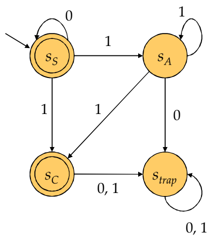


Reading 1 from s_A *or* s_C *or* s_{trap} takes the NFA to s_A *or* s_C *or* s_{trap}, so add a transition from $s_{A,C,trap}$ to $s_{A,C,trap}$ on 1.

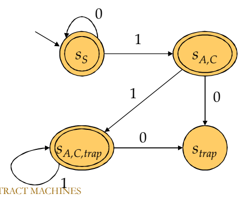


Example: Determinizing a NFA

Given NFA:

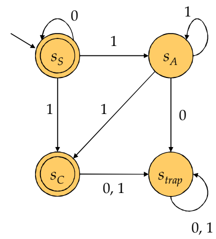


Reading 0 from s_A *or* s_C *or* s_{trap} takes the NFA to (only) s_{trap}, so add a transition from $s_{A,C,trap}$ to s_{trap} on 0.

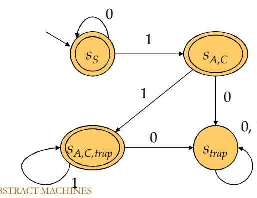


Example: Determinizing a NFA

Given NFA:

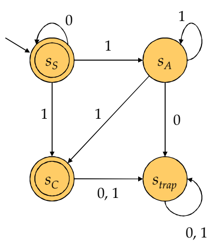


Reading 0 (or 1) from s_{trap} takes the NFA to (only) s_{trap}, so add a transition from s_{trap} to s_{trap} on 0 (resp. 1) .

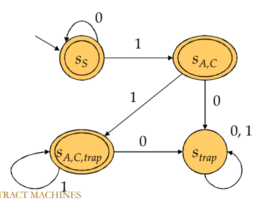


Example: Determinizing a NFA

Given NDNFA:



Equivalent FSA is therefore:

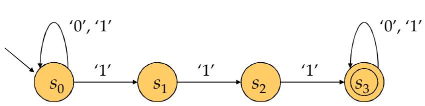


Key points

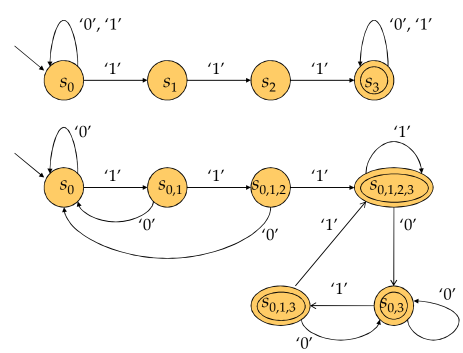
- If the NFA has n states, then the DFSA could have as many as 2^n states.
- All possible combined states are considered.
- Thus, it can be recorded when some path succeeds through some states.

Exercise: Determinize the NFA

- A NFA to recognize binary strings containing a block of three consecutive 1's



Exercise: Determinize the NFA



Application: Programming languages

- Recall that the tokens of a programming language can be described using a regular grammar
- Thus, the tokens of a programming language can be recognized by a NFA
- This NFA can be determinized to yield a FSA that recognizes the tokens of the programming language
- A *lexical analyzer generator* (a program that generates a scanner for use by a compiler) essentially implements these algorithms

Deterministic v.s. Nondeterministic FSAs

- What are the space/time tradeoffs?

Chomsky Heirarchy

The language types form a *strict* hierarchy: