# From PDAs to Turing Machines

April 11, 2011

CS 81: Computability and Logic

# Practical Parsing

Recursive Descent

- ✓ Form of code follows the grammar.
- ✓ Efficient and correct for LL grammars.

Another very practical method: Shift-Reduce Parsing

- ✓ Efficient and correct for LR grammars
- ✓ Parser code is usually computer-generated!

But neither of these handle ALL CFGs…

# CYK (CKY) Algorithm

- ✓ Works for any CFG.
- ✓ Parses inputs in $O(n^3)$ ($n =$ length of input)
- ✓ Requires a grammar in "Chomsky Normal Form"
- ✓ Key ideas:
  - For every substring, what nonterminals produce it?
  - Dynamic programming for efficiency

# Dynamic Programming

Recursive expressions, such as

$$
\begin{array}{lcll}
f(n) & := & 1 & \text{if } n < 3 \\
f(n) & := & f(n-1) + f(n-3) & \text{otherwise}
\end{array}
$$

are very clear, but inefficient if taken literally. Two roughly-equivalent solutions:

- ✓ Memoization: keep track of what f values have been computed
- ✓ Dynamic Programming: Compute all f values in a good order

# CYK Algorithm

Let

$$x = x_1 x_2 \cdots x_n$$

be the string to be parsed.

Define

$$a(i,j) := \{\, B \mid B \Rightarrow^* x_i x_{i+1} \cdots x_j \,\}$$

Then $x \in L(G)$ iff $S \in a(1,n)$

# CYK Algorithm

Let

$$x = x_1 x_2 \cdots x_n$$

be the string to be parsed.

Define

$$a(i,j) := \{\, B \mid B \Rightarrow^* x_i x_{i+1} \cdots x_j \,\}$$

Then $x \in L(G)$ iff $S \in a(1,n)$

$$\begin{array}{rcl} a(i,i) & = & \{\, C \mid C \to x_i \,\} \\ a(i,k) & = & \{\, C \mid C \to AB \wedge A \in a(i,j) \wedge B \in a(j+1,k) \,\} \end{array}$$

# CYK "Wavefront" Matrix

| | | | | | | |
|---|---|---|---|---|---|---|
| a(1, 1) | a(1, 2) | a(1, 3) | a(1, 4) | | a(1, n-1) | a(1, n) |
| | a(2, 2) | a(2, 3) | a(2, 4) | | a(2, n-1) | a(2, n) |
| | | a(3, 3) | a(3, 4) | | a(3, n-1) | a(3, n) |
| | | | a(4, 4) | | | |
| | | | | | | |
| | | | | | a(n-1, n-1) | a(n-1, n) |
| | | | | | | a(n, n) |

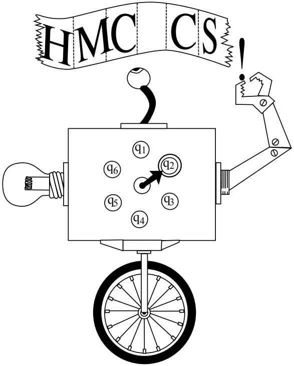
Each entry is computed from entries in its same row and column, e.g.
a(1, 4) from
a(1,1) and a(2, 4),
a(1, 2) and a(3, 4),
a(1, 3) and a(4, 4).

# CYK Example

Let's parse `(()())` using the grammar

$$
\begin{aligned}
S &\rightarrow LT \\
T &\rightarrow SR \\
S &\rightarrow LR \\
S &\rightarrow SS \\
L &\rightarrow ( \\
R &\rightarrow )
\end{aligned}
$$

# Turing Machines



# Turing Machines

- ✓ Named after (not by) Alan Turing
- ✓ Perhaps the most important computational model (if not the most practical)
- ✓ Simple, yet apparently universal
- ✓ Church-Turing Thesis (a.k.a. Church's Thesis)
  - Any "intuitively computable" procedure can be performed by a TM

# Official Definition

A Deterministic TM consists of

- ✓ A finite set $Q$ of control states
- ✓ A finite alphabet $\Sigma$
- ✓ A finite "tape alphabet" $\Gamma$ $\quad (\Sigma \subset \Gamma, \ \sqcup \in \Gamma \setminus \Sigma)$
- ✓ A starting state $q_0 \in Q$
- ✓ Accepting/Rejecting (halting) states $q_{accept}, q_{reject} \in Q$
- ✓ Transitions $\qquad \delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$

# Running a Turing Machine

- ✓ Write the finite input at the start of an infinite blank tape
- ✓ Run the TM, beginning at the start of the tape

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$$

- ✓ TM halts iff we enter states $q_{accept}$ or $q_{reject}$.
  - TM might never halt!
  - Sipser: also reject if you fall off the infinite tape

# Beyond Decision Problems

Rather than accepting a language, we can use a TM to compute a function $f(x) = y$:

- ✓ The machine starts with some input $x$ on the tape
- ✓ The machine halts (however) with some string $y$ on the tape.
- ✓ If the machine diverges (does not halt) on some inputs, it computes a *partial function*.

# TM Demo

http://ironphoenix.org/tril/tm/

# Configurations

An instantanous snapshot of a TM is called a *configuration*

- ✓ The "state" of the "whole machine"
- ✓ Contents?
- ✓ Why are configurations always finite?

# TMs and Languages

- ✓ A TM **accepts** a string if that input leads to $\mathbf{q}_{accept}$.

# TMs and Languages

- ✓ A TM **accepts** a string if that input leads to $q_{accept}$.
- ✓ A language is **recognizable** (a.k.a. recursively enumerable) if there is a TM that accepts exactly these strings.

# TMs and Languages

- ✓ A TM **accepts** a string if that input leads to $q_{accept}$.
- ✓ A language is **recognizable** (a.k.a. recursively enumerable) if there is a TM that accepts exactly these strings.
- ✓ A language is **decidable** (a.k.a. recursive) if it is accepted by a TM that always halts
(i.e., the TM always ends up in $q_{accept}$ or $q_{reject}$).

# TMs and Languages

- A TM **accepts** a string if that input leads to $q_{accept}$.
- A language is **recognizable** (a.k.a. recursively enumerable) if there is a TM that accepts exactly these strings.
- A language is **decidable** (a.k.a. recursive) if it is accepted by a TM that always halts (i.e., the TM always ends up in $q_{accept}$ or $q_{reject}$).
- Key result: there are languages that are accepted by some TM, but not decided by any TM.

# Decidable vs. Recognizable

- ✓ If a language is decidable, then its complement is decidable. Why?

# Decidable vs. Recognizable

- ✓ If a language is decidable, then its complement is decidable. Why?
- ✓ If a language is recognizable, and its complement is recognizable, then the language is decidable. Why?

# TM Programming Tips

- ✓ Divide the work into different phases/subroutines
- ✓ Controller has an arbitrarily large "finite memory".
- ✓ Squares can be "marked" and "unmarked" (finitely many ways)
- ✓ Take advantage of TM extensions

# TM Variations

The following yield no extra power:

- ✓ Adding the option to write or not
- ✓ Adding the option to stay-in-place rather than moving L/R.
- ✓ Making the tape infinite in both directions
- ✓ Adding an extra "Erase Tape" move.
- ✓ Multiple tapes with multiple (independent) read/write heads
- ✓ 2-D infinite tape
- ✓ Nondeterminism (!)

Many attempts to define models of computation; all turn out to be equivalent to Turing Machines.

- ✓ If you can do it in C++, a TM can do it (slowly, encodedly)