From PDAs to Turing Machines

November 16, 2011

CS 81: Computability & 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"
  - All rules of the form $A \to a$ or $A \to BC$.
  - Any CFG can be put into this form
    - Handle the input ϵ separately
    - You may or may not be happy with the new parse tree
- ✓ 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 $S \in a(1, n)$

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

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)

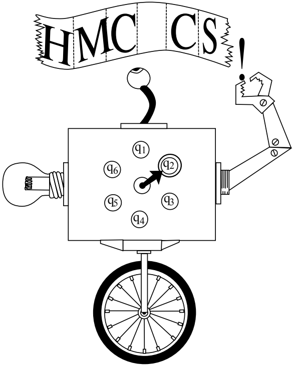
Each entry is computed from entries in its same row and column, e.g.
a(1, 4) from
a(1,1) and a(2, 4),
a(1, 2) and a(3, 4),
a(1, 3) and a(4, 4).

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 K of control states
- ✓ A finite alphabet Σ
- ✓ A finite "tape alphabet" Γ $\quad (\Sigma \subset \Gamma,\ \sqcup \in \Gamma \setminus \Sigma)$
- ✓ A starting state $s \in Q$
- ✓ Halting states $H = \{y, n\} \subseteq K$
- ✓ Transitions taken from

$$\big((Q \setminus H) \times \Gamma\big) \times \big(Q \times \Gamma \times \{L, R\}\big)$$

Running a Turing Machine

- ✓ Write the finite input in the middle of an infinite blank tape
- ✓ Position the
- ✓ Run the TM

$$K \times \Gamma \quad \rightarrow \quad K \times \Gamma \times \{L, R\}$$

- ✓ TM halts iff we enter a halting state "$\mathfrak{y}$" or "$\mathfrak{n}$".
  - The TM *accepts* the input if it ends in $\mathfrak{y}$.
  - The TM *rejects* the input if it ends in $\mathfrak{n}$.
  - TM might never halt!

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/

Question

How is my laptop more like a Finite State Machine than like a Turing Machine?

How is my laptop more like a Turing Machine than like a Finite State Machine?

Configurations

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

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

Universal Turing Machines

UTMs can be shown to exist by constructing them.

Think about what would be required.

- ✓ The tape has to hold the tape of the machine being simulated.
- ✓ The tape has to hold the program of the machine being simulated.
- ✓ The tape has to hold the current state of the machine being simulated.

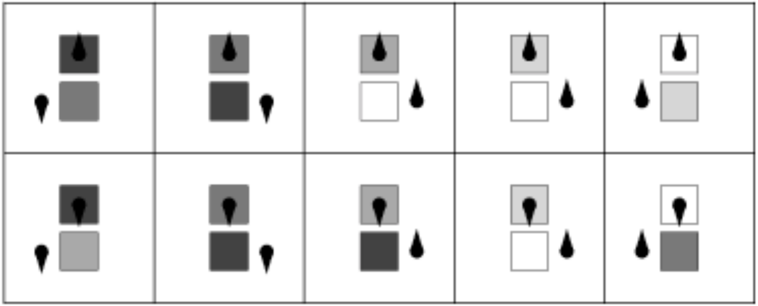
All this is possible, if somewhat laborious to construct.

Specific UTMs

- ✓ The first was constructed by Turing himself.
- ✓ Shannon showed any UTM could be converted either to a 2-symbol machine or to a 2-state machine (with lots more states or symbols, respectively).
- ✓ Minsky (1960) gave a 7-state 6-symbol machine.
- ✓ Watanabe (1961) gave an 8-state 5-symbol machine.
- ✓ Minsky (1962) gave a 7-state 4-symbol machine.
- ✓ Rogozhin (1996) gave a 4-state 6-symbol machine
- ✓ Wolfram and Reed (2002) gave a 2-state 5-symbol machine.
- ✓ Smith and Wolfram (2007) gave a 2-state 3-symbol machine.
- ✓ No 2-state 2-symbol UTM exists.

A Specific UTM

2-state, 5 symbol UTM published by Wolfram in 2002



adapted from Wolfram, S. *A New Kind of Science*. Wolfram Media, p. 707, 2002.

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* on each step.
- ✓ 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 Prolog or Python or C++, a TM can do it (slowly)

TMs and Languages

- ✓ A TM **accepts** a string if that input leads to $\mathfrak{y}$.
- ✓ A language is **semidecidable** (a.k.a. recognizable, recursively enumerable) if there is a TM that accepts exactly the strings in the language.
- ✓ 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 $\mathfrak{y}$ or $\mathfrak{n}$).

Decidable vs. Semidecidable

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

Turing Machines as Enumerators

Several variant definitions. Each specify a language $\mathbf{L}$.

1. A TM that prints out all the members of L, *one at a time* (but not necessarily in any particular order)
2. A TM that prints out all the members of L, *one at at time* (but...) with arbitrarily many repeats.
3. A TM that, *given an integer* n, returns the nth *element of a* sequence like (1) *above*.
4. A TM that, *given an integer* n, returns the nth *element of a* sequence like (2) *above*.

For a fixed language, all these are interconvertable.

Theorem

A language is semidecidable $\iff$ *it can be enumerated.*

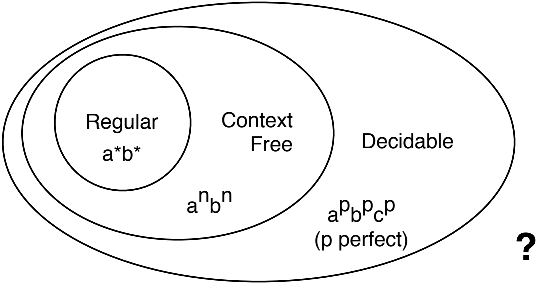
Church-Turing Thesis

If it can be done at all, then it can be done by

- ✓ A Turing Machine
- ✓ Lambda Calculus
- ✓ An Unrestricted Grammar
- ✓ A 2-register machine
- ✓ C
- ✓ ...

(Note: assumes suitably coded inputs and outputs)

Is There More?



Some Languages Aren't Decidable

Given a finite Σ, how many strings are there?

Given a finite Σ, how many languages are there?

How many TMs are there?

QED

But Wait...

How many languages over Σ could I describe (say, in a LaTeX document)?

How many TMs are there?

QED?