Formal Languages and Computational Models

Programming Languages

CS 214

© David Meyer All Rights Reserved.

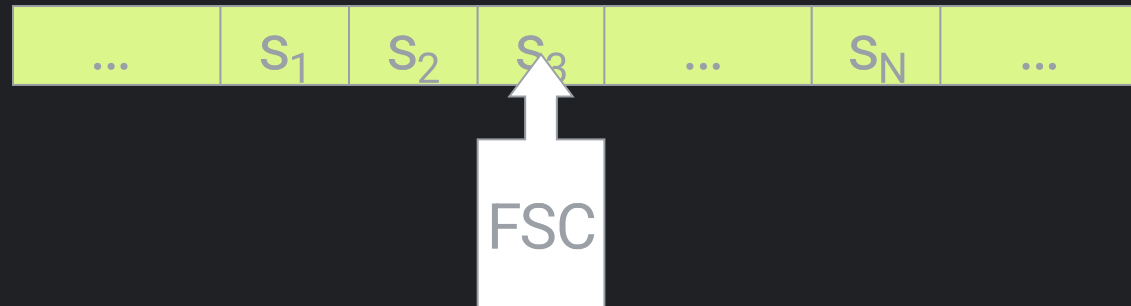
I read a news story the other day about a dog who ran 10 miles to retrieve a stick for his owner.

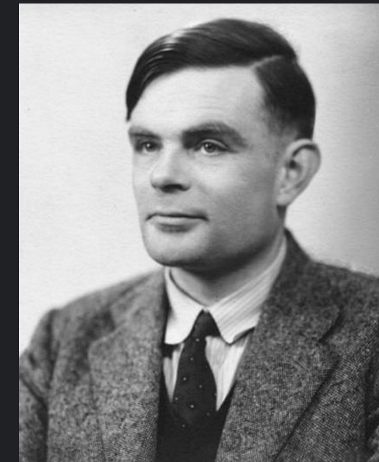
It sounded pretty far-fetched.

Turing Machines

In 1936 (years before the first programmable computer), *Alan Turing* created a model for the process of computation known today as the *Turing Machine* (TM), consisting of:

- An *I/O tape* consisting of an arbitrary number of *cells*, each able to store an arbitrary symbol;
- A *tape head* able to read/write a cell; and
- A *finite-state control* that governs movement of the head over the cells.

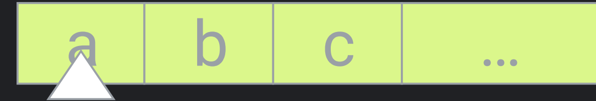




© David Meyer All Rights Reserved.

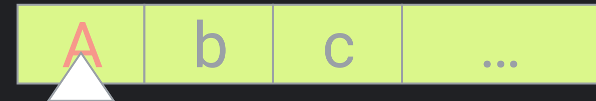
Turing Machines (ii)

Each "execution cycle", a TM reads a *symbol* from the tape.



Depending on that *symbol* and its current **state**, it may then:

- **write a symbol to the tape**;
- **move its head left or right**; and
- **change to a new state**.



The finite state controller starts in state 0: the *start state*, and continues execution until it enters an *accept state*, at which point it halts and its I/O tape contains the result of the computation.

© David Meyer All Rights Reserved.

Example: TM Addition

To add two numbers *m* and *n*:

- Precond: I/O tape contains *m* ones, a zero, and *n* ones.
- Postcond: I/O tape contains *m+n* ones.

Our finite state controller uses these states and rules:

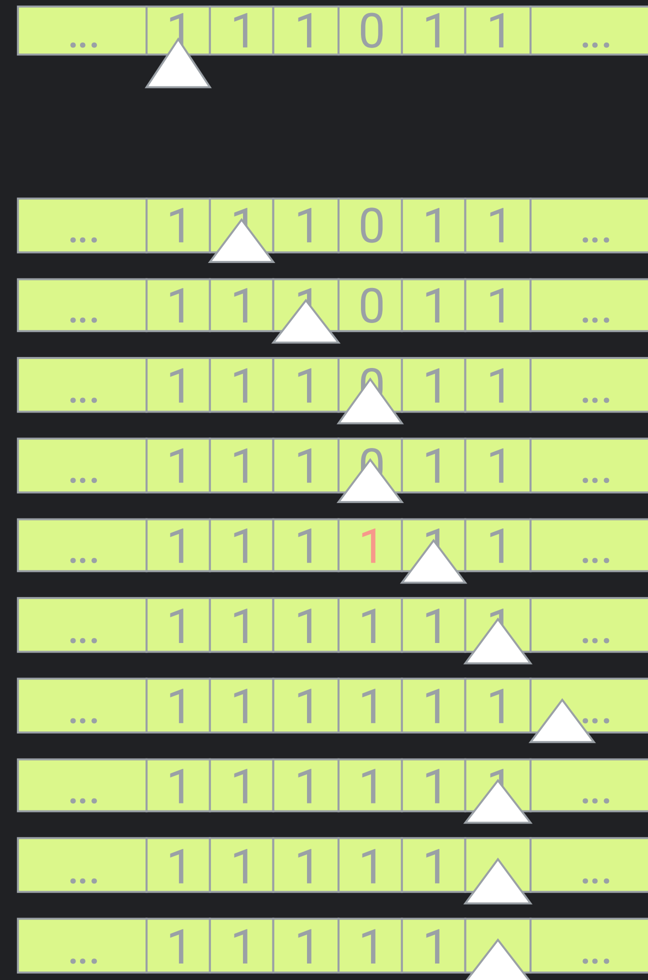
State 0:	If *symbol* is 1 or blank: move head right; goto State 0. If *symbol* is 0: goto State 1
State 1:	Write 1; move head right; goto State 2.
State 2:	If *symbol* is 1: move head right; goto State 2. If *symbol* is blank: move head left; goto State 3
State 3:	Write blank; goto State 4.
State 4:	Accept.

© David Meyer All Rights Reserved.

Example: 3 + 2

To compute 3 + 2, we start with:

Step	$State_i$	Read	Write	Move	$State_j$
1	0	1	-	right	0
2	0	1	-	right	0
3	0	1	-	right	0
4	0	0	-	-	1
5	1	-	1	right	2
6	2	1	-	right	2
7	2	1	-	right	2
8	2	blank	-	left	3
9	3	-	blank	-	4
10	4	-	-	-	-



© David Meyer All Rights Reserved.

TMs and Computability

In 1931, *Kurt Gödell* proved that there exist easily-described functions that cannot be computed.

In 1936, Turing proved that a TM can be built for any computable function.

He later proved that a *universal TM* can be built that can perform the task of any single-function TM, implying:

→ Since it is independent of any particular hardware details, a proof about a UTM applies to <u>every</u> computer that will <u>ever</u> be built!

→ If a function *f* can be computed, then a UTM can compute *f*.

→ If a UTM cannot compute a function *g*,
then *g* cannot be computed (by any computer, ever).

Turing proved the *Halting Problem* cannot be solved by a UTM.

© David Meyer All Rights Reserved.

My neighbor is a cop, and she has a son who refused to take his afternoon nap. She became so exasperated, she finally took him to jail and had him locked him up …

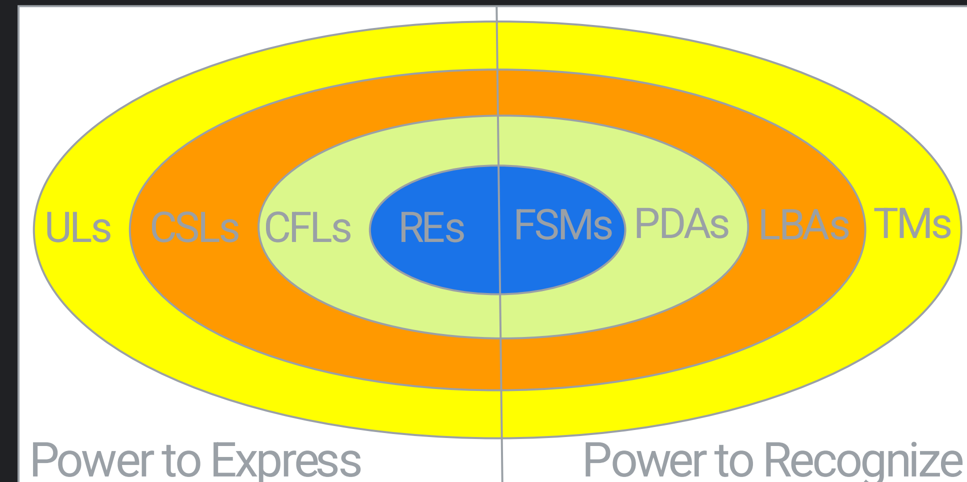
… for resisting a rest!”

The Chomsky Hierarchy

In 1956, *Noam Chomsky* classified languages as follows:

Level	Language	Recognizer
3	regular expressions (REs)	finite state machine (FSM)
2	context free (CFLs)	pushdown automata (PDA)
1	context sensitive (CSLs)	linear bounded automata (LBA)
0	unrestricted (ULs)	Turing machine (TM)

Chomsky's categories form a *hierarchy*, organized by their power of expression (language) and power of recognition (automaton):



© David Meyer All Rights Reserved.

The Chomsky Hierarchy: Level 3

"Regular Expression" languages can take only the form of

$a \rightarrow B\ c \mid B\ d \mid B\ \varepsilon$

Lowercase : terminals.
Uppercase : non-terminals
ε:end

Any non terminal can lead at most to one terminal and one of several non-terminals

"The" must be followed by [Cat Dog Bird Person Thing]

Cat must be followed by...

A finite state machine (FSM), with no memory, can understand this language, but we are very limited by what we can express.

© David Meyer All Rights Reserved.

The Chomsky Hierarchy: Level 2

Context Free languages can take only the form of

a → d B c | B d | B ε

Lowercase : terminals.
Uppercase : non-terminals
ε:end

Non-terminals can map to any pattern of terminals and non-terminals, in this way non-terminals can induce syntactic forms

<preposition> → IF <conjecture> THEN <result>

A **pushdown automata (PDA),** with a single stack, can understand this language. All programming languages and most spoken language falls within this realm.

 © David Meyer All Rights Reserved.

The Chomsky Hierarchy: Level 1

Context Sensitive languages can take only the form of

faH → d B c | B d | B ε

Lowercase : terminals.
Uppercase : non-terminals
ε:end

Non-terminals **in context, defined by both terminals and non-terminals** can map to any pattern of terminals and non-terminals, in this way non-terminals can recognize and induce syntactic forms

A **linear bounded automata (LBA)** with limited memory, can understand this language. Some of spoken languages more sophisticated structures require these sorts of rules

© David Meyer All Rights Reserved.

The Chomsky Hierarchy: Level 0

Unrestricted languages can take any form

$? \rightarrow ?|\ \varepsilon$

Lowercase : terminals.
Uppercase : non-terminals
ε:end

The only rule applied to the rules of an unlimited language is that the left side of the production cannot be empty. The rules for these languages are often too complex for us to make use of them

A Turing machine (TM) with unlimited memory, is needed to understand this language.

 © David Meyer All Rights Reserved.

Chomsky and BNFs

The *Chomsky Hierarchy* specifies that:

- A TM can recognize any language able to be recognized.
- A LBA can recognize CSLs, CFLs, & REs but not ULs.
- A PDA can recognize CFLs & REs but not CSLs or ULs.
- A FSM can recognize REs but not CFLs, CSLs or ULs.

The BNF is a tool for specifying CFL syntax.

- Programming language syntax is relatively “easy”, linquistically.

It can also be used to specify RE syntax (but doing so is overkill -- simpler tools are available).

Different tools are needed to specify CFL and/or UL syntax.

© David Meyer All Rights Reserved.

What chord played when the piano fell down the mine shaft?

A-flat minor!

The Random Access Machine (RAM)

Proving things about TMs was a bit clumsy…

1963: *Shepherdson and Sturgis* devise the RAM as a model that is equivalent to a TM but more convenient to use…

The RAM has four components

- A *memory*: an integer array, indexed from zero.
- A *program*: a sequence of numbered instructions.
- An *input file*.
- An *output file*.

Shepherdson and Sturgis proved a RAM can compute anything a UTM can compute, and vice versa.

© David Meyer All Rights Reserved.

The RAM Instruction Set

- `M[i] = n` → `store n at index i`
- `M[i] = M[j]` → `copy value at j to i`
- `M[i] = M[j] + M[k]` → `add and store`
- `M[i] = M[j] - M[k]` → `subtract and store`
- `M[M[j]] = M[k]` → `indirection`
- `read M[i]` → `input (destructive)`
- `write M[i]` → `output`
- `goto s` → `unconditional branch`
- `if M[i] >= 0 goto s` → `conditional branch`
- `halt` → `terminate execution`

Later extensions added other operators (arithmetic, relational)

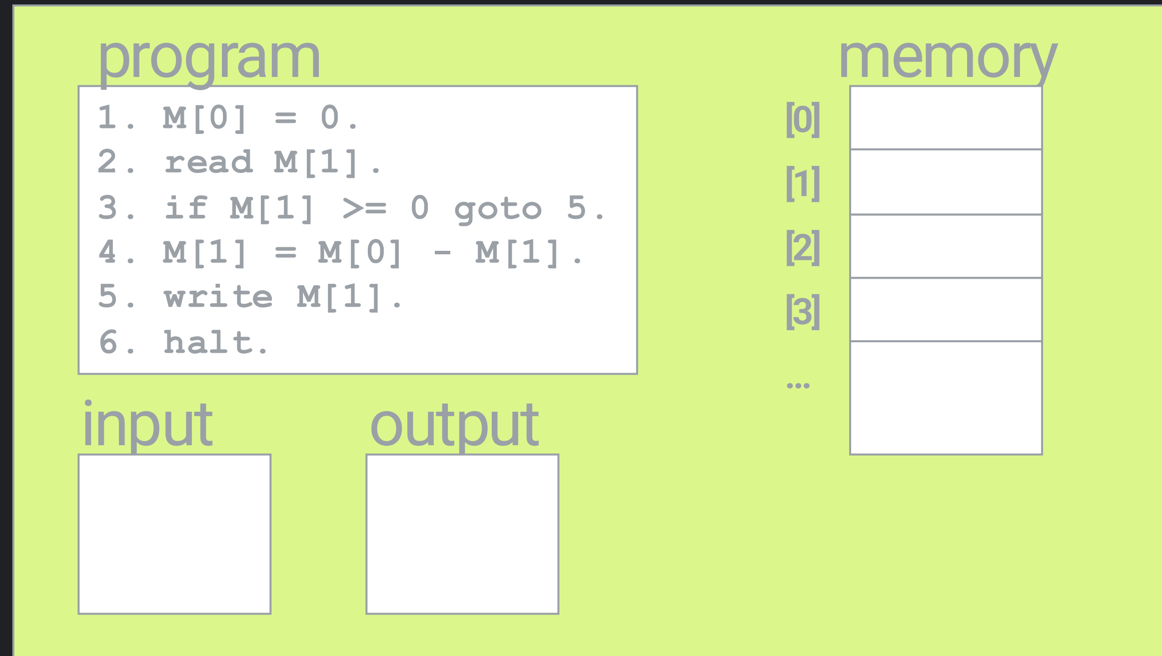
The result was quite similar to a *RISC assembly language*.

© David Meyer All Rights Reserved.

Example 1

Here is a RAM for a computation...

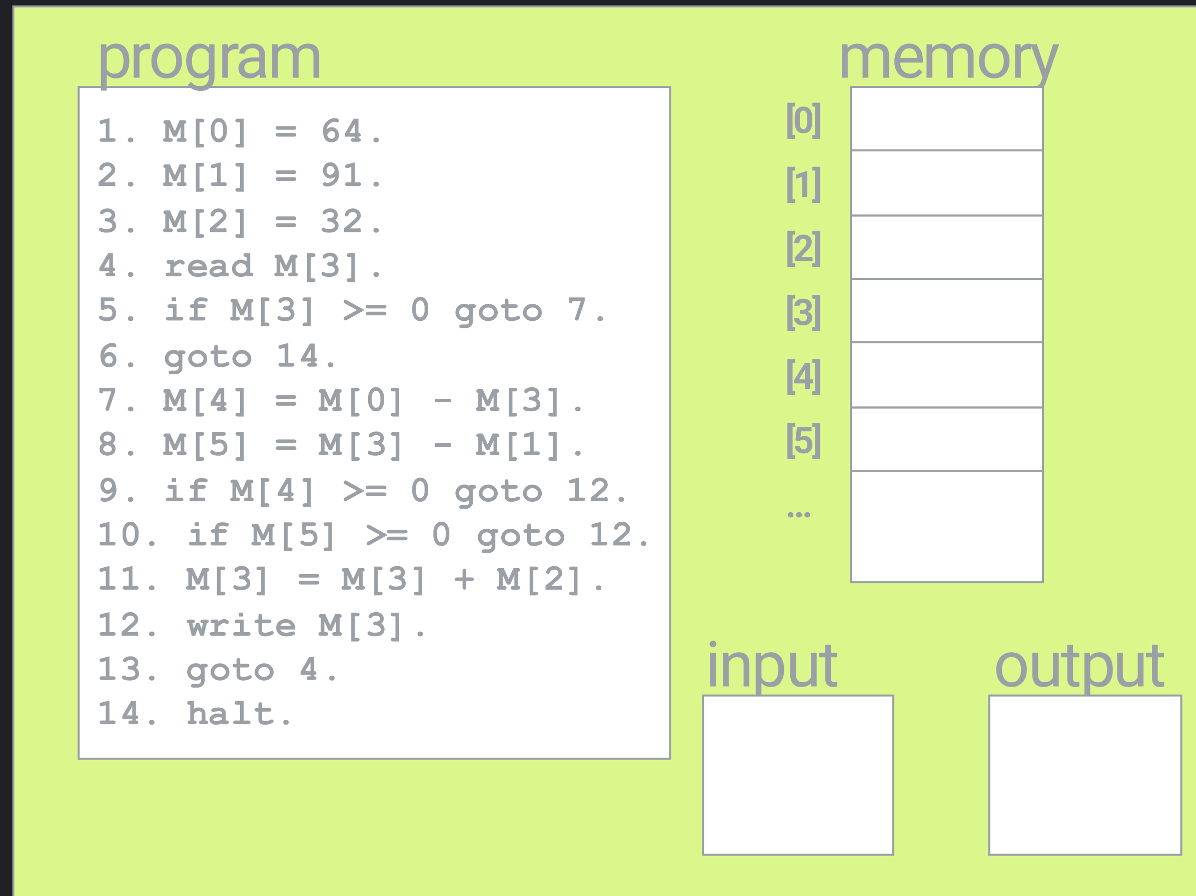
What does it do?



© David Meyer All Rights Reserved.

Example 2

Here is a different RAM. What does it compute?



© David Meyer All Rights Reserved.

RAM Extensions

Like a TM, a RAM can compute anything that is computable.

With these simple extensions:

- **symbolic names** instead of memory locations
- **multiplication** and **division** operators
- other relational **(==, !=, <, >, <=)** operators
- **literals** within arithmetic expressions

it becomes a convenient tool for studying HLL constructs, as a “portable assembly language” to study how a compiler can translate HLL constructs.

© David Meyer All Rights Reserved.

RAM Extension Examples

Example 1 program

```
1. read val.
2. if val >= 0 goto 4.
3. val = 0 - val.
4. write val.
5. halt.
```

Example 2 program

```
1. read ch.
2. if ch < 0 goto 10.
3. lo = ch - 65.
4. hi = ch - 90
5. if lo < 0 goto 8.
6. if hi > 0 goto 8.
7. ch = ch + 32.
8. write ch.
9. goto 1.
10. halt.
```

Even with the improvements, such programs are hard to read because of their coding style (*spaghetti code*), just as assembly language is harder to read than a HLL...

© David Meyer All Rights Reserved.

Summary

The **Chomsky Hierarchy** names four "levels" of language, plus the weakest machine able to recognize at each level:

3 Regular Expressions	→ Finite State Machine
2 Context Free Languages	→ Pushdown Automata
1 Context Sensitive Languages	→ Linear Bounded Automata
0 Unrestricted Languages	→ Turing Machine

The **TM** is the most powerful of the machines, able to
- recognize any language capable of being recognized.
- compute any function capable of being computed.

The **RAM** is a computational model that is
- as powerful as the TM
- more convenient than the TM for studying HLL constructs.

© David Meyer All Rights Reserved.

What is a Closure

A Clojure is a form in programming where we can use the declaration of a function to maintain access to values in its scope at the time of declaration.

```
(defn greeting-builder [name]
(fn [message]
(println (str message ", " name))))

(def hello-world (greeting-builder "World!"))
(hello-world "Hello") ;
 Output: Hello, World!
```

This inverts the way that scope is calculated to prevent items from being garbage collected!

© David Meyer All Rights Reserved.

A Buddhist monk approaches a hot dog stand. The vendor asks, “What do you want on your hot dog?” The monk answers, …

… “Make me one with everything.”

The monk pays for his hotdog with a $20 bill. The vendor says, “Thank you. Next customer!” The monk says, “Hey, what about my change?” The vendor replies, …

… “Change must come from within!”