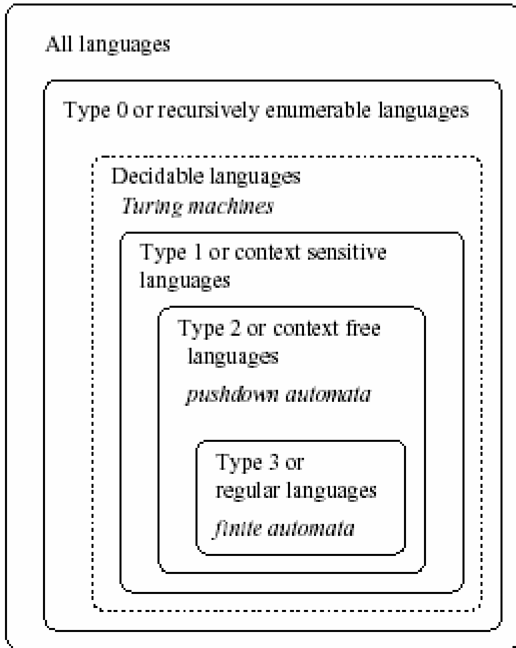
Revision Notes

The Chomsky Hierarchy



A language is a set of words.
A word is a sequence of symbols.
The symbols are given by an alphabet, Σ.

The set of all languages is given by P (Σ*)

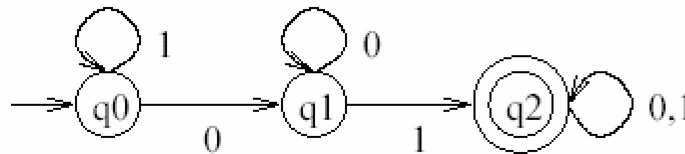
Type 3 (Regular) Languages

Deterministic Finite Automata

The machine which recognizes Regular languages. Equivalent to a computer with finite (fixed) memory.
A DFA, $A = (Q, \Sigma, \delta, q_0, F)$ where
- Q is a finite set of states
- Σ is the alphabet
- δ is the transition function $\delta \in Q \times \Sigma \rightarrow Q$
- q_0 is the initial state
- F is the finite set of final states

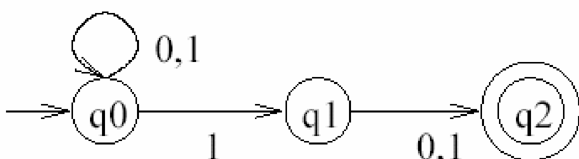
A typical transition sequence looks like this:

$$\begin{aligned}\delta(q_0, 101) &= \delta(\delta(q_0,1), 01)\\ &= \delta(q_0, 01)\\ &= \delta(\delta(q_0, 0), 1)\\ &= \delta(q_1, 1)\\ &= \delta(\delta(q_1, 1), \varepsilon)\\ &= \delta(q_2, \varepsilon)\\ &= q_2\end{aligned}$$



In a deterministic *finite automaton, there is always precisely one transition for each alphabet symbol in each state.*

Nondeterministic Finite Automata



This is very similar to a DFA, except that there can be multiple (or zero) transitions for a symbol in a state. A

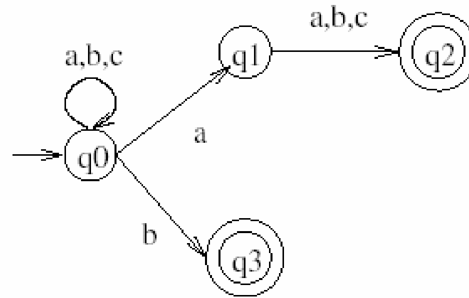
word is accepted if there exists a path through the automaton that ends in a final state.

Constructing a DFA from an NFA

To do this, we use the subset construction.
Write down a table, adding any new states to the bottom. Then work through from the initial state – some states will not be reachable.

E.g.,

	a	b	c
> {q0}	{q0, q1}	{q0, q3}	{q0}
{q1}	{q2}	{q2}	{q2}
{q2}	{}	{}	{}
***{q3}**	{}	{}	{}
{q0, q1}	{q0, q1, q2}	{q0, q3, q2}	{q0, q2}
***{q0, q3}**	{q0, q1}	{q0, q3}	{q0}
{q0, q1, q2}	{q0, q1, q2}	{q0, q3, q2}	{q0, q2}
***{q0, q3, q2}**	{q0, q1}	{q0, q3}	{q0}
{q0, q2}	{q0, q1}	{q0, q3}	{q0}

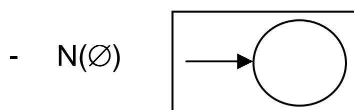


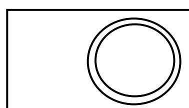
Regular Expressions

Symbol precedence: * (repetition) is strongest, then sequence, then + (or).
I.e., ab* = a (b)* and ab + cd = (ab) + (cd)

Examples:
- Words which contain a 'b'
  (a + b + c)* b (a + b + c)*
- Words which do not contain a b.
  (a + c) *
- Words where all as appear before all the cs
  (a + b)* (b + c) *
- Words s.t. the number of as plus the number of bs is odd.
  c*(a + b)c* ((a + b) c * (a + b) c*) *
- Words which contain the sequence aa.
  (a + b + c) * aa (a + b + c) *
- Words which do not contain the sequence aa.
  (b + c)* (a (b + c) (b + c)*)* (a + ε)

Deriving an NFA from a Regular Expressions

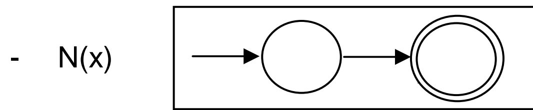
- N(∅) 



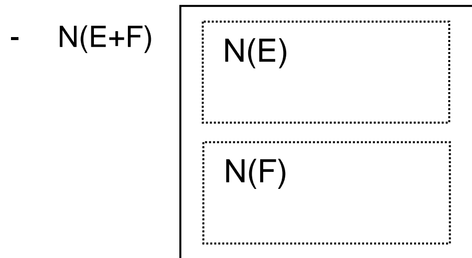
Revision Notes

- N(ε) ⟶

- N(x)

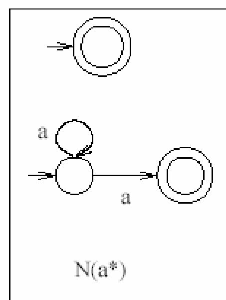


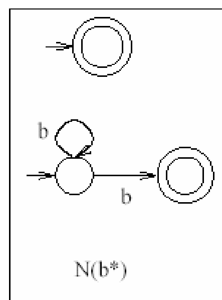
- N(E+F)

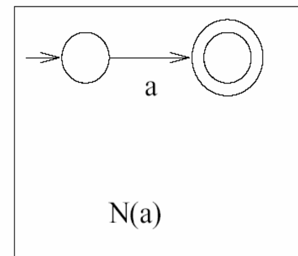


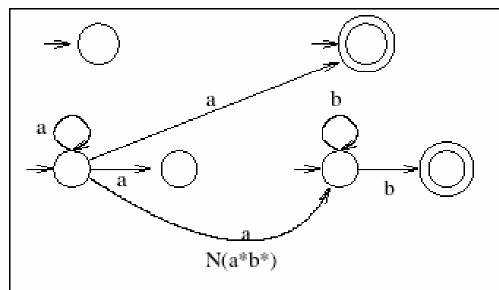
- N(EF): The initial states of E are the initial states of EF
  The final states of F are the final states of EF
  Connect the penultimate states of E to the initial states of F.

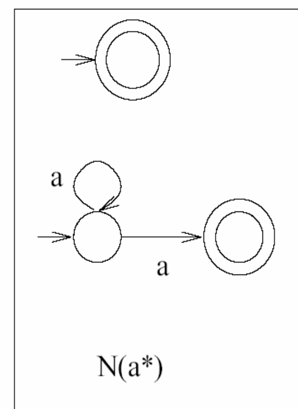
E.g.,











- N(E*): - Link penultimate states to initial states
  - Add an initial state which is final

Showing a Language is not Regular

Revision Notes

Regular languages can be recognized by a computer with finite (fixed) memory. Such a computer corresponds to a DFA.

The language 0^n1^n cannot be recognized using a finite memory – n can be arbitrarily large.

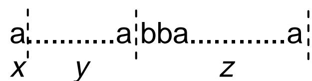
The *Pumming Lemma* can prove a language is/is not regular.

- Given a regular language, L, there is a number $n \in N$ such that all words, $w \in L$ which are longer than n ($|w| >= n$) can be split into three words $w = xyz$ such that:
  - $y \neq \varepsilon$
  - $|xy| <= n$
  - $\forall k \in N . xy^kz \in L$

E.g.,
$L_{pali} = \{wxw^R \mid w \in \Sigma^*, x \in \Sigma\} \cup \{ww^R \mid w \in \Sigma^*\}$ where w^R is the word w backwards.

Proposition: L_{pali} is not regular
Proof: Assume L_{pali} is regular and derive a contradiction.

Let $w = a^nbba^n \in L$
$|w| >= n$ – True.



Remember: $|xy| <= n$.

Consider $xyyyz = aa^{n-1}a^{n-1}a^{n-1}bba^n$ which is plainly not in L_{pali}.
Contradiction – hence, L_{pali} is not regular.

Type 2 (Context-free) Languages

Context Free Grammars

A context-free grammar, $G = (V, \Sigma, S, P)$ where:

- V is the finite set of variables or nonterminal symbols
- Σ is the set of terminal symbols (we assume V and Σ are disjoint)
- S is the start symbol $\in V$
- P is the finite set of productions: $P \subseteq V \times (V \cup T)^*$

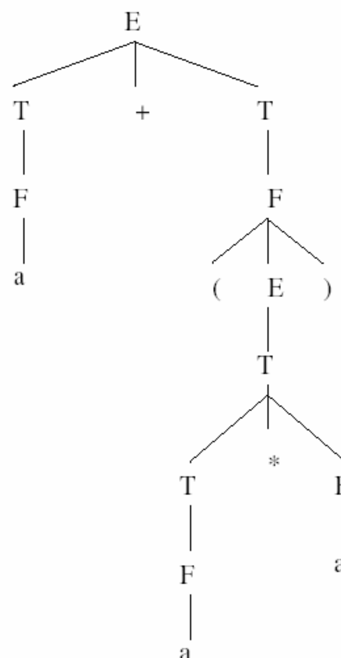
E.g., G = (V = {E, T, F}
Σ = {a,+,*,(,)}
S = E
P = { E → T | E + T
T → F | T * F
F → a | (E) }

Let's try and form a + (a * a):

E → E + T
→ T + T
→ F + T
→ a + T
→ a + F
→ a + (E)
→ a + (T)
→ a + (T * F)
→ a + (F * F)
→ a + (a * F)
→ a + (a * a)

Parse Trees

A graphical representation of the derivation above:



A grammar is said to be ambiguous a word has more than one parse tree.

Adding symbol precedence to a grammar

Example:

G = ({E, A}, {p, q, r, (,), ∨, ∧, ¬, true, false} with P given by:
E → A | ¬E | E ∧ E | E ∨ E | (E)
A → p | q | r | true | false

This is an ambiguous grammar. Suggest an alternative grammar for the same language which is not ambiguous. The grammar should reflect the following conventions on how to read boolean expressions:

- ¬ binds stronger than ∧ and ∨
- ∧ binds stronger than ∨

Add in two more intermediary nonterminals so we have one for each level of precedence. The first (lowest) level of precedence is ∨, the next is ∧, the next is ¬, the next is (), p, q, r, true, false.

The modified grammar is now:

E → F | E ∨ F
F → L | F ∧ L
L → A | ¬L
A → (E) | p | q | r | true | false

N.B. Each nonterminal either goes to the same symbol connected to the next symbol | just the next symbol.

Pushdown Automata

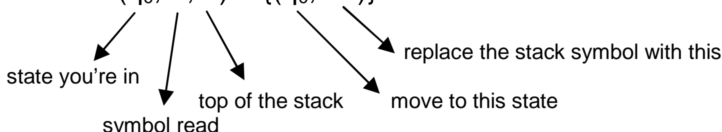
The machine which recognizes Context-free languages. Uses a stack memory (LIFO).

A Pushdown automata, $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ where

Q is a finite set of states
Σ is the alphabet
Γ is the stack symbols
δ is the tranisition function $Q, \Sigma, \Gamma, \delta \in Q \times (E \cup \{\varepsilon\}) \times \Gamma \rightarrow P_{fin}(Q \times \Gamma^*)$
q_0 is the initial state
Z_0 is the initial stack symbol
F is the finite set of final states

A typical transition looks like this:

$\delta(q_0, 0, \#) = \{(q_0, 0\#)\}$



PDAs can accept by final state *or* by empty stack (in which case F is omitted from the definition).

Determinism – a PDA is deterministic is there is never a choice of transition. It may get stuck. NPDAs are more powerful than DPDAs.

Turning a grammar into a PDA

Remember this grammar: G = ({E, T, F}
Σ = {a,+,*,(,)}
S = E
P = { E → T | E + T
T → F | T * F
F → a | (E) }

We can define the PDA, P(G) = ({q_0}, {(,), a,+, *}, {E, T, F, (,), a,+, *}, δ, q_0,E) with:
δ (q0, ε,E) = {(q_0, T), (q_0, E + T)}
δ (q0, ε, T) = {(q_0, F), (q_0, T * F)}
δ (q0, ε, F) = {(q_0, a), (q_0, (E))}
δ (q0, (, () = {(q_0, ε)}
δ (q0,),)) = {(q_0, ε)}
δ (q0, a, a) = {(q_0, ε)}
δ (q0, +, +) = {(q_0, ε)}
δ (q0, *, *) = {(q_0, ε)}

LL(1) Parsers

In order to be unambiguous, a parser must have only one choice at each transition stage. The class of grammars where this is possible is called LL(1).
For each non-terminal symbol, A, we define First (A) which is the set of terminals that A can start with, and Follow (A) which is the set of terminals that can come immediately after A.
We then define a Lookahead set for each production. A grammar is LL(1) if each lookahead set is different.
E.g.
The grammar P is given by:
Prog → {Stmts}
Stmts → ε | Stmt Stmts
Stmt → Name = Expr;
| if (Expr) Stmt
| while (Expr) Stmt
| print Expr ;
| ;
| Prog
Expr → Name | Num | (Expr Op Expr)
Name → x | y | z
Op → + | * | - | div
Num → 0 | 1

First(Prog) = { { }
First(Stmts) = First(Stmt) = {x, y, z, if, while, print, ;, {}
First(Expr) = {x, y, z, 0, 1, (}
First(Name) = {x, y, z}
First(Op) = {+, *, -, div}
First(Num) = {0, 1}

Follow(Prog) = {$, }, x, y, z, if, while, print, ;, {}
Follow(Stmts) = {}}
Follow(Stmt) = First (Stmts) = { }, x, y, z, if, while, print, ;, { }
Follow(Expr) = {;,), +, *, -, div}
Follow(Name) = {;,), +, *, -, div, =}
Follow(Op) = First(Expr) = {x, y, z, 0, 1, (}
Follow(Num) = First(Expr) = {x, y, z, 0, 1,)}

Revision Notes

Lookahead(Prog → {Stmts}) = {{}
Lookahead(Stmts → ε) = {}}
Lookahead(Stmts → Stmt Stmts) = {x, y, z, if, while, print, ;, {}
Lookahead(Stmt → Name = Expr;) = {x, y, z}
Lookahead(Stmt → if (Expr) Stmt) = {if}
Lookahead(Stmt → while (Expr) Stmt) = {while}
Lookahead(Stmt → print Expr ;) = {print}
Lookahead(Stmt → ;) = {;}
Lookahead(Stmt → Prog) = {{}
Lookahead(Expr → Name) = {x, y, z}
Lookahead(Expr → Num) = {0, 1}
Lookahead(Expr → (Expr Op Expr)) = {(}
Lookahead(Name → x) = {x}
Lookahead(Name → y) = {y}
Lookahead(Name → z) = {z}
Lookahead(Op → +) = {+}
Lookahead(Op → *) = {*}
Lookahead(Op → -) = {-}
Lookahead(Op → div) = {div}
Lookahead(Num → 0) = {0}
Lookahead(Num → 1) = {1}

Yes, the grammar is LL(1).

Type 0 (Recursively Enumerable) Languages

Turing Machines

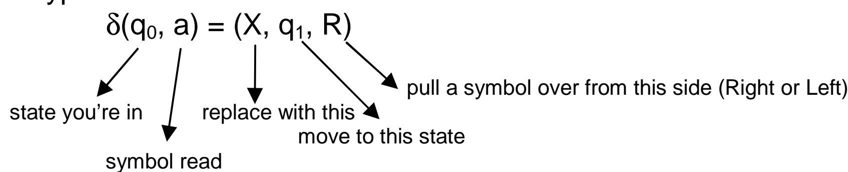
Like a PDA, but uses an infinite *tape* not a *stack*.
A language is *Recursive* or *Decidable* if there is a Turing Machine which will always stop.

A Turing Machine M = (Q, Σ, Γ, δ, q_0, B, F) where

- Q is the finite set of states
- Σ is the alphabet
- Γ is the set of tape symbols
- δ is the transition function: $\delta \in Q \times \Gamma \rightarrow \{stop\} \cup Q \times \Gamma \times \{L,R\}$
- q_0 is the initial state
- B $\in \Gamma$ is the blank tape symbol
- F is the set of final states

A typical transition looks like this:

$\delta(q_0, a) = (X, q_1, R)$



For example - (X,q_0, aYbZc)
You're in q_0, and read an 'a' (always read from the right).
Replace this a with an X: (X,q_0, XYbZc)
Go into q_1: (X,q_1, XYbZc)
Then pull a symbol over from the right: (XX,q_0,YbZc)

A grammar is said to be context-sensitive (type 1) if for any production, the left hand side is at least as long as the right hand side.