2 Parsing For The Lexicographic Unit
2 Language For Lexical Unit-RA,RG,FSA, For General Parsing CFG, AG
2 Lexical Unit- Three ways of Defining LanguageRG(a) , FSA(b), RE (c)

3 Finite State Automata (FSA)-Definitions
3 The FSA-State Diagram:
3 The Language Defined By an FSA - The accepted Input Sequences
3 Ambiguity

4 Designing FSAs- Examples
4 Problem: Design an d-FSA which accepts all strings of *0*'s and *1*'s that contain two or more successive 1's.
4 Problem: Design a d-FSA, M2, which accepts all strings of 0's and 1's that do not contain two or more successive 1's.
4 Problem: Design a d-FSA which accepts all strings of *0*'s and *1*'s in which the number of *1*'s is even.

5 EQUIVALENCES OF FSA's AND OTHER LANGUAGE SPECIFICATIONS TRANSFORMATION OF AN FSA INTO A RG (Regular Grammar)
5 For every FSA (either a d-FSA or an nd-FSA) there is an equivalent RG and vice-versa
5 Any FSA (Deterninistic or Non-Deterministic) to an RG
6 Any RG (Deterninistic or Non-Deterministic to an FSA
6 Definition of a Deterministic and non-deterministic Regular Grammar d-RG
7 Transforming a non-deterministic nd-FSA, nd-M **to an equivalent d-FSA,** d-M **.**
7 Examples nd-RG to nd- M to d-M Equivalent To nd-M
8 Determinism <---> Non-Determinism Examples

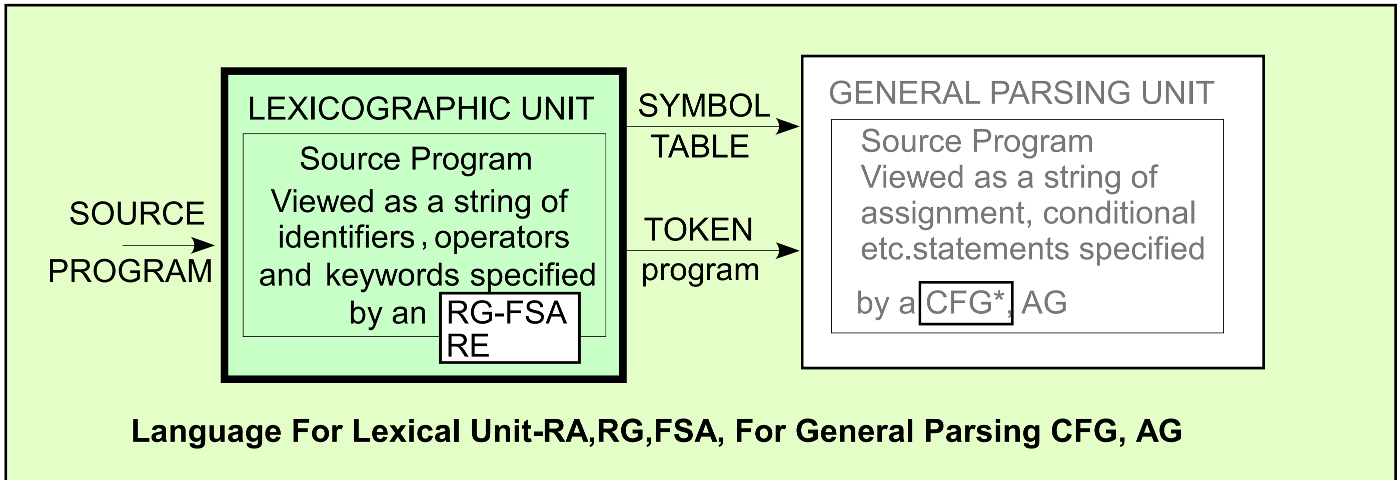
9 Regular Expressions
9 Examples Of Regular Expressions
10 Transforming an RG to an equivalent RE.
11 RG to RE Examples

12 Parsing With FSA-Implementation Realized With A Program

CONTENTS

© 2006 M C Paull

Parsing For The Lexicographic Unit



In Notes 2a I described CFGs and indicated that there are two classes of CFGs, usually used in a compiler implementation.

The more complex of these, used in the General Parsing section is a CFG that can be parsed in O(n) time. We showed examples of grammars in the classes LL(k), and LR(k) strings of whose language could be parsed in time O(n) and outline the algorithms for doing so and the machines which could generate the parse. (We did not give a set of restrictions on the CFG rules whose enforcement would guarantee the language to be of one of these types)

The second type of CFG is used in the initial, Lexicographic section, of the complier. For this a Regular Grammar is generally sufficient. In this case parsing is also O(n). The theory for such grammars is very rich.

Lexical Unit- Three ways of Defining Language RG(a) , FSA(b), RE (c)

(a) There are simple restrictions on a CFG grammar which will guarantee that it will be a **RG**:
As long as rules are restricted to two types namely:.
N ----> t, or **N ---> t** M where **t** is a single TS and **N** and **M** are single NTSs, perhaps the same, the resultant grammar is a RG

(b) Furthermore from a given **RG we can mechanically construct a simple machine, a Finite State Automata (FSA)** to parse any string in its language in time O(n). Typically such machines are represented in one of two equivalent wqys - either as a **state diagram** or a **state table**. They are either **deterministic** or **non-deterministic**. If non-deterministic it can be determine whether it is possible to construct a deterministic version and if so how to do that.

(c) As has been described there are shorthand representations aspects of CFGs. EBNF provides a number of these. For example **A ---> {0 | 1}** indicates that from NTS. A, any number n >= 0, of 0*s* or 1*s* interspersed in any way can be derived. One may ask-**for which CFGs, G, with starting symbol S, can the entire language derived from S be expressed as S = α,** where **α is a string of TSs and the operators {...} and |.** It turns out that for every RG, G the language it derives can be so described. Traditionally in expressing the language derived from an RG, the notation used is **Regular Expressions (REs) instead of EBNF** . In RE the operator (...)* is used instead {...}. The expression above for the language derived from A becomes **A ---> (0 | 1)*.** In addition to * for repetition, | is used as in EBNF for or and juxtaposition for concatenation.

Regular Expressions, RE, which is completely equivalent to RG*s* and FSA*s*. **Given an RG there is a mechanical transformation to an equivalent RE.** Some computer languages allow use of REs to describe classes of strings to be searched

© 2006 M C Paull

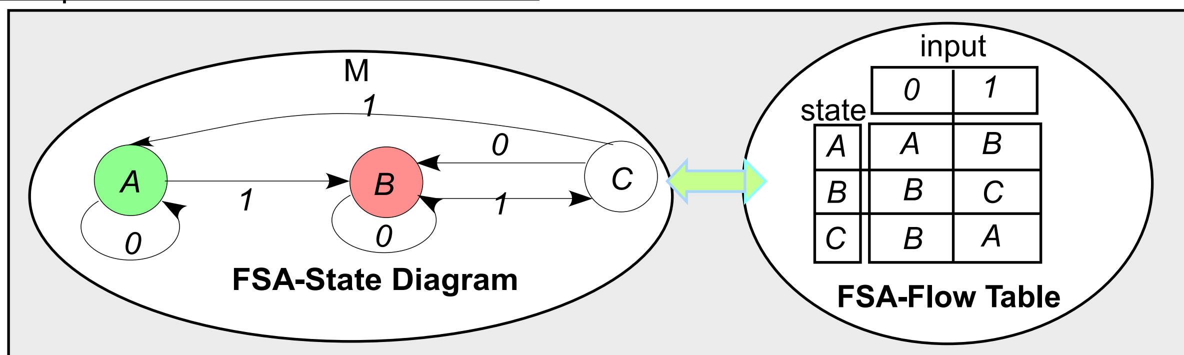
Finite State Automata (FSA)-Defimitios

A finite state automata (FSA) consists of
1. A finite set of states, S.
2. A finite set of inputs, I,(including the symbol representing a 0 length string ε)
3. A next state function δ: from a subset of S X I into S. (1 pair in S X I can map > 1 member of S)
4. One state in S, called the starting state.
5. One or more states in S designated final states.

For example consider the machine M:
1. finite set of states, S: { *A, B, C* }
2. finite set of inputs, I: { *0, 1* }
3. next state function δ: $\delta(A, 0) = A$ $\delta(B, 0) = B$ $\delta(C, 0) = B$
   $\delta(A, 1) = B$ $\delta(B, 1) = C$ $\delta(C, 1) = A$
4. *A* is the start state,
5. *B* is the final state (When the input stops only if FSA is in **Final State is input accepted**)

There are a number of ways to represent an FSA. One is by enumerating the δ function with arguments as above. Other ways are shown in the figure below: giving the δ function in tabular form, **FSA-Flow Table**, and in a graphically with a state diagram, **FSA-State Diagram**. Also representations are **Regular Grammars, RGs**, and **Regular Expression, REs**.

The FSA-State Diagram:
For each state there is a circle containing the identity of the state. and if there is a next state function $\delta(X, 0) = Y$ then there is a directed edge from state circle *X* to state circle *Y* labelled *0*. Notice that for each state there is a directed edge for each possible input. **ε**
When an edge for some input, say *i*, is not shown emerging from a state, say *X*, the implication is that input *i* cannot occur while in state *X*



The Language Defined By an FSA = The accepted Input Sequences
Let the input sequence to the FSA, (deterministic in this example) M, above be $i_1, i_2, \ldots, i_5 = 1\ 0\ 1\ 1\ 0$. The starting state is *A*, so with input $i_1 = 1$ the next state is $\delta(A, 1) = B$, then with input *0* the next state is $\delta(B, 0) = B$, and so on

. Thus M passes through the state sequence *ABBCAA*. At the end of sequence the last state is A which is not final so *1 0 1 1 0* is *not accepted* by M. On the other-hand *1 0 1 0* takes M through states *ABBCB*. B is the last state so *1 0 1 0* is *accepted* by M. Note that a state may be both a start and a final state and an input can take an FSA from a final state to another state, which itself may or may not be final. This will happen whenever M accepts any string as well as one of its sub-strings.
In general: A sequence of inputs ***J* is a string in the language of FSA M** iff that sequence takes M from its starting to its final state, and **the state sequence(s) t**hat in going from the start to final state constitute the **parse of *J* in M.**

Ambiguity
Every FSA will define a language, that is an acceptable set of strings, but it is not true that every FSA will produce an unambiguous parse. To see why this consider cases in which the same input **takes a state S to more than 1 state,** say the set of states $\{S_1 \ldots S_m\}$ we say $\delta(\{S\},i) = \{S_1 \ldots S_m\}$
In such a case

we can assume that after that input the FSA is in the entire set of states to which that input takes the FSA. If M can be in a set of states, $\{S_1....S_m\}$ we need a definition of the next set of states to which M will go for an input **i**. it is **union of the set of all states** $\boldsymbol{\delta(S_1, i)=U[\delta(S_2, i)U...U\delta(S_m, i)]}$. Now generally,every input sequence, **I**, is either unacceptable because one of the inputs in I is applied to a set of states none of which accept that it, or I takes M from its start to to a set of states none of which is a final state of M.

A state represents information about past inputs. There are only a finite number of states so we can only remember a limited amount about past inputs . A state, X, must represent. enough information about past inputs to know the state to which the FSA will be taken for each input received.

Designing FSAs- Examples

Note: If one can design an FSA from a description of a language, L, then one can easily obtain an RG describing L using a simple transformation: FSA to RG given later.

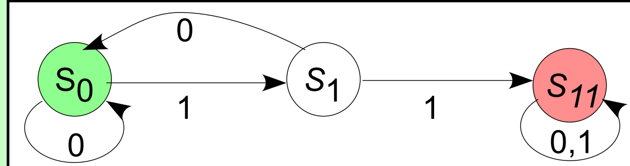
1) Problem: Design an d-FSA which accepts all strings of *0*'s and *1*'s that contain two or more successive 1's.

(a) Decide what states are needed and their interpretations

1. S_0 is the state when exactly zero succesive *1*'s have been received
2. S_1 is the state when exactly one succesive *1* has been received just before the current input and no occurence of two succesive ones have preceded it.
3. S_{11} is the state when exactly two succesive *1*'s has been recieved and zero or more additional inputs have occurs.

We give the next-state function for an d-FSA that performs in the required way, together with next state functions

$\delta(S_0, 0) = S_0$ $\delta(S_1, 0) = S_0$ $\delta(S_{11}, 0)=S_{11}$
$\delta(S_0, 1) = S_1$ $\delta(S_1, 1) =S_{11}$ $\delta(S_{11}, 1)= S_{11}$
S_0 is the start state, S_{11} is the final state



2)

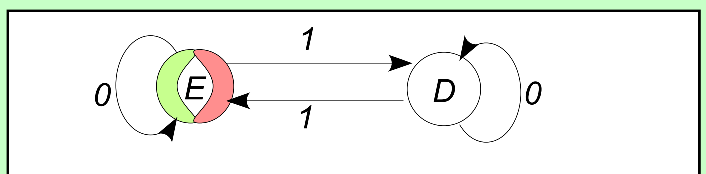
The states and the next-state function (δ) are the same as in the previous problem, but this machine is the inverse of M1, i.e., it will accept whenever the other doesn't. So we just have to label all states which were not final in M1, final in M2, and vice-versa. In M2 then S_0 and S_1 are final states and as in M1 S_0 is the start state. This is an example of an automata with more than one final state as wellas one with a state which is both starting and final and is neither in accepting some strings.

3) Problem: Design a d-FSA which accepts all strings of *0*'s and *1*'s in which the number of *1*'s is even.

The two states in this design are labelled *E* (even) and *O* (odd). *E* is the automata state when an even number of *1*'s has been received *O* is the state after an odd number of *1*'s. This immediately accounts for the following design., i.e., if there have been an odd number of *1*'s so the machine is in state *O*, anda *1* is received the machine goes to state *E*.

state \ input	*0*	*1*
E	*E*	*D*
D	*D*	*E*

E is a start as well as the only final state.



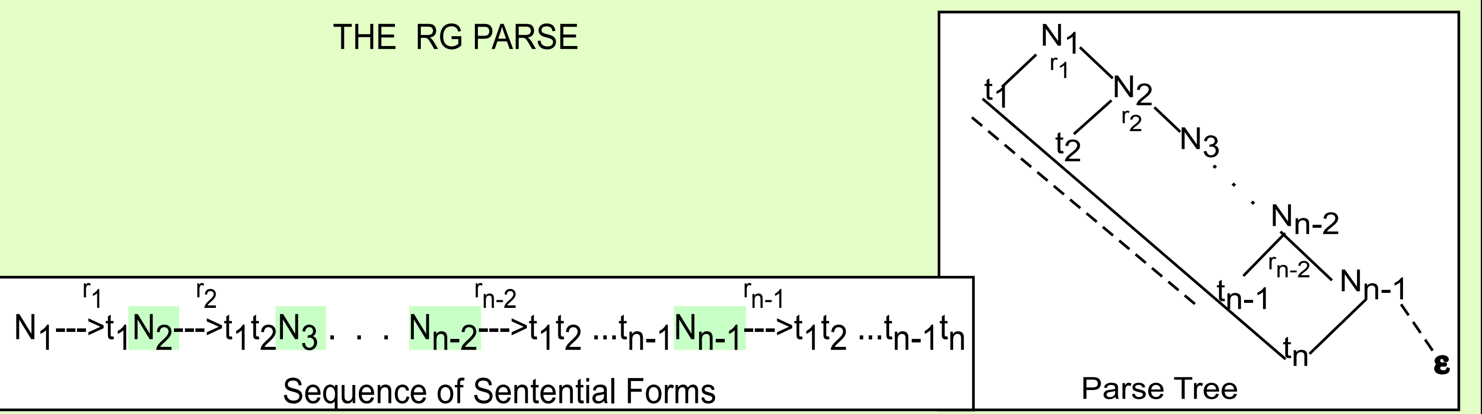
© 2006 M C Paull

1.1 EQUIVALENCES OF FSA's AND OTHER LANGUAGE SPECIFICATIONS
TRANSFORMATION OF AN FSA INTO A RG (Regular Grammar)

An FSA is equivalent to an RG if every strings accepted by the FSA is deriveable from the RG and vice-versa.

For every FSA (either a d-FSA or an nd-FSA) there is an equivalent RG and vice-versa

Before considering this relation between an RG and FSAs the effect of the use of ε in RGs needs clarification. A RG has been defined as Context Free Grammar in which the only two rules allowed are N--->t,and N--->tM where N and M are NTSs and t is a TS. Here we will include the use of the 0 length string, ε as a TS, This then allows us to include rules of the form N--->ε, and N--->εM (= N--->M) . These additions are a convenience but do not extend the languages that can be defined with a RG. To get incite into the effect of these rules consider the nature of a derivation from a RG



Notice that every sentential form except the last contains 1 NTS and that NTS is the last symbol in the sentential form. If, then there is a rule N---> **ε** that is applied to an NTS, the result will be a string consisting of all terminals, and the derivation will be complete, Thus any string of terminals, τ , preceding N produced by a derivation (any terminal string that is to be parsed whose derivation precedes N) will be in the language when ε is substituted for N. Since in a FSA there will generally be a state N corresponding to the NTS N, the rule N ---> ε will be represented in that FSA by making the state labelled N a final state.

Any FSA (Deterninistic or Non-Deterministic to an RG)

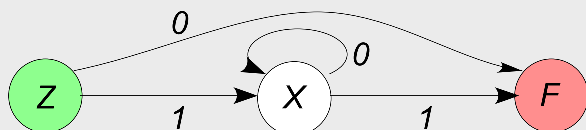
Given FSA M with *k* states. We design an RG, G, with *k* NTS's, one for each state in M. There is in G one TS for every input in M. The NTS N in this grammar is interpreted as the set of all strings that lead from N to any final state.

. 1 If *S* is the starting state of M then *S* is the starting symbol of G. If S is also the final state then S----> ε is a rule in G.

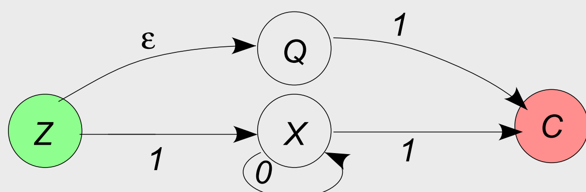
2. If δ(*X, i*) = *Y* is in M then *X* ----> *iY* (all the strings reaching the final state from *X* include those starting with *i* followed by all strings from Y to final) is a rule in G, (whether or not *Y* is a final or start state)

3. If δ(*X, i*) = *Y* is in M and *Y* is a final state then *X*----> *i* is a rule in G.

This transformation is good for any FSA, deterministic or not. The resultant RG will have different properties depending on whether the FSA is deterministic or not. If the FSA is a d-FSA the RG will be a deterministic RG (d-RG) as defined in describing the next transformation.]

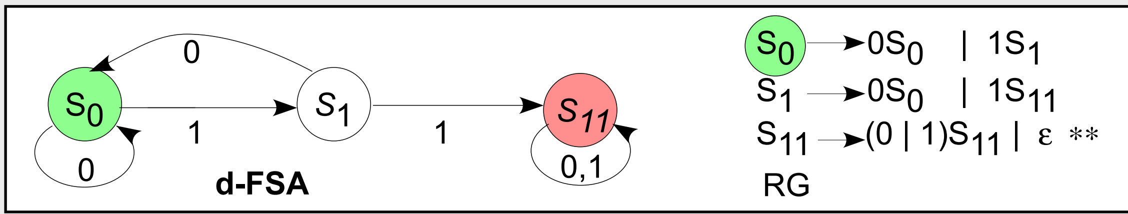


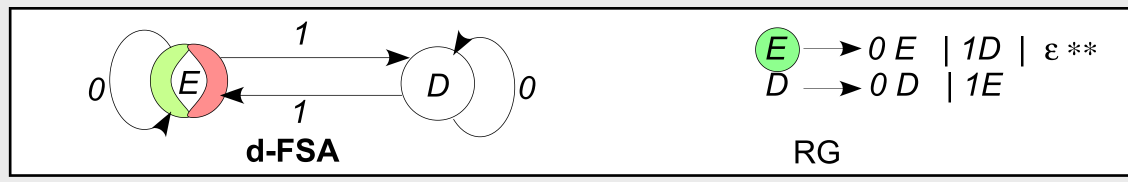
Z ---> *1 X | 0*
X ---> *0X | 1*



Z --->ε *Q | 1X*
X ---> *0X | 1C*
Q ---> *1C*
C ---> ε

© 2006 M C Paull





Any RG (Deterministic or Non-Deterministic) to an FSA

There is also a simple recipe for constructing an FSA for any given RG. That construction will often produce a non deterministic FSA (nd-FSA). Later we will give the procedure for generating a d-FSA equivalent to a given nd-FSA.

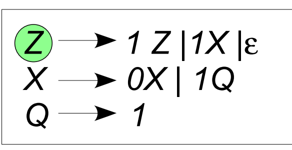
1 If *S* is the starting symbol of G then *S* is the starting state of M.

2 If *X* ---> ε is a rule in G then *X* is a final state of M

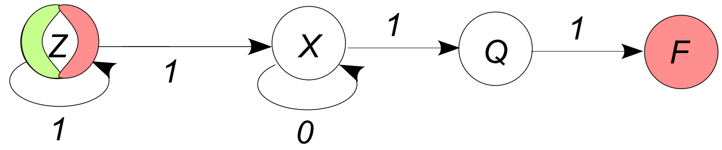
3 There is also a unique final state *F* with no outward edges in M

4 if *X---> iY in G then* δ*(X, i)* = *Y* is in M

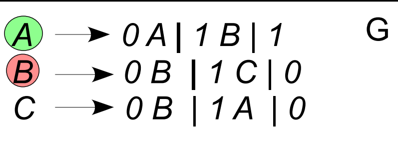
5. If *X---> i in G then* δ*(X, i)* = *F* is in M



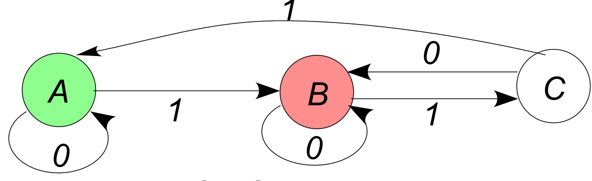
RG



FSA-State Diagram



RG



FSA-State Diagram

Deterministic, d-RG, and Non-deterministic Regular Grammar, nd-RG

In a deterministic FSA (a d-FSA), M, an input in any state takes M to one unique next state.

An RG (after removal of ε rules) is a d-RG provided: whenever wo right sides of rules with the same left size start with the same terminal symbolone is of length 1 and the other of l ength 2. That is in a d-RG we can have two rules *X---> iY* and X---> *i,* but not two rules *X---> iY* and *X---> iZ* with *Z != Y* .

If M is a non-deterministic FSA (nd-FSA), then when in, a state, it may go to several states simultaneously as the result of a single or an ε input. So *an input sequence, J,* takes M through one or more sequences of states to the final states. If one or more of these last states is a final state of M then M accepts *J*. Since a parse of consists of the *J* consists of the sequence of states it passes through in a accepting *J,* this can make the parse ambigouos. or even though non-ambiguous the correct selection of the multiple states arrived at after input I may depend on inputs following I.

** **Interpreting ε:** ε is ony necessary in an RG if the empty string is to be amongst the strings derived from the starting NTS-then if S is the starting NTS there needs to be a rule S ---> ε. ε may be used elsewhere as in the RGs above.The appearance of **ε** on the right a rule for any other NTS, N, ex. N ---> χ | ε, where χ is any string of NTS and TSs, and implies that N is terminal . The ε can be removed if all rules of the form A ---> tN are replaced by two rules A ---> tN, A ---> t.

Because of the ε input the FSA in figure below, is actually in a state combining Z and Q initially--it is thus non-deterministic. An equivalent d-FSA is given in (b). An algorithm fo getting an equivalent d-FSA for a given nd-FSA is given next.

Transforming a non-deterministic nd-FSA, nd-M **to an equivalent d-FSA,** d-M **.**

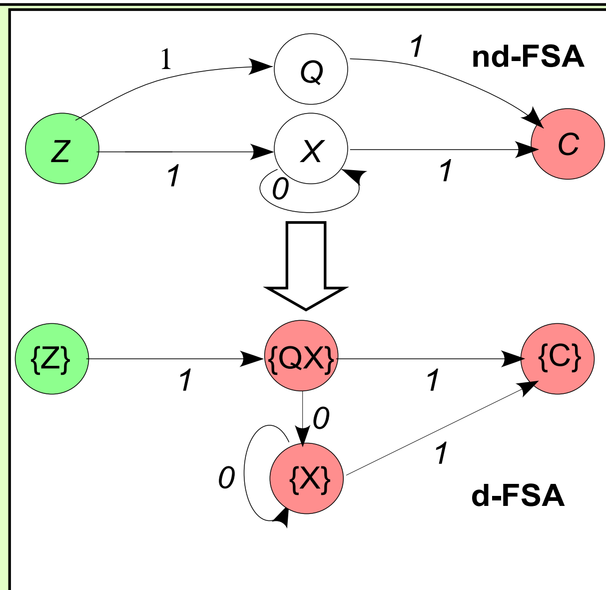
"Equivalent" means that nd-M and d-M accept exactly the same set of strings.

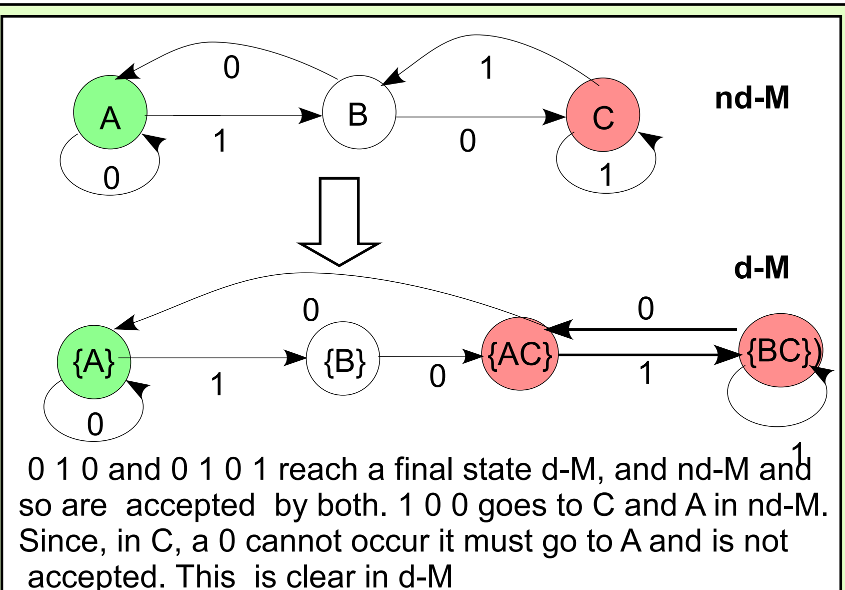
1 a In general each state in d-M is represented as a set of one or more states from nd-M.

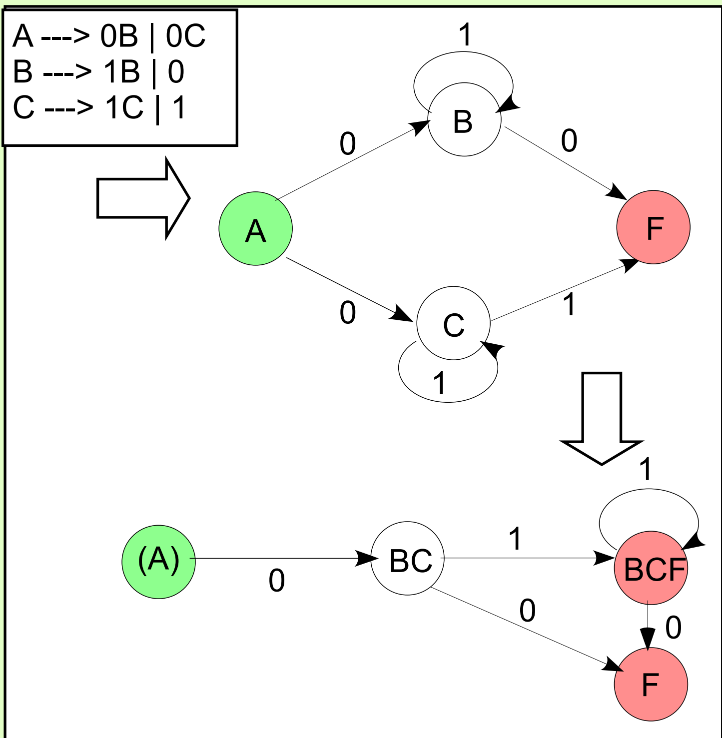
  b For each state *S* in nd-M there is a set of 1 state in state {*S*} in d-M.

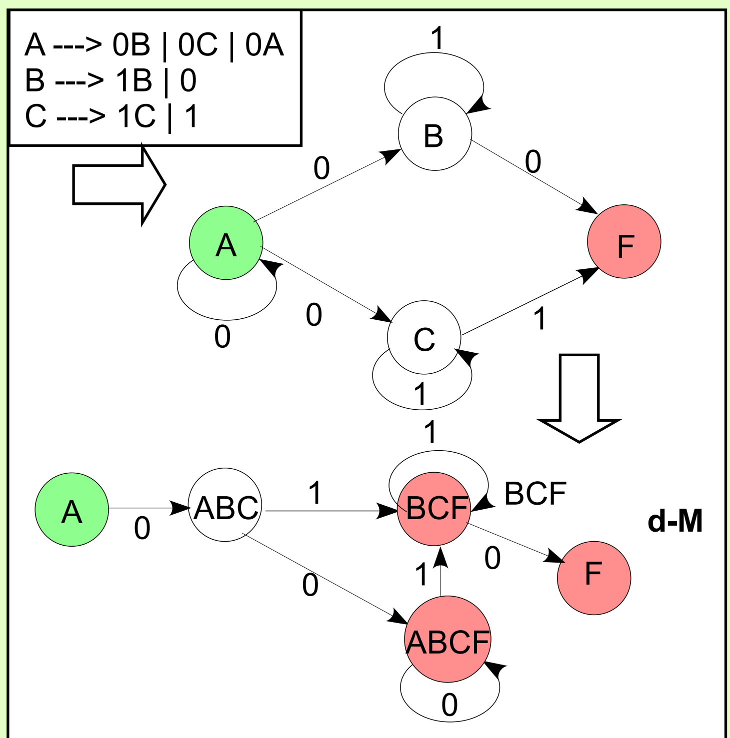
  c If $Z = \{S_1, S_2, ...S_n\}$ is a state in d_M then $\delta(Z, i) = Q$, where $X \varepsilon Q$ iff for a state $S_j \varepsilon Z$, $\delta(S_j, i) = X$. This process introduces new state(s) into *d-M.*

2 If any member of a state, $Z = (S_1, S_2, ...S_n\}$ in d-M, S_j , is final in nd-M then $\{S_1, S_2, ...S_n\}$ is final in d-M. If *X* is the starting state of nd-M then {X} is the starting state of the equivalent d-M. Any state not reachable from the starting state of d-M can be removed.

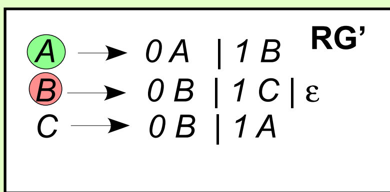








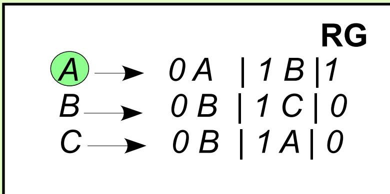
Examples nd-RG to nd- M to d-M Equivalent To nd-M



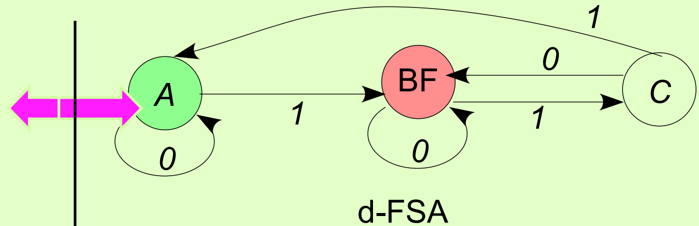
B ----> ε so B final state

Or the ε an be removed and an equivalent RG WITHOUT AN ε can be obtained:

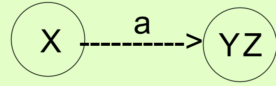
If B ---> ε, then by substitution
B ---> 0 B | 1 C | 0ε = 0
A ---> 0 A | 1B | 1ε = 0
C ---> 0 B | 1A | 0ε = 0



RG is equivalent to RG'

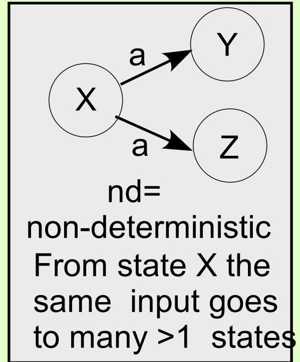


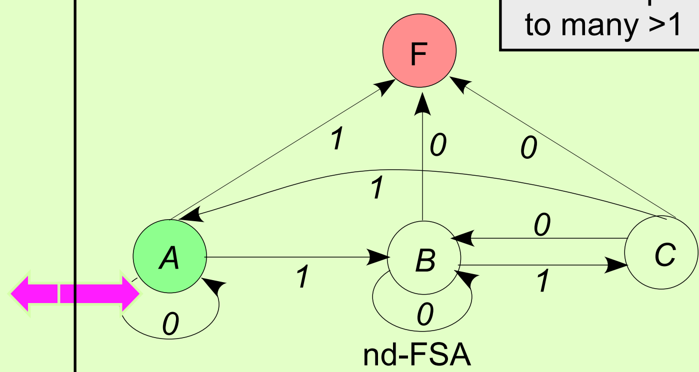
d-FSA



note if X or Y is final in nd-FSA then XY is final in d-FSA

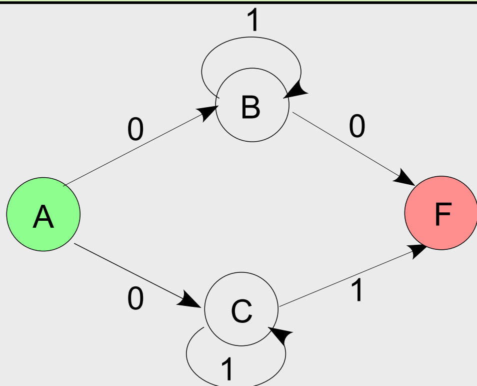
Two Different FSAs are Produced from The two equivalent RGs. These are then shown to be equivalent



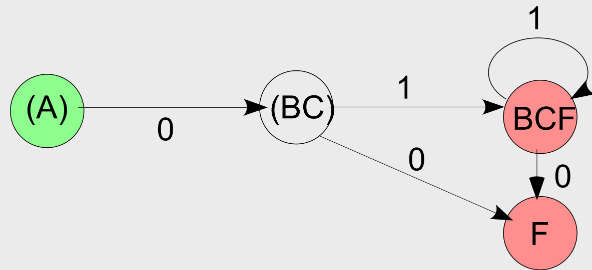


nd-FSA

EXAMPLE OF EQUIVALENT TRANSFORMATIONS



The parse for 0111...1 is ACCC...F
The parse for 0111...0 is ABBB... F



The parse for 0111...1 is A [BC [BCF] [BCF]... [BCF]
The parse for 01 is A [BCF]
The parse for 0111...0 is A [BC] [BCF] [BCF]...[F]
The parse for 00 is A [BC]..[F]

Both Accept the same strings-Define the same Language
But they have very different Parses for the same strings

Determinism <---> Non-Determinism Examples

© 2006 M C Paull

2 Regular Expressions

A regular expression (RE) is a string of symbols which represents(=>) a set of strings. The symbols used in writing an RE, together with the sets of strings they represent are given below.

What an RE represents (Informally)

1 **ε** represents { () }, a set whose only member is the 0 length string, ()

2. A finite set of symbols, called the alphabet S. Each of these represents a set containing a string whose only member is that symbol. 0 represents (0)
3. Three functions symbols
   (a) **|** : A two argument function representing Set Union.
   if R1=> {(0),(1)}, and R2 => {(a), (b)}, then R1 | R2 => {0, 1, a, b }

   (b) Juxtaposition: A binary function represents Set Cross Concatenation (sometimes symbolized with X). The result of cross concatenating of two sets of strings is a set containing every string that can be formed by concatenating a member of the .first with a member of the second.
   if R1=> {(0),(1)}, and R2 => {(a), (b)}, then R1 R2 => {(0 a), (0 b), (1 a), (1 b)}

   (c) _* : A unary function called "Kleene Closure". If RE, R represents set S then R* represents the set, S* which by definition contains ε together with every string that can be formed by concatenating copies of members of S in any order. Any number of copies of each member may be used in a concatenation.

An RE is not any arbitrary string of the above symbols. It must be constructed in a particular way. Exactly which expressions involving these symbols are RE's is defined recursively below, with their interpretation as sets of strings. If γ is an RE then L(**γ**) is the set of strings it describes.

What an RE looks Like and the Language it Represents (Formally)

1. **ε** is an RE. L(**ε**) = *{<>}*
2. Each distinct alphabetic symbol **x** in S is an RE. L(**x**) = *{(***x***) }*
3. If $\mathbf{R_1}$, $\mathbf{R_2}$, and $\mathbf{R_3}$ are each regular expression then so is each of the expressions below.

   Let L(**R**) be the set of strings represented by the regular expression **R**:

   a. RE =$\mathbf{(R_1)}$ then L($\mathbf{(R_1)}$) = L($\mathbf{R_1}$). This definition allows one to use parentheses to separate parts of an RE. It does not define a new set. To minimize parenthesese, the rule that * has precedence over "juxtaposition", which has precedence over **|** is adopted.

   b. RE = $\mathbf{R_1|R_2}$ then L($\mathbf{R_1|R_2}$) = L($\mathbf{R_1}$) U L($\mathbf{R_2}$)

   c. RE = $\mathbf{(R_1)(R_2)}$ then L($\mathbf{(R_1)\ (R_2)}$) = L($\mathbf{R_1}$)| X |L($\mathbf{R_2}$)

   d. RE = $\mathbf{(R_1)^*}$ then L($\mathbf{(R_1)^*}$) = L($\mathbf{(R_1)}$)*

3. No string which is not constructable as describe above is an RE.

EXAMPLES:

The following RE's assume the alphabet, S, is {0, 1}. Under each RE the set it defines is given.

ε,	**0,**	**1**	**1 | 0** *Union*	**(1 | 0) 1** *Cross Concatenation*
{ () },	*{ (0) },*	*{ (1) }*	*{ (1), (0) }*	*{ (1), (0) } X { (1) } = { (1 1), (0 1) }*

(1|0)(0|1)(0)	**(010|1|00)(1|0)**
{ (1 0 0),(1 1 0),(0 0 0),(0 1 0) }	*{ (0 1 0 1), (0 1 0 0), (1 1), (1 0), (0 0 1),(0 0 0) }*

(0)*=0*	**(0|1)***
{ ( ),(0), (0 0),(000), . . .}	*{ (),(0 >,(1),(01),(10 >, . . .}*
[ε and all strings of 0's]	[ε and all binary numbers of lengths 1, 2,..]

Examples Of Regular Expressions

© 2006 M C Paull

Transforming an RG to an equivalent RE.

In representing an RG we use EBNF notation So for each NTS in the language there is one equation of the form

$A \dashrightarrow \phi A \mid \eta$ this "equation" represents the set of **all** rules with A on the left. This works becauseusing EBNF notation,

$\phi = (\phi_1 \mid \phi_2 \mid \ldots \mid \phi_p)$ and $\eta = (\eta_1 \mid \eta_2 \mid \ldots \mid \eta_m)$ ***that is***

$A \dashrightarrow (\phi_1 \mid \phi_2 \mid \ldots \mid \phi_p)\ A \mid (\eta_1 \mid \eta_2 \mid \ldots \mid \eta_m)$ *where* ϕ_j *and* η_k *are strings of TSs and NTSs*

So an RG can be interpreted as a set of equations, one for each NTS in the RG . The solution of the set for the value of the starting NTS relies on two operations:

(1) **Substitution**

If the grammar contains rule $A \dashrightarrow \gamma$, and rule $X \dashrightarrow \alpha A \mid \beta$ we can replace the A in the latter with $X \dashrightarrow \alpha\gamma \mid \beta$.

(2) **Recursion Removal**

Remove any reference to the NTS on left of ----> from its right using the * operation
If the grammar contains

$A \dashrightarrow \phi A \mid \eta$ *(expanded form* $A \dashrightarrow (\phi_1 \mid \phi_2 \mid \ldots \mid \phi_p)\ A \mid (\eta_1 \mid \eta_2 \mid \ldots \mid \eta_m))$

T*hat rule* can be replaced with

$A \dashrightarrow \phi^*(\eta)$ *(expanded form* $A \dashrightarrow (\phi_1 \mid \phi_2 \mid \ldots \mid \phi_p)^* (\eta_1 \mid \eta_2 \mid \ldots \mid\mid \eta_m))$

Because A *includes* η, A *includes* $\phi A \mid \eta$, So A includes $\phi^2 A \mid \eta$ **,** *etc. In genral then . . .* A *includes* $\phi^m A \mid \eta$, *for all* $m>=0$, *or substituting* η *for* A $\phi^m\eta$, *for all* $m>=0$, *or compactly* $A = \phi^*\eta$

Solution of a set of production rules (RG) for representing the RE derived from the starting symbol .

Using the two operations, the solution of the set of equation can now be described. Given a number of equations of the form described above

$A_1 \dashrightarrow \phi_1 A_1 \mid \eta_1$

$A_2 \dashrightarrow \phi_2 A_2 \mid \eta_2$

.

$A_j \dashrightarrow \phi_j A_j \mid \eta_j$ $\quad A \dashrightarrow (\phi_{j1} \mid \phi_{j2} \mid \ldots \mid \phi_{jp})\ A \mid (\eta_{j1} \mid \eta_{j2} \mid \ldots \mid \eta_m)$

where A_m *is the start symbol*

$A_m \dashrightarrow \phi_m A_m \mid \eta_m$

Remove recursion from $A_1 \dashrightarrow \phi_1 A_1 \mid \eta_1$ *replace it with* $A_1 \dashrightarrow \phi_1^*(\eta_1)$ clearly A_1 *does not appear in* $\phi_1^*(\eta_1)$ so we now can remove A_1 from the right side of all equations by substituting $\phi_1^*(\eta_1)$ for A_1 *in all right sides in which it appears. Now* The same process is repeated fo*r* $A_j \dashrightarrow \phi_j(1_j\text{-}1)\ A_j(1\text{-}j_1) \mid \eta_j(1\text{-}j\text{-}1)$ *where* $Z_j(1_j\text{-}1)$ *represents* Z_j *after substitution for* $A_1, A_2, \ldots, A_j$ and by substitution in all other equations for A_k with $k > j$ (but not for $k < j$ unless we need to have regular expressions for all NTSs and not just for the start NTS). When this Process is completed with substitution for A_{m-1} *for j= m-1 the result s a right side of the equation for* A_m *has no NTSs*

E ---> 0 *E* | 1*D* | ε
D ---> 0 *D* | 1*E*

D ---> 0* 1*E* — Recursiion Removal

E ---> (0 | 10*1)*E* | ε — Substitution

E ---> (0 | 10*1)* **RE** — Recursiion Removal

any string of 0s and 1s with an even number of 1s

Z ---> 1 *X* | *Q*
X ---> 0*X* | 1
Q ---> 1

Z ---> 1 *X* | 1 — Substitution

X ---> 0* 1 — Recursiion Removal

Z ---> 1 0*1 | 1 **RE** — Substitution

1 or any strings of 0s and 1s which is 1 followed by any number of 0s followed by a 1

S_0 ---> $0S_0$ | $1S_1$
S_1 ---> $0S_0$ | $1S_{11}$
S_{11} ---> (0 | 1) S_{11} | ε

S_0 ---> (0 | 10) S_0 | $11S_{11}$ — Having substituted for S_1

S_{11} ---> (0 | 1)*ε = (0 | 1)* — Having recursed on S_{11}

S_0 ---> (0 | 10) S_0 | 11 (0 | 1)* — Having substituted for S_{11}

S_0 - --> (0 | 10)* (11 (0 | 1)*) **RE** — Having recursed on S0

*any string with **no two** adjacent 1s*
followed by two successive 1s
followed by any string of 0s and 1s

RG to RE Examples

TRANSFORMATION OF FSA INTO A PROGRAM THAT IMPLEMENTS THE FSA.

For each state of M there is a location in the corresponding program P labelled with that state. There is a variable INPUT into which succesive inputs are read. Each input string is terminated with a special symbol “!”.

1. Let location s correspond to the start state M. Then program P starts at location s. P also has a location INPUT into which succesive inputs are read. Assume there are k inputs:
2. If X is not a final state and $\delta(X, i_1) = Y_1, \ldots, \delta(X, i_r) = Y_r$, then the following code is added at location X.

X: read(INPUT);
    if INPUT = ! then (write(“unaccepted”);go to S;);
    if INPUT = i_1 then go to Y_1;

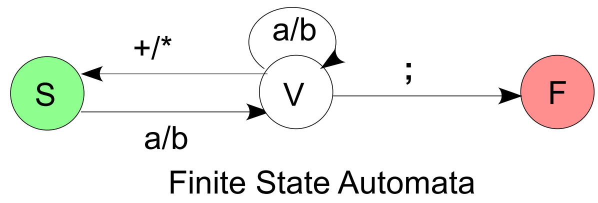
    if INPUT = i_r then go to Y_f

3. If X is a final state and $\delta(X, i_1) = Y_1, \ldots, \delta(X, i_r) = Y_r$, then the code added at location X is the same as in 2 except for line 2 which becomes
    if INPUT = ! then (write(“accepted”);go to S;).

© 2006 M C Paull

S ---> a V | b V
V ---> a V | b V | + S | * S | ;

Regular Grammar



Finite State Automata

((a |b) (a | b)+ (+|*))*

Regular Expression

For Greatly Simplified LEX Unit

```
S: getchar(x)
   if (x==a || x==b) psh(x, Vstack)  ; goto V;
   else goto ERR;
 V: getcharx)
    if ((x==a || x==b)  psh(x, Vstack); goto V;
     if ((x==+ || x==*) popall(Vstack, var);
                        ST = var;  /:ST is Symbol Table:/
                        psh(Tokenize(var), L);
                        psh(x, L);  goto S;
     if ((x== ;) (  popall(Vstack, var);
                   ST = var;
                    psh(Tokenize(var), L);
                   psh(x, L); OK Done
     else goto ERR;
```

FSA PROGRAM WITH **gotos**

```
while(state == S || state == V)
 {  getchar(x)
    case state of
      S:  { if( x== "a" || x= "b"){ psh(x, Vstack); state=V;}}

      V:  { if( x== "a" || x= "b") {psh(x, Vstack); state=V;}
            else
               if( x== "+" || x= "*") {popall(Vstack, var);
                                        ST = var;
                                       psh(Tokenize(var), L);
                                       psh(x, L); state=S;
      ELSE:  if( x== ";")   state=F;
  }
  if (state != F) ERR(state)
```

FSA PROGRAM WIT**H case statement, while, and if**

```
S: C=1
while(C=1)
 {  getchar(x)
    if (x==a || x==b)
     {  psh(x, Vstack) ;
     getchar(x);
     while( (x==a || x==b )
        { psh(x, Vstack) ;
        getchar(x);
        }
      if i((x==+ || x==*)
        {  popall(Vstack, var);
           ST = var;
           psh(Tokenize(var), L);
           psh(x, L);
           C=1;
         }
      else
         {   if i((x==;)
              {  C=0;
                 popall(Vstack, var);
                 ST = var;
                 psh(Tokenize(var), L);
                 psh(x, L);
                 tag=0
               }
              else tag=1;
         }
     }
    else  tag = 2;
 }
if(tag > 0)ERR(tag);
```

FSA PROGRAM With **whiles and if**

The FSAs are Implemented As DescribedOn the Previous pagein black. The blue gives a set of actions that realize the Symbol Table entries and the tokenizing of the inputs.

Parsing With FSA-Implementation Realized With A Program

© 2006 M C Paull