Compiling: Examples and Sample Problems

ICS312
Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

REs for Keywords

- It is easy to define a RE that describes all keywords

  Key = 'if' | 'else' | 'for' | 'while' | 'int' | ..

- These can be split in groups if needed

  Keyword = 'if' | 'else' | 'for' | …
  Type = 'int' | 'double' | 'long' | …

- The choice depends on what the next component (i.e., the parser) would like to see

RE for Numbers

- Straightforward representation for integers
  - digits = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
  - integer = digits$^+$
- RE systems allow the use of '-' for ranges, sometimes with '[' and ']'
  - digits = [0-9]+
- Floating point numbers are much more complicated
  - 2.00, .12e-12, 312.00001E+12, 4, 3.141e-12
- Here is one attempt
  - ('+'|'-'|ε)(digit$^+$ '.'? | digits* ('.' digit$^+$)) (('E'|'e')('+'|'-'|ε) digit+)))?
- Note the difference between meta-character and language-characters
  - '+' versus +, '-' versus -, '(' versus (, etc.
- Often books/documentations use different fonts for each level of language

RE for Identifiers

- Here is a typical description
  - letter = a-z | A-Z
  - ident = letter (letter | digit | '_')*
    - Starts with a letter
    - Has any number of letter or digit or '_' afterwards
- In C: ident = (letter | '_') (letter | digit | '_')*

RE for Phone Numbers

- Simple RE
  - digit = 0-9
  - area = digit digit digit
  - exchange = digit digit digit
  - local = digit digit digit digit
  - phonenumber = ‘(’ area ‘)’ ‘ ‘? exchange (’-’|’ ‘) local
- The above describes the 10^{3+3+4} strings of the L(phonenumber) language

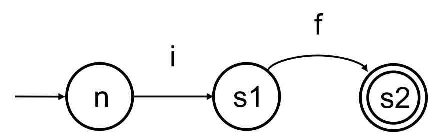
Regular Expression Practice

- Write regular expressions for
  - All strings over alphabet {a,b,c}
  - All strings over alphabet {a,b,c} that contain substring ‘abc’
  - All strings over alphabet {a,b,c} that consist of one or more a’s, followed by two b’s, followed by whatever sequence of a’s and c’s
  - All strings over alphabet {a,b,c} such that they contain at least one of substrings ‘abc’ or ‘cba’

Regular Expression Practice

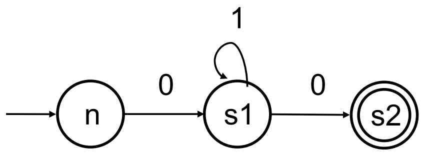
- Write regular expressions for
  - All strings over alphabet {a,b,c}
    - (a|b|c)*
  - All strings over alphabet {a,b,c} that contain substring ‘abc’
    - (a|b|c)*abc(a|b|c)*
  - All strings over alphabet {a,b,c} that consist of one or more a’s, followed by two b’s, followed by whatever sequence of a’s and c’s
    - $a^+bb(a|c)^*$
  - All strings over alphabet {a,b,c} such that they contain at least one of substrings ‘abc’ or ‘cba’
    - ((a|b|c)*abc(a|b|c)* | (a|b|c)*cba(a|b|c)*)

Automaton Examples



- This automaton accepts input ‘if’

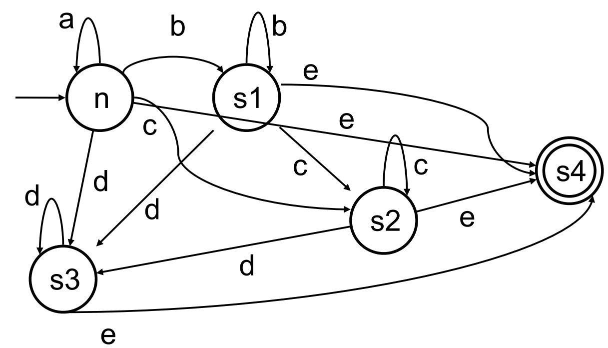
Automaton Examples



- This automaton accepts strings that start with a 0, then have any number of 1's, and end with a 0
- Note the natural correspondence between automata and REs: 01^*0
- Question: can we represent all REs with simple automata?
- Answer: yes
- Therefore, if we write a piece of code that implements arbitrary automata, we have a piece of code that implements arbitrary REs, and we have a lexer!
  - Not _this_ simple, but close

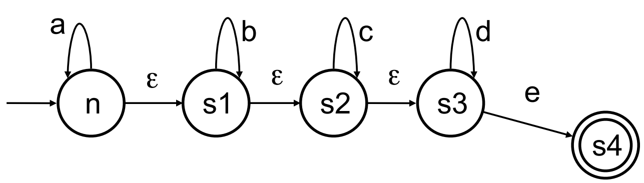
Example REs and DFA

- Say we want to represent RE '$a^*b^*c^*d^*e$' with aDFA



Example REs and NFA

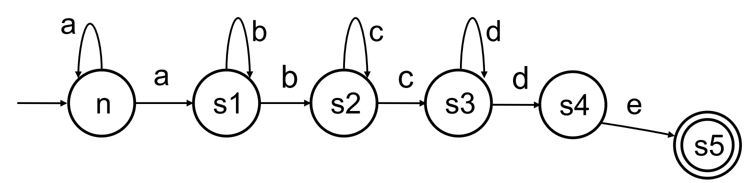
- '$a^*b^*c^*d^*e$': much simpler with a NFA



- With ε-transitions, the automaton can 'choose' to skip ahead, non-deterministically

Example REs and NFA

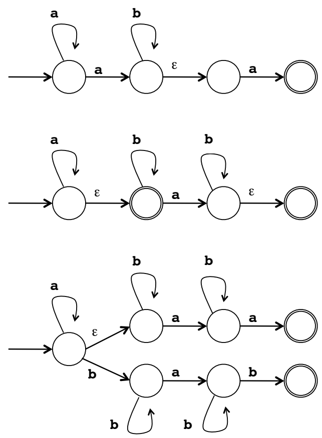
- '$a^+b^+c^+d^+e$': easy modification



- But now we have multiple choices for a given character at each state!
  - e.g., two 'a' arrows leaving n

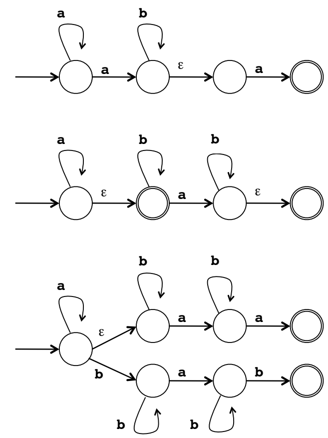
Automaton vs. RE Practice

- Write REs for the following NFAs



Automaton vs. RE Practice

- Write REs for the following NFAs



a^+b^*a

$a^*b^*(\varepsilon|ab^*)$

$a^*b^*(ab^*a \mid bab^+)$

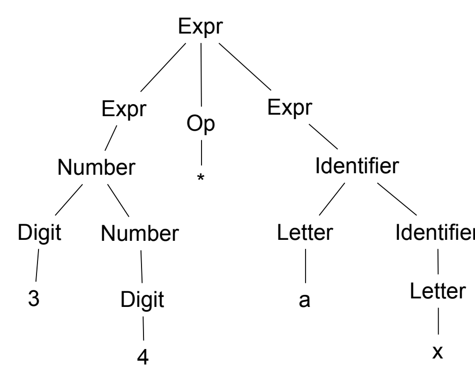
A Grammar for Expressions

Expr	➔ Expr Op Expr
Expr	➔ Number | Identifier
Identifier	➔ Letter | Letter Identifier
Letter	➔ a-z
Op	➔ "+" | "-" | "*" | "/"
Number	➔ Digit Number | Digit
Digit	➔ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Expr ➔ Expr Op Expr ➔ Number Op Expr ➔
Digit Number Op Expr ➔ 3 Number Op Expr ➔ 34 Op Expr ➔
34 * Expr ➔ 34 * Identifier ➔ 34 * Letter Identifier ➔
34 * a Identifier ➔ 34 * a Letter ➔ 34 * ax

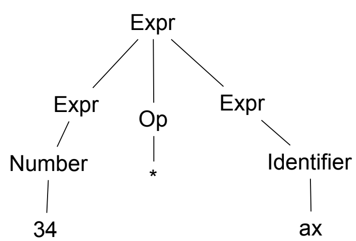
Derivations as Trees

- A convenient and natural way to represent a sequence of derivations is a syntactic tree or parse tree
- Example: Expr ➔ Expr Op Expr ➔ Number Op Expr ➔ Digit Number Op Expr ➔ 3 Number Op Expr ➔ 34 Op Expr ➔ 34 * Expr ➔ 34 * Identifier ➔ 34 * Letter Identifier ➔ 34 * a Identifier ➔ 34 * a Letter ➔ 34 * ax



Derivations as Trees

- In the parser, derivations are implemented as trees
- Often, we draw trees without the full derivations
- Example:



Grammar Practice

- Consider the CFG:

S	→	(L) | a
L	→	L , S | S

Draw parse trees for:

(a, a)

(a, ((a, a), (a, a)))

Grammar Practice

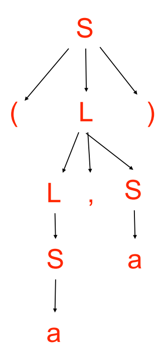
- Consider the CFG:

S	→	(L) | a
L	→	L , S | S

Draw parse trees for:

(a, a)

(a, ((a, a), (a, a)))



Grammar Practice

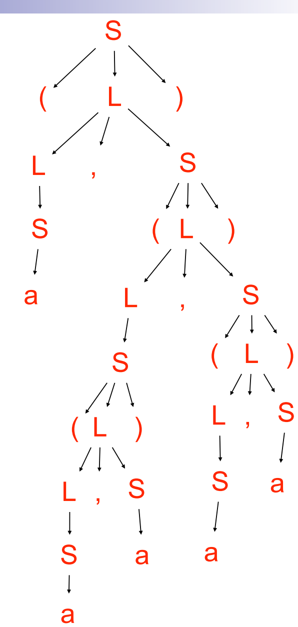
- Consider the CFG:

S	→	(L) | a
L	→	L , S | S

Draw parse trees for:

(a, a)

(a, ((a, a), (a, a)))



Grammar Practice

- Write a CFG for the language of well-formed parenthesized expressions
  - (), (()), ()(), (()()), etc.: OK
  - ()),)(, ((()), (((, etc.: not OK

Grammar Practice

- Write a CFG for the language of well-formed parenthesized expressions
  - (), (()), ()(), (()()), etc.: OK
  - ()),)(, ((()), (((, etc.: not OK

P → () | PP | (P)

Grammar Practice

- Is the following grammar ambiguous?

A → A "and" A | "not" A | "0" | "1"

Grammar Practice

- Is the following grammar ambiguous?
  A → A "and" A | not A | 0 | 1