RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

CS415 Compilers

Lexical Analysis
Part 3

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University

Announcements

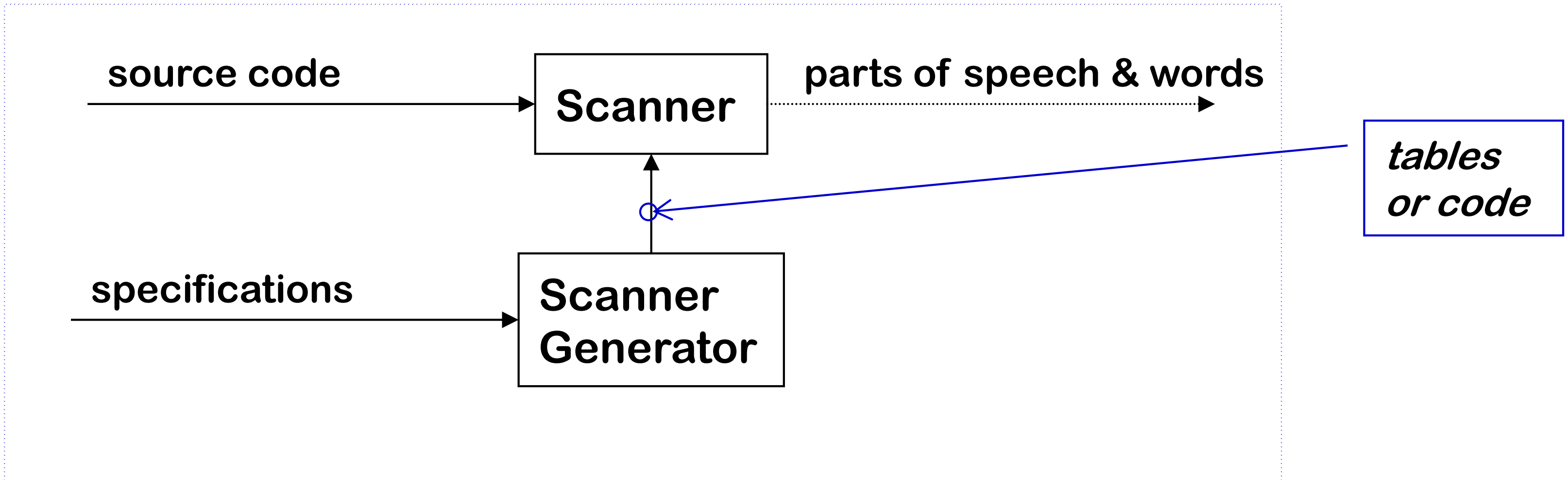
- Homework solutions for homeworks 1 and 2 have been posted on canvas under "Files" tab

- Third homework will be posted after exam.

- First project (local instruction scheduler) has been posted
  Deadline for code: March 2
  Deadline for report: March 4

- First midterm: This Wednesday, February 23
  In class exam, 60 minutes,
  Topics: ILOC, instruction scheduling, register allocation

Constructing a Scanner - Quick Review



- The scanner is the first stage in the front end
- Specifications can be expressed using regular expressions
- Build tables and code from a DFA

RUTGERS

Goal

- We will show how to construct a finite state automaton to recognize any RE
- Overview:
  - → Direct construction of a nondeterministic finite automaton (NFA) to recognize a given RE
    - Requires ε-transitions to combine regular subexpressions
  - → Construct a deterministic finite automaton (DFA) to simulate the NFA
    - Use a set-of-states construction
  - → Minimize the number of states
    - Hopcroft state minimization algorithm
  - → Generate the scanner code
    - Additional specifications needed for details

More Regular Expressions

- All strings of 1s and 0s ending in a 1

  $(\underline{0} \mid \underline{1})^* \underline{1}$

- All strings over lowercase letters where the vowels (a,e,i,o, & u) occur exactly once, in ascending order

  $\mathit{Cons} \rightarrow (\underline{b} \mid \underline{c} \mid \underline{d} \mid \underline{f} \mid \underline{g} \mid \underline{h} \mid \underline{j} \mid \underline{k} \mid \underline{l} \mid \underline{m} \mid \underline{n} \mid \underline{p} \mid \underline{q} \mid \underline{r} \mid \underline{s} \mid \underline{t} \mid \underline{v} \mid \underline{w} \mid \underline{x} \mid \underline{y} \mid \underline{z})$

- All strings of 1s and 0s that do not contain three 0s in a row:

More Regular Expressions

- All strings of 1s and 0s ending in a 1

  $(\underline{0} \mid \underline{1})^* \, \underline{1}$

- All strings over lowercase letters where the vowels (a,e,i,o, & u) occur exactly once, in ascending order

  $Cons \rightarrow (\underline{b}\mid\underline{c}\mid\underline{d}\mid\underline{f}\mid\underline{g}\mid\underline{h}\mid\underline{j}\mid\underline{k}\mid\underline{l}\mid\underline{m}\mid\underline{n}\mid\underline{p}\mid\underline{q}\mid\underline{r}\mid\underline{s}\mid\underline{t}\mid\underline{v}\mid\underline{w}\mid\underline{x}\mid\underline{y}\mid\underline{z})$

  $Cons^* \; \underline{a} \; Cons^* \; \underline{e} \; Cons^* \; \underline{i} \; Cons^* \; \underline{o} \; Cons^* \; \underline{u} \; Cons^*$

- All strings of $\underline{1}$s and $\underline{0}$s that do not contain three $\underline{0}$s in a row:

More Regular Expressions

- All strings of 1s and 0s ending in a 1

  $(\underline{0} \mid \underline{1})^* \underline{1}$

- All strings over lowercase letters where the vowels (a,e,i,o, & u) occur exactly once, in ascending order

  $Cons \rightarrow (\underline{b}|\underline{c}|\underline{d}|\underline{f}|\underline{g}|\underline{h}|\underline{j}|\underline{k}|\underline{l}|\underline{m}|\underline{n}|\underline{p}|\underline{q}|\underline{r}|\underline{s}|\underline{t}|\underline{v}|\underline{w}|\underline{x}|\underline{y}|\underline{z})$
  $Cons^* \ \underline{a} \ Cons^* \ \underline{e} \ Cons^* \ \underline{i} \ Cons^* \ \underline{o} \ Cons^* \ \underline{u} \ Cons^*$

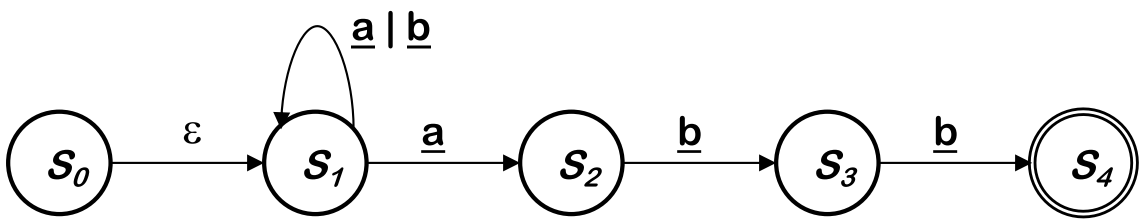
- All strings of 1s and 0s that do not contain three 0s in a row:
  $(\underline{1}^* (\varepsilon \mid \underline{01} \mid \underline{001}) \underline{1}^*)^* (\varepsilon \mid \underline{0} \mid \underline{00})$

Non-deterministic Finite Automata

Each RE corresponds to a *deterministic finite automaton* (DFA)

- May be hard to directly construct the right DFA

What about an RE such as (a | b)* abb ?



This is a little different

- S_0 has a transition on ε
- S_1 has two transitions on a

This is a *non-deterministic finite automaton* (NFA)

Non-deterministic Finite Automata

- An NFA accepts a string *x* iff ∃ a path though the transition graph from s_0 to a final state such that the edge labels spell *x*
- Transitions on ε consume no input
- To "run" the NFA, start in s_0 and *guess* the right transition at each step
  - → Always guess correctly
  - → If some sequence of correct guesses accepts x then accept

Why study NFAs?

- They are the key to automating the RE→DFA construction
- We can paste together NFAs with ε-transitions

Relationship between NFAs and DFAs

DFA is a special case of an NFA

- DFA has no ε transitions
- DFA's transition function is single-valued
- Same rules will work

DFA can be simulated with an NFA

→ *Obviously*

NFA can be simulated with a DFA *(less obvious)*

- Simulate sets of possible states
- Possible exponential blowup in the state space
- Still, one state transition per character in the input stream

Automating Scanner Construction

To convert a specification into code:

1. Write down the RE for the input language
2. Build a big NFA
3. Build the DFA that simulates the NFA
4. Systematically shrink the DFA
5. Turn it into code

Scanner generators

- Lex and Flex work along these lines
- Algorithms are well-known and well-understood
- Key issue is interface to parser
- You could build one in a weekend!

Automating Scanner Construction

RE → NFA *(Thompson's construction)*

- Build an NFA for each term
- Combine them with ε-moves

NFA → DFA *(subset construction)*

- Build the simulation

DFA → Minimal DFA

- Hopcroft's algorithm

DFA →RE *(Not part of the scanner construction)*

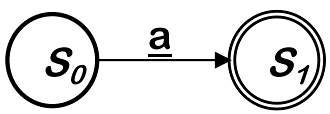
- All pairs, all paths problem
- Take the union of all paths from s_0 to an accepting state

The Cycle of Constructions

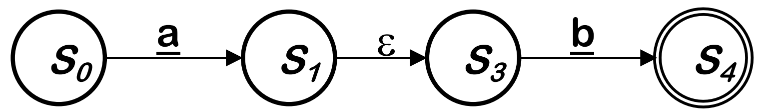
RE →NFA using Thompson's Construction

Key idea

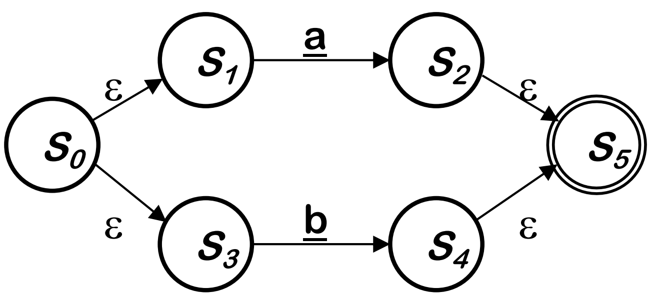
- NFA pattern for each symbol and each operator
- Each NFA has a single start and accept state
- Join them with ε moves in precedence order



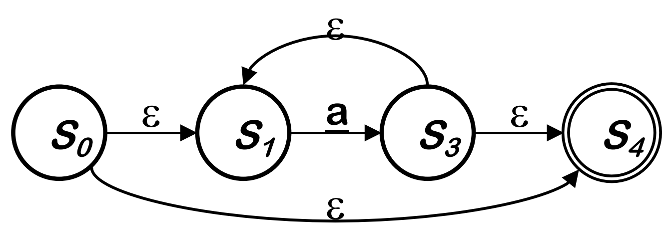
NFA for a



NFA for ab



NFA for a | b



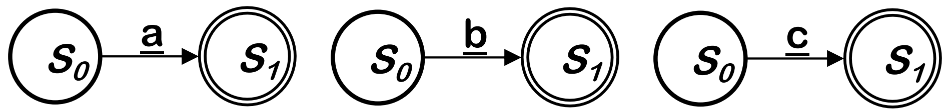
NFA for a^*

Ken Thompson, CACM, 1968

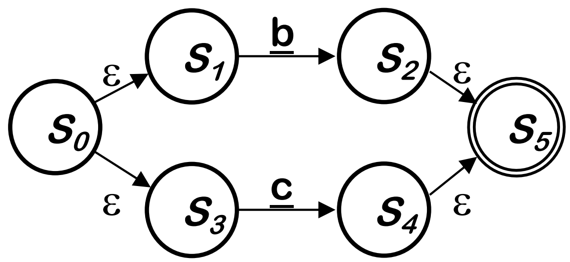
Example of Thompson's Construction

Let's try a (b | c)*

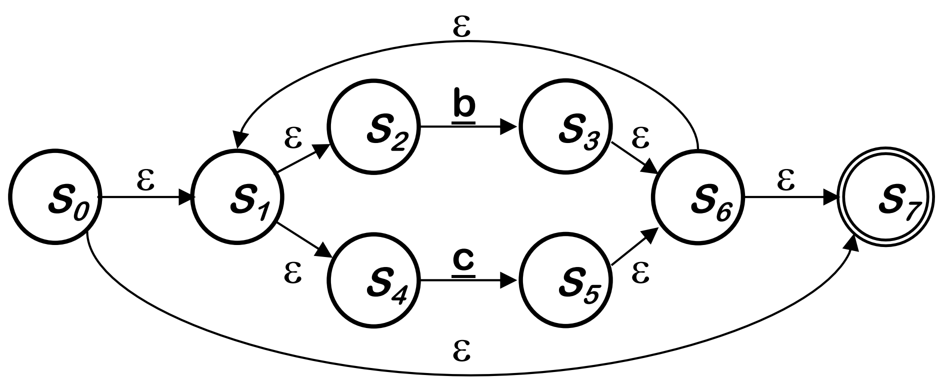
1. a, b, & c



2. b | c

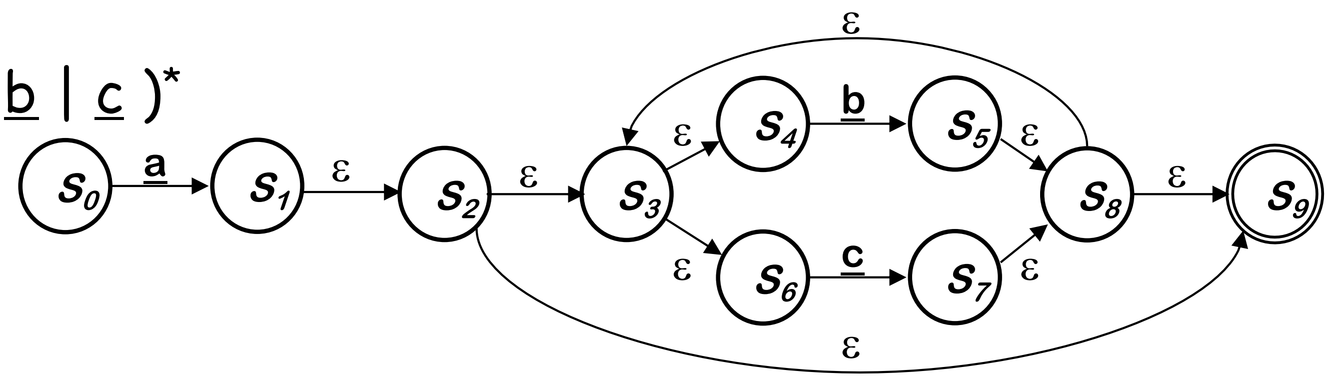


3. (b | c)*

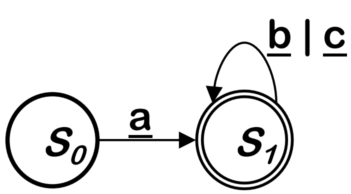


Example of Thompson's Construction (*con't*)

4. a (b | c)*



Of course, a human would design something simpler ...



But, we can automate production of the more complex one ...

NFA →DFA with Subset Construction

Need to build a simulation of the NFA

Note: s_i are sets of states of the NFA, which together constitute a single state in the simulating DFA

Two key functions

- *move*(s_i , $\underline{a}$) is set of states reachable from s_i by $\underline{a}$
- *ε-closure*(s_i) is set of states reachable from s_i by ε

The algorithm (sketch):

- Start state derived from s_0 of the NFA
- Take its ε-closure S_0 = ε-closure(s_0)
- For each state S, compute move(S, a) for each $a \in \Sigma$, and take its ε-closure
- Iterate until no more states are added

Sounds more complex than it is…

NFA →DFA with Subset Construction

The algorithm:

$s_0 \leftarrow \varepsilon$-closure({ q_0 })

add s_0 to S

while (S is still changing)
  for each $s_i \in S$
    for each $a \in \Sigma$
      $s_? \leftarrow \varepsilon$-closure(move($s_i$,a))
      if ($s_? \notin S$) then
        add $s_?$ to S as s_j
        T[s_i,a] $\leftarrow s_j$
      else
        T[s_i,a] $\leftarrow s_?$

Let's think about why this works

The algorithm halts:

1. *S* contains no duplicates (test before adding)
2. 2^Q is finite
3. while loop adds to *S*, but does not remove from *S (monotone)*

⇒ the loop halts

S contains all the reachable NFA states

It tries each symbol in each s_i.

It builds every possible NFA configuration.

⇒ *S and T form the DFA*

NFA →DFA with Subset Construction

Example of a *fixed-point* computation

- Monotone construction of some finite set
- Halts when it stops adding to the set
- Proofs of halting & correctness are similar
- These computations arise in many contexts

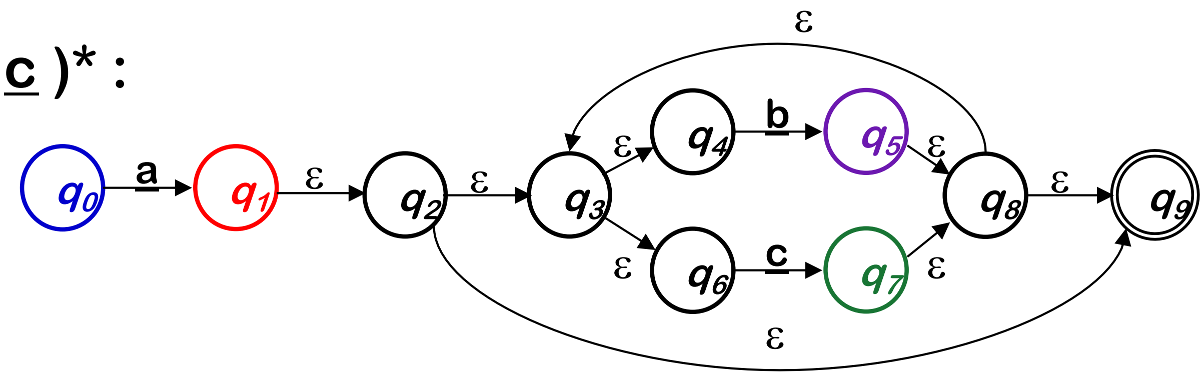
Other fixed-point computations

- Canonical construction of sets of LR(1) items
  - → Quite similar to the subset construction
- Classic data-flow analysis
  - → Solving sets of simultaneous set equations
- DFA minimization algorithm (coming up!)

We will see many more fixed-point computations

NFA →DFA with Subset Construction

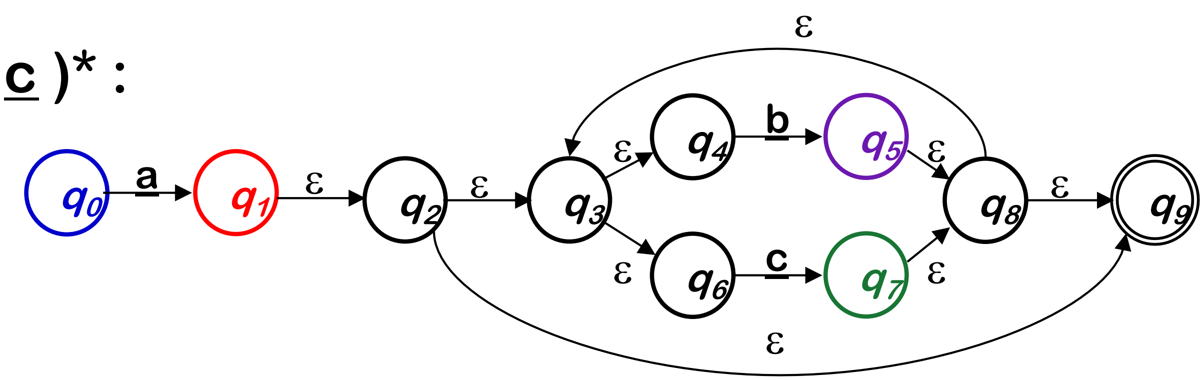
a (b | c)* :



Applying the subset construction:

NFA →DFA with Subset Construction

a (b | c)* :



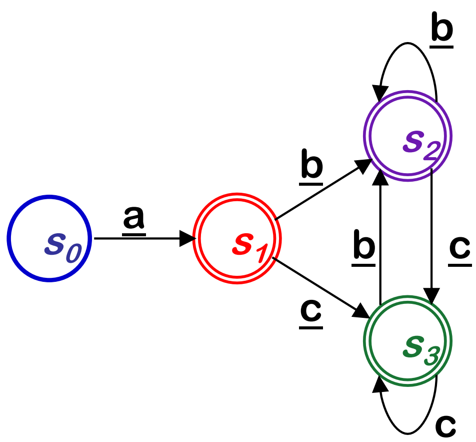
Applying the subset construction:

	NFA states	ε-closure(move(s,*)) a	b	c
s_0	q_0	$q_1, q_2, q_3, q_4, q_6, q_9$	none	none
s_1	$q_1, q_2, q_3, q_4, q_6, q_9$	none	$q_5, q_8, q_9, q_3, q_4, q_6$	$q_7, q_8, q_9, q_3, q_4, q_6$
s_2	$q_5, q_8, q_9, q_3, q_4, q_6$	none	s_2	s_3
s_3	$q_7, q_8, q_9, q_3, q_4, q_6$	none	s_2	s_3

Final states

NFA →DFA with Subset Construction

The DFA for a (b | c)*



δ	a	b	c
s_0	s_1	-	-
s_1	-	s_2	s_3
s_2	-	s_2	s_3
s_3	-	s_2	s_3

- Ends up smaller than the NFA
- All transitions are deterministic

Next class

More Lexical Analysis

Syntax Analysis (top-down parsing)

Read EaC: Chapter 3.1 - 3.3