RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

CS415 Compilers

Instruction Scheduling and Lexical Analysis

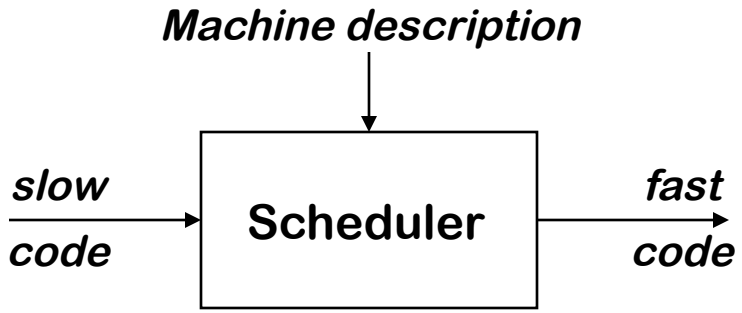
These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University

Instruction Scheduling (Engineer's View)

The Problem

Given a code fragment for some target machine and the latencies for each individual operation, reorder the operations to minimize execution time

The Concept



The task

- **Produce correct code**
- **Minimize wasted (idle) cycles**
- **Operate efficiently**

Data Dependences (stmt./instr. level)

Dependences ⇒ defined on memory locations / registers and not values

Statement/instruction b depends on statement/instruction a if there exists:

- true of flow dependence
  a writes a location/register that b later reads (RAW conflict)

- anti dependence
  a reads a location/register that b later writes (WAR conflict)

- output dependence
  a writes a location/register that b later writes (WAW conflict)

Dependences define ORDER CONSTRAINTS that need to be respected in order to generate correct code.

true	anti	output
a =	= a	a =
= a	a =	a =

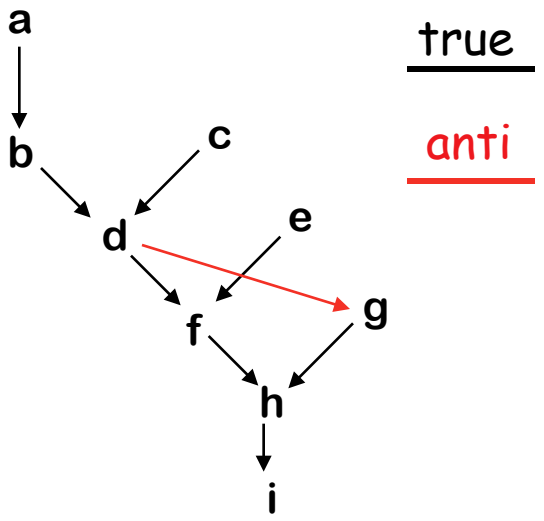
Instruction Scheduling (The Abstract View)

To capture properties of the code, build a dependence graph *G*

- Nodes $n \in G$ are operations with *type(n)* and *delay(n)*
- An edge $e = (n_1, n_2) \in G$ iff n_2 depends on n_1

```
a:  loadAI   r0,@w  ⇒ r1
b:  add      r1,r1  ⇒ r1
c:  loadAI   r0,@x  ⇒ r2
d:  mult     r1,r2  ⇒ r1
e:  loadAI   r0,@y  ⇒ r3
f:  mult     r1,r3  ⇒ r1
g:  loadAI   r0,@z  ⇒ r2
h:  mult     r1,r2  ⇒ r1
i:  storeAI  r1     ⇒ r0,@w
```

The Code



The Dependence Graph

(all output dependences are covered, i.e., are satisfied through other dependences)

Instruction Scheduling

The big picture

1. Build a dependence graph, *P*
2. Compute a *priority function* over the nodes in *P*
3. Use list scheduling to construct a schedule, one cycle at a time (can only issue/schedule at most one instructions per cycle)
   a. Use a queue of operations that are ready
   b. At each cycle
      I. Choose a ready operation and schedule it
      II. Update the ready queue

Local list scheduling

- The dominant algorithm for twenty years
- A greedy, heuristic, local technique

Local (Forward) List Scheduling

```
Cycle ← 1
Ready ← leaves of P
Active ← Ø

while (Ready ∪ Active ≠ Ø)
   if (Ready ≠ Ø) then
      remove an op from Ready
      S(op) ← Cycle
      Active ← Active ∪ op

   Cycle ← Cycle + 1

   for each op ∈ Active
       if (S(op) + delay(op) ≤ Cycle) then
          remove op from Active
          for each successor s of op in P
              if (s is ready) then
                 Ready ← Ready ∪ s
```

Removal in priority order

op has completed execution

If successor's operands are ready, put it on Ready

Scheduling Example

Operation	Cycles
load	3
loadI	1
loadAI	3
store	3
storeAI	3
add	1
mult	2
fadd	1
fmult	2
shift	1
branch	0 to 8

- Loads & stores may or may not block
  - > Non-blocking ⇒fill those issue slots
- Branches typically have delay slots
  - > Fill slots with operations unrelated to branch condition evaluation
- Branch Prediction may hide branch latencies (hardware feature)

Build a simple local scheduler (basic block)

- non-blocking loads & stores
- different latencies load/store vs. arith. etc. operations
- different heuristics
- forward / backward scheduling

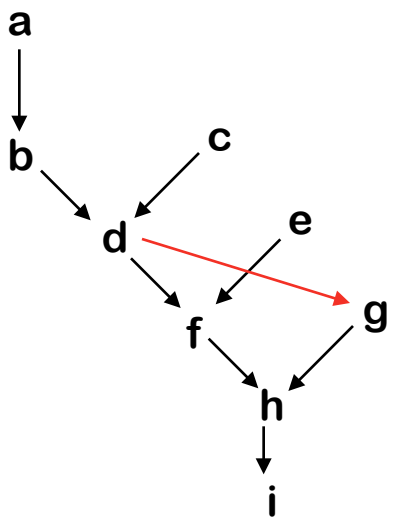
Scheduling Example

Operation	Cycles
load	3
loadI	1
loadAI	3
storeAI	3
add	1
mult	2

1. Build the dependence graph

```
1   a:  loadAI   r0,@w  ⇒ r1
4   b:  add      r1,r1  ⇒ r1
5   c:  loadAI   r0,@x  ⇒ r2
8   d:  mult     r1,r2  ⇒ r1
9   e:  loadAI   r0,@y  ⇒ r3
12  f:  mult     r1,r3  ⇒ r1
13  g:  loadAI   r0,@z  ⇒ r2
16  h:  mult     r1,r2  ⇒ r1
18  i:  storeAI  r1     ⇒ r0,@w
21
```

The Code



true

anti

The Dependence Graph

⇒ **20 cycles**

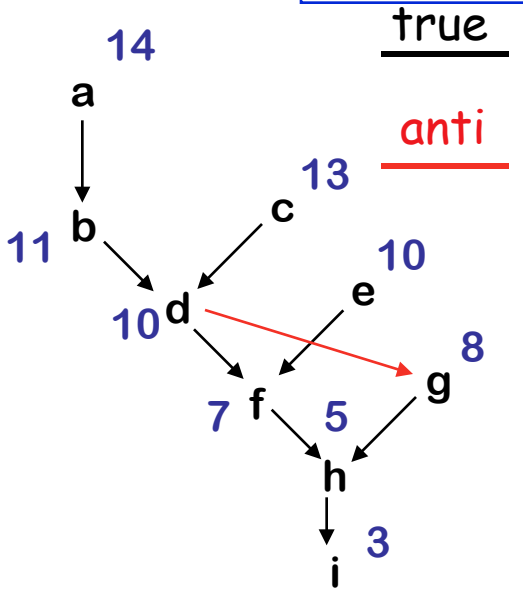
Scheduling Example

Operation	Cycles
load	3
loadI	1
loadAI	3
storeAI	3
add	1
mult	2

1. Build the dependence graph
2. Determine priorities: longest latency-weighted path

```
a:  loadAI   r0,@w  ⇒ r1
b:  add      r1,r1  ⇒ r1
c:  loadAI   r0,@x  ⇒ r2
d:  mult     r1,r2  ⇒ r1
e:  loadAI   r0,@y  ⇒ r3
f:  mult     r1,r3  ⇒ r1
g:  loadAI   r0,@z  ⇒ r2
h:  mult     r1,r2  ⇒ r1
i:  storeAI  r1     ⇒ r0,@w
```

The Code



The Dependence Graph

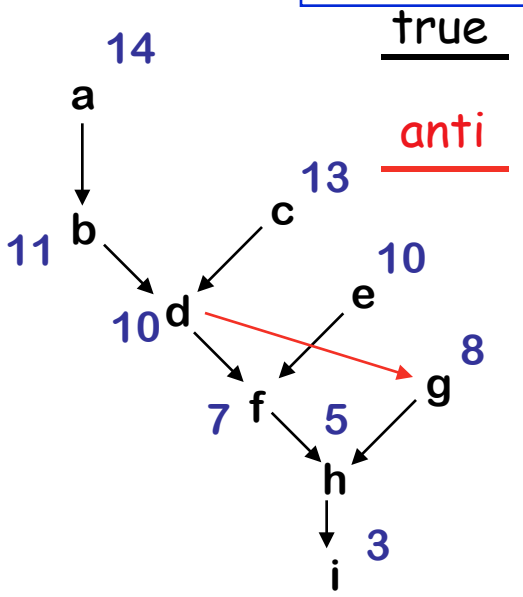
Scheduling Example

Operation	Cycles
load	3
loadI	1
loadAI	3
storeAI	3
add	1
mult	2

1. Build the dependence graph
2. Determine priorities: longest latency-weighted path

```
a:  loadAI   r0,@w  ⇒ r1
b:  add      r1,r1  ⇒ r1
c:  loadAI   r0,@x  ⇒ r2
d:  mult     r1,r2  ⇒ r1
e:  loadAI   r0,@y  ⇒ r3
f:  mult     r1,r3  ⇒ r1
g:  loadAI   r0,@z  ⇒ r2
h:  mult     r1,r2  ⇒ r1
i:  storeAI  r1     ⇒ r0,@w
```

The Code



The Dependence Graph

Note: Here we assume that operation has to finish to satisfy an anti dependence.
Our ILOC simulator takes only one cycle to satisfy an anti dependence since read-stage is executed before write stage

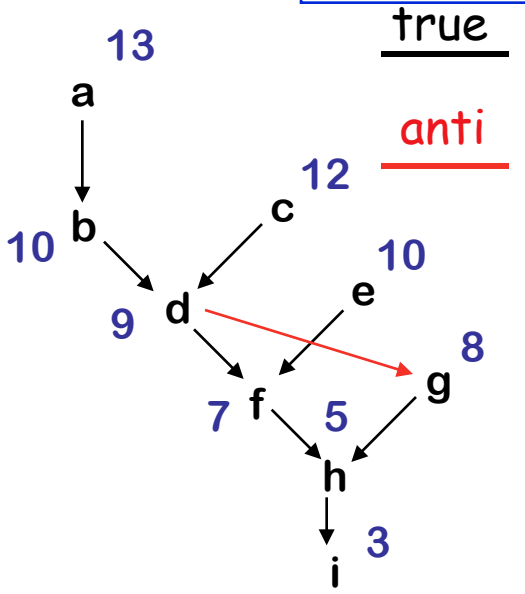
Scheduling Example

Operation	Cycles
load	3
loadI	1
loadAI	3
storeAI	3
add	1
mult	2

1. Build the dependence graph
2. Determine priorities: longest latency-weighted path

```
a:  loadAI   r0,@w  ⇒ r1
b:  add      r1,r1  ⇒ r1
c:  loadAI   r0,@x  ⇒ r2
d:  mult     r1,r2  ⇒ r1
e:  loadAI   r0,@y  ⇒ r3
f:  mult     r1,r3  ⇒ r1
g:  loadAI   r0,@z  ⇒ r2
h:  mult     r1,r2  ⇒ r1
i:  storeAI  r1     ⇒ r0,@w
```

The Code



The Dependence Graph

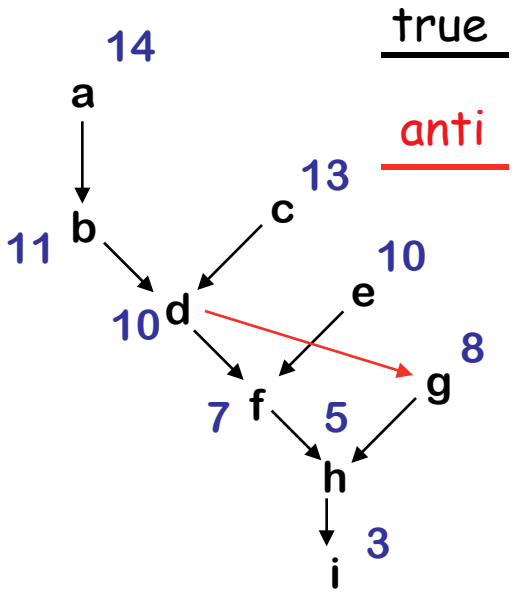
Our ILOC simulator **takes only one cycle to satisfy an anti dependence** since read-stage is executed before write stage (EaC).

Scheduling Example

1. Build the dependence graph
2. Determine priorities: longest latency-weighted path
3. Perform list scheduling (forward)

```
a:  loadAI   r0,@w  ⇒ r1
b:  add      r1,r1  ⇒ r1
c:  loadAI   r0,@x  ⇒ r2
d:  mult     r1,r2  ⇒ r1
e:  loadAI   r0,@y  ⇒ r3
f:  mult     r1,r3  ⇒ r1
g:  loadAI   r0,@z  ⇒ r2
h:  mult     r1,r2  ⇒ r1
i:  storeAI  r1     ⇒ r0,@w
```

The Code



The Dependence Graph

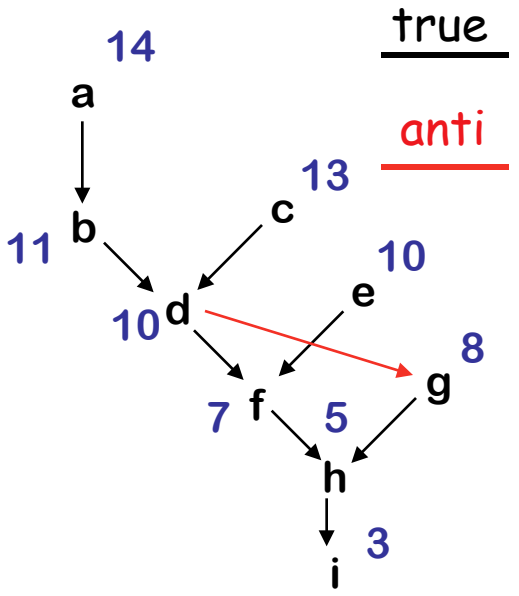
Scheduling Example

1. Build the dependence graph
2. Determine priorities: longest latency-weighted path
3. Perform list scheduling (forward)

```
1   a:  loadAI   r0,@w   ⇒ r1
2   c:  loadAI   r0,@x   ⇒ r2
3   e:  loadAI   r0,@y   ⇒ r3
4   b:  add      r1,r1   ⇒ r1
5   d:  mult     r1,r2   ⇒ r1
7   g:  loadAI   r0,@z   ⇒ r2
8   f:  mult     r1,r3   ⇒ r1
10  h:  mult     r1,r2   ⇒ r1
12  i:  storeAI  r1      ⇒ r0,@w
15
```

The Code

⇒ 14 cycles



The Dependence Graph

Our ILOC simulator takes only one cycle to satisfy an anti dependence

More on Scheduling

Forward list scheduling

- start with available ops
- work forward
- ready ⇒ all operands available

Backward list scheduling

- start with no successors
- work backward
- ready ⇒ latency covers operands

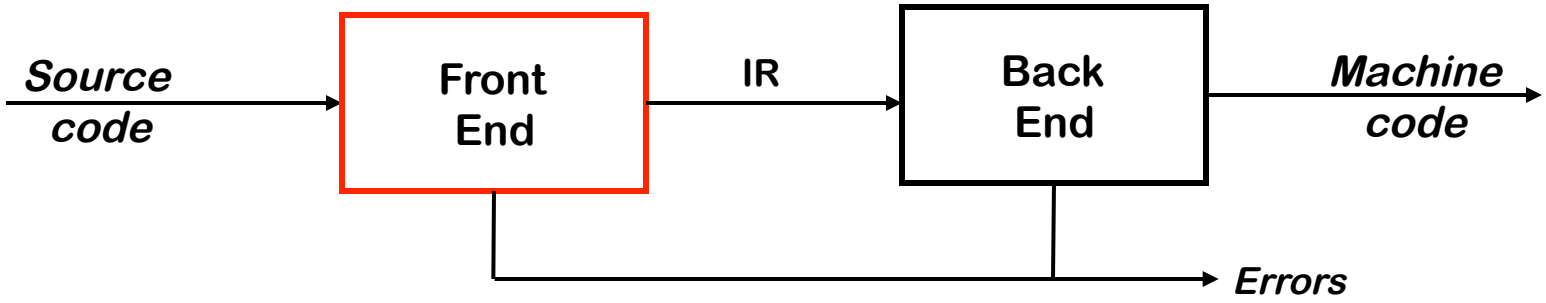
Different heuristics (forward) based on Dependence Graph

1. Longest latency weighted path to root (⇒ critical path)
2. Highest latency instructions (⇒ more overlap)
3. Most immediate successors (⇒ create more candidates)
4. Most descendents (⇒ create more candidates)
5. …

Interactions with register allocation

- perform dynamic register renaming (⇒ may require spill code)
- move life ranges around (⇒ may remove or require spill code)
- …

The Front End

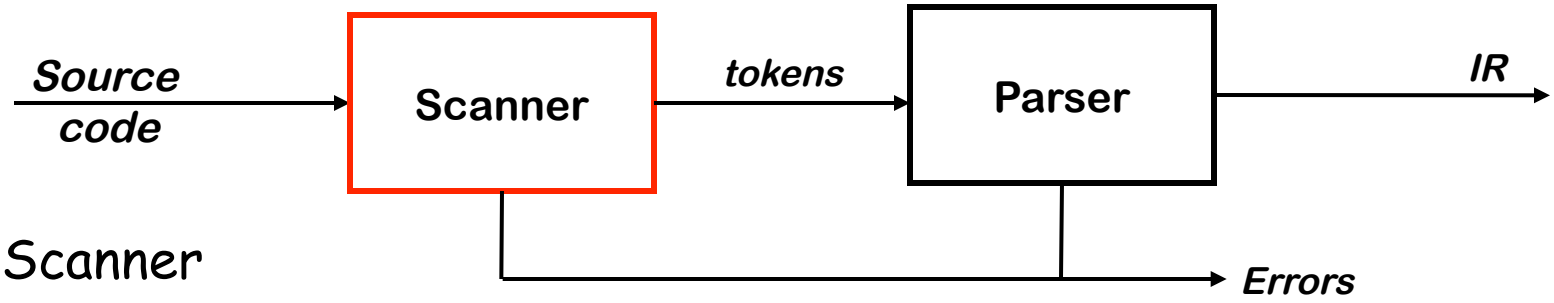


The purpose of the front end is to deal with the input language

- Perform a membership test: code ∈ source language?
- Is the program well-formed (semantically) ?
- Build an IR version of the code for the rest of the compiler

The front end is not monolithic

The Front End



Scanner

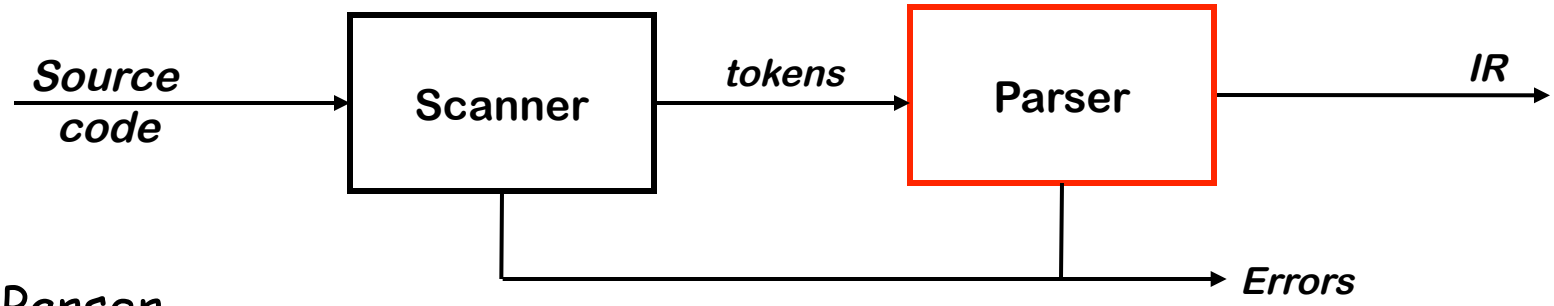
- Maps stream of characters into words
  - → Basic unit of syntax
  - → x = x + y ; *becomes*

    <id,x> <eq,=> <id,x> <pl,+> <id,y> <sc,; >
- Characters that form a word are its *lexeme*
- Its *part of speech* (or *syntactic category*) is called its *token type*
- Scanner discards white space & (often) comments

Speed is an issue in scanning

⇒ use a specialized recognizer

The Front End



Parser

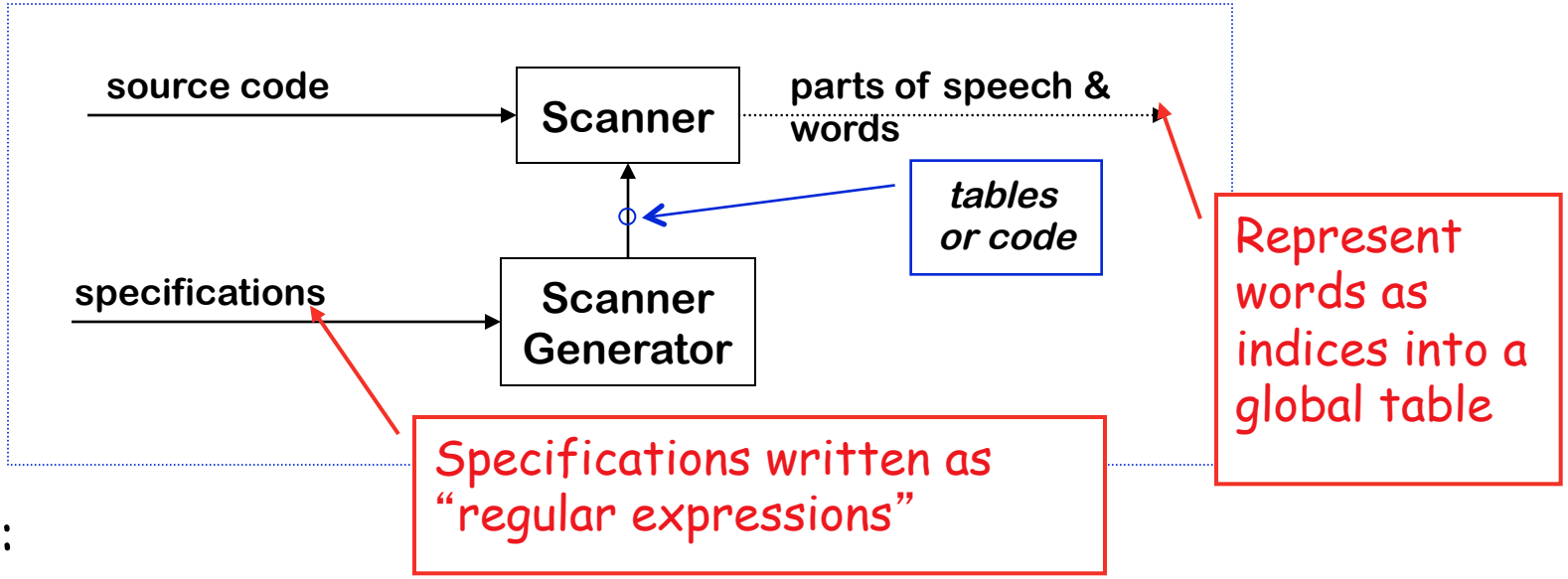
- Checks stream of classified words (*parts of speech*) for grammatical correctness
- Determines if code is syntactically well-formed
- Builds an IR representation of the code

We'll get to parsing in the next lectures

The Big Picture

Why study lexical analysis?

- We want to avoid writing scanners by hand



Goals:

→ To simplify specification & implementation of scanners

→ To understand the underlying techniques and technologies

Regular Expressions

Lexical patterns form a *regular language*

**** any finite language is regular ****

Regular expressions (REs) describe regular languages

Ever type "rm *.o a.out" ?

Regular Expression (over alphabet Σ)

- ε is a RE denoting the set {ε}
- If a is in Σ, then a is a RE denoting {a}
- If *x* and *y* are REs denoting *L(x)* and *L(y)* then
  - → *x* | *y* is an RE denoting *L(x)* ∪ *L(y)*
  - → *xy* is an RE denoting *L(x)L(y)*
  - → x^* is an RE denoting *L(x)**

Precedence: *closure*, then *concatenation*, then *alternation*

Set Operations

Operation	Definition
Union of L and M Written L ∪ M	$L \cup M = \{s \mid s \in L \text{ or } s \in M\}$
Concatenation of L and M Written LM	$LM = \{st \mid s \in L \text{ and } t \in M\}$
Kleene closure of L Written L^*	$L^* = \bigcup_{0 \le i \le \infty} L^i$
Positive Closure of L Written L^+	$L^+ = \bigcup_{1 \le i \le \infty} L^i$

These definitions should be well known

Examples of Regular Expressions

Identifiers:

Letter → (a|b|c| … |z|A|B|C| … |Z)

Digit → (0|1|2| … |9)

Identifier → *Letter* (*Letter* | *Digit*)*

Numbers:

Integer → (+|-|ε) (0| (1|2|3| … |9)(*Digit* *))

Decimal → *Integer* . *Digit* *

Real → (*Integer* | *Decimal*) E (+|-|ε) *Digit* *

Complex → (*Real* , *Real*)

Numbers can get much more complicated!

Regular Expressions (the point)

Regular expressions can be used to specify the words to be translated to parts of speech by a lexical analyzer

Using results from automata theory and theory of algorithms, we can automatically build recognizers from regular expressions

⇒ We study REs and associated theory to automate scanner construction !

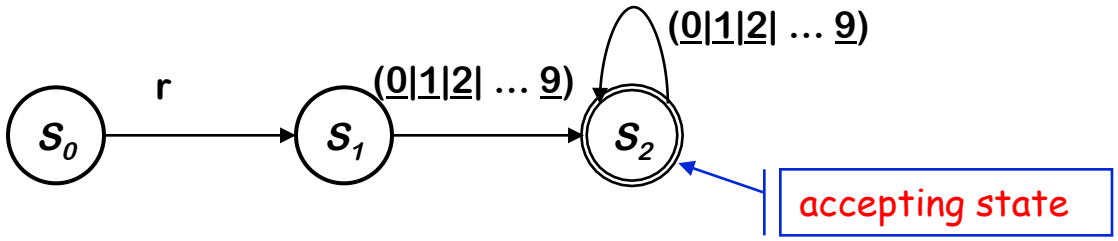
Example

Consider the problem of recognizing ILOC register names

Register → r (0|1|2| … | 9) (0|1|2| … | 9)*

- Allows registers of arbitrary number
- Requires at least one digit

RE corresponds to a recognizer (or DFA)



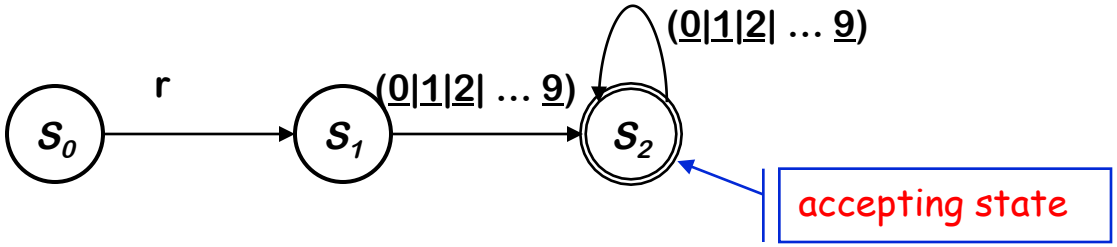
Recognizer for *Register*

Transitions on other inputs go to an error state, s_e

Example (continued)

DFA operation

- Start in state S_0 & take transitions on each input character
- DFA accepts a word *x iff* x leaves it in a final state (S_2)



Recognizer for *Register*

So,

- r17 takes it through s_0, s_1, s_2 and accepts
- r takes it through s_0, s_1 and fails
- a takes it straight to error state s_e (not shown here)

Example (continued)

To be useful, recognizer must turn into code

```
Char ← next character
State ← s0

while (Char ≠ EOF)
    State ← δ(State,Char)
     Char ← next character

if (State is a final state )
    then report success
    else  report failure
```

Skeleton recognizer

δ	r	0,1,2,3,4, 5,6,7,8,9	*All others*
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

Table encoding RE

Example (continued)

To be useful, recognizer must turn into code

```
Char ← next character
State ← s0

while (Char ≠ EOF)
    State ← δ(State,Char)
    perform specified action
     Char ← next character

if (State is a final state )
    then report success
    else  report failure
```

Skeleton recognizer

δ	r	0,1,2,3,4, 5,6,7,8,9	*All others*
s_0	s_1 *start*	s_e *error*	s_e *error*
s_1	s_e *error*	s_2 add	s_e *error*
s_2	s_e *error*	s_2 add	s_e *error*
s_e	s_e *error*	s_e *error*	s_e *error*

Table encoding RE

What if we need a tighter specification?

r *Digit Digit** allows arbitrary numbers

- Accepts r00000
- Accepts r99999
- What if we want to limit it to r0 through r31 ?

What if we need a tighter specification?

r *Digit Digit** allows arbitrary numbers

- Accepts r00000
- Accepts r99999
- What if we want to limit it to r0 through r31 ?

Need a more complicated regular expression

→ *Register* → r ((0|1|2) (*Digit* | ε) | (4|5|6|7|8|9) | (3|30|31))

→ *Register* → r0|r1|r2| … |r31|r00|r01|r02| … |r09

What if we need a tighter specification?

r *Digit Digit** allows arbitrary numbers

- Accepts r00000
- Accepts r99999
- What if we want to limit it to r0 through r31 ?

Need a more complicated regular expression

→ *Register* → r ((0|1|2) (*Digit* | ε) | (4|5|6|7|8|9) | (3|30|31))

→ *Register* → r0|r1|r2| … |r31|r00|r01|r02| … |r09

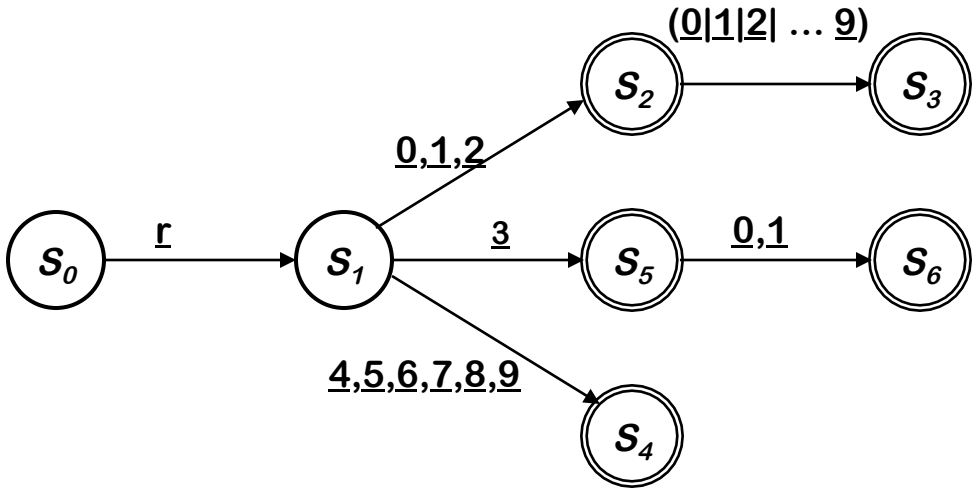
Produces a more complex DFA

- Has more states
- Same cost per transition
- Same basic implementation

Tighter register specification (continued)

The DFA for

Register → r ((0|1|2) (*Digit* | ε) | (4|5|6|7|8|9) | (3|30|31))



- Accepts a more constrained set of registers
- Same set of actions, more states

Tighter register specification (continued)

δ	r	0,1	2	3	4-9	All others
s_0	s_1	s_e	s_e	s_e	s_e	s_e
s_1	s_e	s_2	s_2	s_5	s_4	s_e
s_2	s_e	s_3	s_3	s_3	s_3	s_e
s_3	s_e	s_e	s_e	s_e	s_e	s_e
s_4	s_e	s_e	s_e	s_e	s_e	s_e
s_5	s_e	s_6	s_e	s_e	s_e	s_e
s_6	s_e	s_e	s_e	s_e	s_e	s_e
s_e	s_e	s_e	s_e	s_e	s_e	s_e

Runs in the same skeleton recognizer

Table encoding RE for the tighter register specification

More Regular Expressions

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

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

  Cons → (b|c|d|f|g|h|j|k|l|m|n|p|q|r|s|t|v|w|x|y|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

  (0 | 1)* 1

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

  Cons → (b|c|d|f|g|h|j|k|l|m|n|p|q|r|s|t|v|w|x|y|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

  (0 | 1)* 1

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

  Cons → (b|c|d|f|g|h|j|k|l|m|n|p|q|r|s|t|v|w|x|y|z)
  *Cons** a *Cons** e *Cons** i *Cons** o *Cons** u *Cons**

- 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

  (0 | 1)* 1

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

  Cons → (b|c|d|f|g|h|j|k|l|m|n|p|q|r|s|t|v|w|x|y|z)
  *Cons** a *Cons** e *Cons** i *Cons** o *Cons** u *Cons**

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

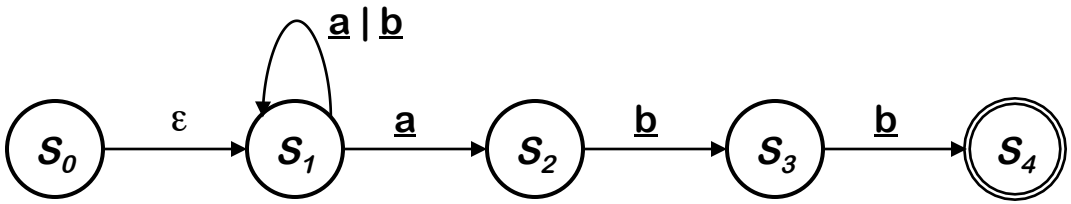
  Next class

Non-deterministic Finite Automata

Each RE corresponds to a non*deterministic finite automaton* (NFA)

- May be challenging 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 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 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 *(define all parts of speech)*
- You could build one in a weekend!

Next class

More Lexical Analysis; Syntax Analysis

Read EaC: Chapters 2.1 – 2.5; 3.1 – 3.3

Homework Problem Set 2 is posted.

No class on Feb 25, Thursday.