CS415 Compilers
Instruction Scheduling &
Lexical Analysis

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

• Second homework has been posted (bottom-up allocation & instruction scheduling); due on Friday, February 15, 11:59pm

• First project will be on list scheduling. Will be posted this week.
Instruction Scheduling

EaC  Chapter 12
12.1 - 12.3

Part of the compiler’s back end

Instruction Selection → Register Allocation → Instruction Scheduling → Machine code

Errors

m register  k register
Dependences ⇒ defined on memory locations / registers

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

- **true** of flow dependence
  \( a \) writes a location/register that \( b \) later reads \hspace{1cm} (RAW conflict)

- **anti** dependence
  \( a \) reads a location/register that \( b \) later writes \hspace{1cm} (WAR conflict)

- **output** dependence
  \( a \) writes a location/register that \( b \) later writes \hspace{1cm} (WAW conflict)

Dependences define ORDER CONSTRAINTS that need to be respected in order to generate correct code.
To capture properties of the code, build a **precedence graph** $G$

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

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

**The Precedence/Dependence Graph**

All other dependences (output & anti) are covered, i.e., are satisfied through the dependencies shown.
### Example latencies

**Machine model (ISA) with different latencies/delays**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycles (latency/delay)</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>3</td>
</tr>
<tr>
<td>loadl</td>
<td>1</td>
</tr>
<tr>
<td>loadAl</td>
<td>3</td>
</tr>
<tr>
<td>store</td>
<td>3</td>
</tr>
<tr>
<td>storeAl</td>
<td>3</td>
</tr>
<tr>
<td>add</td>
<td>1</td>
</tr>
<tr>
<td>mult</td>
<td>2</td>
</tr>
<tr>
<td>fadd</td>
<td>1</td>
</tr>
<tr>
<td>fmult</td>
<td>2</td>
</tr>
<tr>
<td>shift</td>
<td>1</td>
</tr>
</tbody>
</table>
A **correct schedule** $S$ maps each $n \in N$ into a non-negative integer representing its **cycle number** such that

1. $S(n) \geq 0$, for all $n \in N$, obviously
2. If $(n_1, n_2) \in E$, $S(n_1) + \text{delay}(n_1) \leq S(n_2)$
3. For each type $t$, there are no more operations of type $t$ in any cycle than the target machine can issue;
   (Note: we only use a single type here - single pipeline)

The **length** of a schedule $S$, denoted $L(S)$, is

$$L(S) = \max_{n \in N} (S(n) + \text{delay}(n))$$

The goal is to find the shortest possible correct schedule. $S$ is **time-optimal** if $L(S) \leq L(S_1)$, for all other schedules $S_1$

Note: we are trying to minimize execution time here.
Local (Forward) List Scheduling

\[
\begin{align*}
\text{Cycle} & \leftarrow 0 \\
\text{Ready} & \leftarrow \text{leaves of } P \\
\text{Active} & \leftarrow \emptyset \\
\text{while } (\text{Ready} \cup \text{Active} \neq \emptyset) \quad & \text{\quad \quad \quad \quad \quad \quad \text{Removal in priority order}} \\
& \quad \text{if } (\text{Ready} \neq \emptyset) \text{ then} \\
& \quad \quad \text{remove an op from Ready} \\
& \quad \quad S(op) \leftarrow \text{Cycle} \\
& \quad \quad \text{Active} \leftarrow \text{Active} \cup \text{op} \\
\text{Cycle} & \leftarrow \text{Cycle} + 1 \\
\text{for each } & \text{op } \in \text{Active} \\
& \quad \text{if } (S(op) + \text{delay}(op) \leq \text{Cycle}) \text{ then} \\
& \quad \quad \text{remove op from Active} \\
& \quad \quad \text{for each successor s of op in P} \\
& \quad \quad \text{if } (s \text{ is ready}) \text{ then} \\
& \quad \quad \quad \text{Ready} \leftarrow \text{Ready} \cup s \\
\end{align*}
\]
### Scheduling Example

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>3</td>
</tr>
<tr>
<td>loadI</td>
<td>1</td>
</tr>
<tr>
<td>loadAI</td>
<td>3</td>
</tr>
<tr>
<td>store</td>
<td>3</td>
</tr>
<tr>
<td>storeAI</td>
<td>3</td>
</tr>
<tr>
<td>add</td>
<td>1</td>
</tr>
<tr>
<td>mult</td>
<td>2</td>
</tr>
<tr>
<td>fadd</td>
<td>1</td>
</tr>
<tr>
<td>fmult</td>
<td>2</td>
</tr>
<tr>
<td>shift</td>
<td>1</td>
</tr>
</tbody>
</table>

- **Loads & stores may or may not block**
  - Non-blocking \(\Rightarrow\) fill those issue slots
- **Branches typically have delay slots**
  - Fill slots with operations unrelated to branch condition evaluation
  - Percolates branch upward
- **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
1. Build the dependence graph

The Code

\[
\begin{align*}
S(n): & \\
0 & a: \text{loadAI} \quad r_0, @w & \rightarrow r_1 \\
3 & b: \text{add} \quad r_1, r_1 & \rightarrow r_1 \\
4 & c: \text{loadAI} \quad r_0, @x & \rightarrow r_2 \\
7 & d: \text{mult} \quad r_1, r_2 & \rightarrow r_1 \\
8 & e: \text{loadAI} \quad r_0, @y & \rightarrow r_3 \\
11 & f: \text{mult} \quad r_1, r_3 & \rightarrow r_1 \\
12 & g: \text{loadAI} \quad r_0, @z & \rightarrow r_2 \\
15 & h: \text{mult} \quad r_1, r_2 & \rightarrow r_1 \\
17 & i: \text{storeAI} \quad r_1 & \rightarrow r_0, @w \\
20 & & \\
\end{align*}
\]

⇒ 20 cycles

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

The Code

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

The Code

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a:</td>
<td>loadAI</td>
<td>r0,@w ⇒ r1</td>
</tr>
<tr>
<td>b:</td>
<td>add</td>
<td>r1,r1 ⇒ r1</td>
</tr>
<tr>
<td>c:</td>
<td>loadAI</td>
<td>r0,@x ⇒ r2</td>
</tr>
<tr>
<td>d:</td>
<td>mult</td>
<td>r1,r2 ⇒ r1</td>
</tr>
<tr>
<td>e:</td>
<td>loadAI</td>
<td>r0,@y ⇒ r3</td>
</tr>
<tr>
<td>f:</td>
<td>mult</td>
<td>r1,r3 ⇒ r1</td>
</tr>
<tr>
<td>g:</td>
<td>loadAI</td>
<td>r0,@z ⇒ r2</td>
</tr>
<tr>
<td>h:</td>
<td>mult</td>
<td>r1,r2 ⇒ r1</td>
</tr>
<tr>
<td>i:</td>
<td>storeAI</td>
<td>r1 ⇒ r0,@w</td>
</tr>
</tbody>
</table>

The Dependence Graph

Note: Here we assume that an 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 (EaC).
1. Build the dependence graph
2. Determine priorities: longest latency-weighted path

The Code

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 Dependence Graph

Note: Here we assume that an 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 (EaC).
Scheduling Example

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

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

The Dependence Graph

We assume full latency for anti-dependences here
1. Build the dependence graph
2. Determine priorities: longest latency-weighted path
3. Perform list scheduling (forward)

The Dependence Graph

We assume full latency for anti-dependences here
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 (Note: we are not doing this)
- perform dynamic register renaming (⇒ may require spill code)
- move life ranges around (⇒ may remove or require spill code)
- ...

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

- Perform a membership test: \( \text{code} \in \text{source language?} \)
- 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
    \[ \langle \text{id}, x \rangle \langle \text{eq,=} \rangle \langle \text{id,} x \rangle \langle \text{pl,}+ \rangle \langle \text{id,} y \rangle \langle \text{sc,}; \rangle \]
- 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
- Guides checking at deeper levels than syntax
- Builds an IR representation of the code

We’ll get to parsing in the next lectures
The Big Picture

• Language syntax is specified with *parts of speech*, not *words*
• Syntax checking matches *parts of speech* against a grammar
• Here is an example context free grammar (CFG) $G$:

$G$ in BNF form

1. $goal \rightarrow expr$
2. $expr \rightarrow expr \ op \ term$
3. $\mid \ term$
4. $term \rightarrow number$
5. $\mid \ id$
6. $op \rightarrow +$
7. $\mid -$ 

$S = goal$
$T = \{ \text{number, id, +, -} \}$
$N = \{ goal, expr, term, op \}$
$P = \{ 1, 2, 3, 4, 5, 6, 7 \}$

$G = (S, T, N, P)$
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

Specifications written as “regular expressions”
Represent words as indices into a global table

Source code → Scanner Generator → Scanner → parts of speech & words (tokens)

Specifications

“Regular expressions”
Lexical patterns form a **regular language**

*** any finite language is regular ***

Regular expressions (REs) describe regular languages.

Regular Expression (over alphabet \( \Sigma \))

- \( \varepsilon \) is a RE denoting the set \( \{ \varepsilon \} \)
- If “a” is in \( \Sigma \), 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 \( L(x) \cup L(y) \)
  - \( xy \) is an RE denoting \( L(x)L(y) \)
  - \( x^* \) is an RE denoting \( L(x)^* \)
  - \( (x) \) is an RE denoting \( L(x) \)

Ever type “rm *.o a.out” ?

Precedence is closure, then concatenation, then alternation
### Set Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Union of L and M</strong></td>
<td>( L \cup M = { s \mid s \in L \text{ or } s \in M } )</td>
</tr>
<tr>
<td><strong>Written L \cup M</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Concatenation of L and M</strong></td>
<td>( LM = { st \mid s \in L \text{ and } t \in M } )</td>
</tr>
<tr>
<td><strong>Written LM</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Kleene closure of L</strong></td>
<td>( L^* = \bigcup_{0 \leq i \leq \infty} L^i )</td>
</tr>
<tr>
<td><strong>Written L^</strong>*</td>
<td></td>
</tr>
<tr>
<td><strong>Positive Closure of L</strong></td>
<td>( L^+ = \bigcup_{1 \leq i \leq \infty} L^i )</td>
</tr>
<tr>
<td><strong>Written L^+</strong></td>
<td></td>
</tr>
</tbody>
</table>

**These definitions should be well known**
Examples of Regular Expressions

**Identifiers:**

\[
\begin{align*}
\text{Letter} & \rightarrow (a|b|c| \ldots |z|A|B|C| \ldots |Z) \\
\text{Digit} & \rightarrow (0|1|2| \ldots |9) \\
\text{Identifier} & \rightarrow \text{Letter} ( \text{Letter} | \text{Digit} )^* \\
\end{align*}
\]

**Numbers:**

\[
\begin{align*}
\text{Integer} & \rightarrow (\pm|\varepsilon)(0|1|2|3| \ldots |9)(\text{Digit}^*) \\
\text{Decimal} & \rightarrow \text{Integer} \cdot \text{Digit}^* \\
\text{Real} & \rightarrow (\text{Integer} | \text{Decimal})E(\pm|\varepsilon)\text{Digit}^* \\
\text{Complex} & \rightarrow (\text{Real} \pm \text{Real}) \\
\end{align*}
\]

*Numbers can get much more complicated!*
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!
Consider the problem of recognizing ILOC register names

\[ \text{Register} \to r \ (0|1|2| ... | 9) \ (0|1|2| ... | 9)^* \]

- 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, \( s_e \)
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 ($S_2$)

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)
To be useful, recognizer must turn into code

Char ← next character
State ← $s_0$

while (Char ≠ EOF)
    State ← $\delta$(State, Char)
    Char ← next character

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

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>r</th>
<th>0,1,2,3,4,5,6,7,8,9</th>
<th>All others</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
</tbody>
</table>

Skeleton recognizer

Table encoding RE

Example (continued)
To be useful, recognizer must turn into code

Char ← next character
State ← $s_0$

while (Char ≠ EOF)
    State ← $\delta$(State, Char)
    *perform specified action*
    Char ← next character

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

Char ← next character
State ← $s_0$

while (Char ≠ EOF)
    State ← $\delta$(State, Char)
    perform specified action
    Char ← next character

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

---

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>r</th>
<th>0,1,2,3,4,5,6,7,8,9</th>
<th>All others</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td>$s_1$ start</td>
<td>$s_e$ error</td>
<td>$s_e$ error</td>
</tr>
<tr>
<td>$s_1$</td>
<td>$s_e$ error</td>
<td>$s_2$ add</td>
<td>$s_e$ error</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$s_e$ error</td>
<td>$s_2$ add</td>
<td>$s_e$ error</td>
</tr>
<tr>
<td>$s_e$</td>
<td>$s_e$ error</td>
<td>$s_e$ error</td>
<td>$s_e$ error</td>
</tr>
</tbody>
</table>

*Skeleton recognizer*

*Table encoding RE*
r Digit Digit* allows arbitrary numbers

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

Write a tighter 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
The DFA for

\[ \text{Register} \rightarrow r \ ( (0\mid 1\mid 2) \ (\text{Digit} \mid \varepsilon) \ | \ (4\mid 5\mid 6\mid 7\mid 8\mid 9) \ | \ (3\mid 30\mid 31)) \]

- Accepts a more constrained set of registers
- Same set of actions, more states
### Tighter register specification (continued)

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>$r$</th>
<th>0,1</th>
<th>2</th>
<th>3</th>
<th>4-9</th>
<th>All others</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_5$</td>
<td>$s_4$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$s_e$</td>
<td>$s_3$</td>
<td>$s_3$</td>
<td>$s_3$</td>
<td>$s_3$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_3$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_4$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_5$</td>
<td>$s_e$</td>
<td>$s_6$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_6$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
</tbody>
</table>

*Table encoding RE for the tighter register specification*

Runs in the same skeleton recognizer
The scanner is the first stage in the front end
- Specifications can be expressed using regular expressions
- Build tables and code from a DFA
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
  \((0 | 1)^*1\)

- All strings over lowercase letters where the vowels (a,e,i,o, & u) occur exactly once, in ascending order
  \[\text{Cons} \rightarrow (\text{b|c|d|f|g|h|j|k|l|m|n|p|q|r|s|t|v|w|x|y|z})\]
  \[\text{Cons}^* \text{ a Cons}^* \text{ e Cons}^* \text{ i Cons}^* \text{ o Cons}^* \text{ u Cons}^*\]

- 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
  \[(0 \mid 1)^*1\]

• All strings over lowercase letters where the vowels (a,e,i,o, & u) occur exactly once, in ascending order
  \[Cons \rightarrow (b\mid c\mid d\mid f\mid g\mid h\mid j\mid k\mid l\mid m\mid n\mid p\mid q\mid r\mid s\mid t\mid v\mid w\mid x\mid y\mid z)\]
  \[Cons^* \ a \ Cons^* \ e \ Cons^* \ i \ Cons^* \ o \ Cons^* \ u \ Cons^*\]

• All strings of 1s and 0s that do not contain three 0s in a row:
  \[(1^* ( \varepsilon \mid 01 \mid 001 ) 1^*)^* ( \varepsilon \mid 0 \mid 00 )\]
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 \((a \mid b)^* \text{abb}\)?

This is a little different

• \(S_0\) has a transition on \(\varepsilon\)
• \(S_1\) has two transitions on \(a\)

This is a non-deterministic finite automaton (NFA)
Non-deterministic Finite Automata

- An NFA accepts a string $x$ iff $\exists$ a path though the transition graph from $s_0$ to a final state such that the edge labels spell $x$
- Transitions on $\varepsilon$ consume no input
- To “run” the NFA, start in $s_0$ and \textit{guess} the right transition at each step
  \begin{itemize}
  \item Always guess correctly
  \item If some sequence of correct guesses accepts $x$ then accept
  \end{itemize}

Why study NFAs?
- They are the key to automating the RE$\to$DFA construction
- We can paste together NFAs with $\varepsilon$-transitions
DFA is a special case of an NFA

• DFA has no \( \varepsilon \) transitions
• DFA’s transition function is single-valued
• Same rules will work

DFA can be simulated with an NFA

\[ \rightarrow \text{Obviously} \]

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 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 (define all parts of speech)
- 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
Key idea

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

---

Ken Thompson, CACM, 1968
More Lexical Analysis
Lexical Analysis

Read EaC: Chapters 2.1 – 2.5;