Lexical Analysis
Outline

• Informal sketch of lexical analysis
  - Identifies tokens in input string

• Issues in lexical analysis
  - Lookahead
  - Ambiguities

• Specifying lexers
  - Regular expressions
  - Examples of regular expressions
Lexical Analysis

• What do we want to do? Example:
  
  ```plaintext
  if (i == j)
    Z = 0;
  else
    Z = 1;
  ```

• The input is just a string of characters:
  
  ```plaintext
  \tif (i == j)\n  \tZ = 0;\n  \telse\n  \tZ = 1;
  ```

• Goal: Partition input string into substrings
  - Note: humans have visual clues compiler doesn't: it just sees a sequence of bytes
Lexical Analysis

• What do we want to do? Example:
  ```c
  if (i == j)
      Z = 0;
  else
      Z = 1;
  ```

• The input is just a string of characters:
  ```c
  \tif (i == j)\n      tz = 0;\n  \else\n      tz = 1;
  ```

• Goal: Partition input string into substrings
  - And it’s not just substrings: it’s *tokens*
What’s a Token?

• A syntactic category
  - In English:
    
    noun, verb, adjective, ...
  
  - In a programming language:
    
    Identifier, Integer, Keyword, Whitespace, ...
    
    Also: individual characters: {, }, (,), ;, ...
    
    Classifies lexeme according to its role
Tokens

• Tokens correspond to sets of strings.
  - But tokens are NOT sets of strings. Token itself is typically a tuple, e.g., <token class, lexeme>

• Identifier: strings of letters or digits, starting with a letter

• Integer: a non-empty string of digits
  - Note possibly unusual: 001, rather than 1

• Keyword: “else” or “if” or “begin” or …

• Whitespace: a non-empty sequence of blanks, newlines, and tabs
What are Tokens For?

• Classify program substrings according to role
  - Which is, in effect, the goal of lexical analysis

• Output of lexical analysis is a stream of tokens...
  . . . which is input to the parser

• Parser relies on token distinctions
  - An identifier is treated differently than a keyword
Example

• “foo = 42” tokenized as:

  <Identifier, “foo”>
  < = , “=”>
  <Integer, “42”>

  — Note that “42” is a string. To the lexical analyzer, all things are strings.
Note:

• The lexical analyzer must be able to break down into tokens any string that represents a valid program in the language

• So, somehow we’ll have to specify this:
  - Not only whether the string is a valid program
  - But also what each piece of the string represents (in terms of tokens)
  - AND there can be no ambiguity!
Designing a Lexical Analyzer: Step 1

• Define a finite set of tokens
  - Tokens describe all items of interest
  - Choice of tokens depends on language, design of parser
Example

• Recall
  \tif (i == j)\n  \nt\tz = 0;\n  \ntelse\n  \nt\tz = 1;

• Useful tokens for this expression:
  Integer, Keyword, Relation, Identifier, Whitespace,
  (, ), =, ;

• N.B., (, ), =, ; are tokens, not characters, here
  - And often in token classes all by themselves
Designing a Lexical Analyzer: Step 2

• Describe which strings belong to each token

• Recall:
  - Identifier: *strings of letters or digits, starting with a letter*
  - Integer: *a non-empty string of digits*
  - Keyword: “else” or “if” or “begin” or …
  - Whitespace: *a non-empty sequence of blanks, newlines, and tabs*
Lexical Analyzer: Implementation

• An implementation must do two things:

1. Recognize substrings corresponding to tokens

2. Return the value or *lexeme* of the token
   - The *lexeme* is the substring
Example

- Recall:

```
\tif (i == j) \nt\tz = 0; \ntelse \nt\tz = 1;
```

(W)hitespace  Also (,),=, etc.
(O)perator
(K)eypword
(I)dentifier
(N)umber
Example

• Recall:

\tif (i == j) \n\t tz = 0; \n\t else \n\t tz = 1;

(\textit{W})\textit{hitespace} \quad \textit{Also (,)=, etc.}

(\textit{O})\textit{perator}

(\textit{K})\textit{eyword}

(\textit{I})\textit{dentifier}

(\textit{N})\textit{umber}
Lexical Analyzer: Implementation

• The lexer usually discards “uninteresting” tokens that don’t contribute to parsing.

• Examples: Whitespace, Comments
True Crimes of Lexical Analysis

• Is it as easy as it sounds?

• Not quite!

• There are lots of examples of language design that kind of got this wrong. We don’t have time for that, but here are a couple of examples:
Fortran Example

- **Consider**
  - DO 5 I = 1,25
  - DO 5 I = 1.25

- What is the difference here?
  - One is the first line in a loop
  - The other is a variable declaration
Lexical Analysis in FORTRAN (Cont.)

- Two important points:
  1. The goal is to partition the string. This is implemented by reading left-to-write, recognizing one token at a time
  2. “Lookahead” may be required to decide where one token ends and the next token begins
    1. In this example, need to read to 11th character before knowing what token you have
Lookahead

• As you might expect, lookahead complicates the process of lexical analysis (making for a more complicated compiler)
  - So languages are designed to minimize the need for lookahead

• This being said, some lookahead is almost always required

• For example...
Lookahead

- Even our simple example has lookahead issues

  \[ i \text{ vs. } if \]
  
  \[ = \text{ vs. } == \]

  \[
  \text{if (i == j)} \]
  
  \[
  Z = 0; \]
  
  \[
  \text{else} \]
  
  \[
  Z = 1; \]
More Lookahead

• And yet more:

```plaintext
if (i == j)
    Z = 0;
else
    Z = 1;
```

When we read the “e” in else, we can’t know whether we have a variable name or a keyword until we’ve read through to the space after the second “e”
Lexical Analysis in PL/I

• PL/I stands for Programming Language 1
  - Was supposed to be THE programming language (at least on IBM machines)
  - Supposed to encompass every feature any programmer would ever need
  - And so was supposed to be very, very general and have very few restrictions. So...

• PL/I keywords are not reserved
  - You could have variables named same as keywords
Lexical Analysis in PL/I

• PL/I keywords are not reserved
  IF ELSE THEN THEN = ELSE; ELSE ELSE = THEN

• So you can’t know whether you have a keyword or a variable name until you’ve seen the entire line of code

• Which, as you might expect, makes lexical analysis in PL/I quite challenging
Experience

• Fortran and PL/I taught folks a lot about what to do (and not do) in language design to help make lexical analysis easier.
• But unfortunately modern languages have similar problems (see C++)
Review

• The goal of lexical analysis is to
  – Partition the input string into lexemes
  – Identify the token of each lexeme

• Left-to-right scan => lookahead sometimes required
Next

- We still need
  - A way to describe the lexemes of each token
  - A way to resolve ambiguities
    - Is if two variables i and f?
    - Is == two equal signs = =?
Regular Languages

• There are several formalisms for specifying tokens

• Regular languages are the most popular
  - Simple and useful theory
  - Easy to understand
  - Efficient implementations
Formal Languages

**Def.** Let $\Sigma$ be a set of characters. A *language over* $\Sigma$ is a set of strings of characters drawn from $\Sigma$.

(Note: not every string consisting of characters from $\Sigma$ need be in the language)
Examples of Languages

- Alphabet = English characters
- Language = English sentences
- Not every string of English characters is an English sentence
  - And defining which strings of characters are valid English sentences would be tricky

- Alphabet = ASCII
- Language = C programs
- Note: ASCII character set is different from English character set
Notation

• Languages are sets of strings.

• Need some notation for specifying which sets we want

• The standard method for expressing regular languages is regular expressions.
  - But it is not the only way this can be done.
Atomic Regular Expressions

- Single character: represents the language consisting of one string
  \[ 'c' = \{ "c" \} \]

- Epsilon: also represents a language consisting of one string (does NOT represent the "empty language")
  \[ \varepsilon = \{ "\"\"\" \} \]
Compound Regular Expressions

- **Union**
  \[ A + B = \{ s | s \in A \text{ or } s \in B \} \]

- **Concatenation**
  \[ AB = \{ ab | a \in A \text{ and } b \in B \} \]

- **Iteration**
  \[ A^* = \bigcup_{i \geq 0} A^i \text{ where } A^i = A \ldots i \text{ times} \ldots A \]

*Kleene closure of A*
Compound Regular Expressions

- **Union**
  
  \[ A + B = \{ s \mid s \in A \text{ or } s \in B \} \]

- **Concatenation**

  \[ AB = \{ ab \mid a \in A \text{ and } b \in B \} \]

- **Iteration**

  \[ A^* = \bigcup_{i \geq 0} A^i \text{ where } A^i = A \ldots i \text{ times } \ldots A \]

  Note \( A^0 \) is \( \epsilon \)

Note these are all mappings from an expression (piece of syntax) to a set of strings.
Regular Expressions

• **Def.** The *regular expressions over* $\Sigma$ *are the smallest set of expressions including*

  $\epsilon$  
  'c' where $c \in \Sigma$  
  $A + B$ where $A, B$ are rexp over $\Sigma$  
  $AB$  
  $A^*$ where $A$ is a rexp over $\Sigma$
Examples

• Assume $\Sigma = \{0, 1\}$

• $1^*$

• $(1 + 0)1$

• $0^* + 1^*$

• $(0 + 1)^*$
Note These are Not Unique

• Assume $\Sigma = \{0,1\}$

• $1^*$ same as $1^* + 1$

• $(1 + 0)1$ same as $11 + 01$

• $0^* + 1^*$

• $(0 + 1)^*$ (a.k.a. $\Sigma^*$)
Segue

• Regular expressions are simple, almost trivial
  - But they are useful!

• Reconsider informal token descriptions . . .

• And let’s see how to use regular expressions to specify different aspects of programming languages
Example: Keyword

Keyword: “else” or “if” or “begin” or ...

‘else’ + ‘if’ + ‘begin’ + ... 

Note: ‘else’ abbreviates ‘e’ ‘l’ ‘s’ ‘e’

(which is technically how you express the concatenation of these four single character regular expressions)
Example: Integers

**Integer: a non-empty string of digits**

digit = '0'+'1'+'2'+'3'+'4'+'5'+'6'+'7'+'8'+'9'

integer = digit digit*

Why not digit*? 

**Abbreviation:** $A^+ = AA^*$

Note: most tools allow for the naming of a regular expression (as we did with “digit” above)
Example: Identifier

Identifier: *strings of letters or digits, starting with a letter*

\[
\text{letter} = 'A' + \ldots + 'Z' + 'a' + \ldots + 'z' \\
\text{identifier} = \text{letter} (\text{letter} + \text{digit})^* \\
\]

Is \((\text{letter}^* + \text{digit}^*)\) the same?
Example: Identifier

Identifier: *strings of letters or digits, starting with a letter*

*character range*, supported by most tools

\[
\text{letter} = \text{‘A’} + \ldots + \text{‘Z’} + \text{‘a’} + \ldots + \text{‘z’} = [A-Z] + [a-z]
\]

\[
\text{identifier} = \text{letter (letter + digit)*} = [A-Za-z]
\]

\[
= [a-zA-Z]
\]

Is \((\text{letter* + digit*})\) the same?
Example: Whitespace

Whitespace: a non-empty sequence of blanks, newlines, and tabs

\((\ '   + \n + \t)^+\)
Example: Whitespace

Whitespace: a non-empty sequence of blanks, newlines, and tabs

\((\ '\ ' + \ 'n' + \ 't')^+\)

Note: we sometimes need a way of naming some characters that don’t have a very nice print representation

Typical way: some sort of escape sequences
Let’s look at some non-programming language examples
Example: Phone Numbers

- Regular expressions are all around you!
- Consider (555)-867-5309

\[
\begin{align*}
\Sigma & = \text{digits} \cup \{-,(),\} \\
\text{exchange} & = \text{digit}^3 \\
\text{phone} & = \text{digit}^4 \\
\text{area} & = \text{digit}^3 \\
\text{phone}\_\text{number} & = '(' \text{area }')-': \text{exchange }'-' \text{ phone}
\end{align*}
\]
Example: Email Addresses

• Consider anyone@cs.richmond.edu

\[
\sum = \text{letters } \cup \{.,@\}
\]

name = letter^+

address = name ' @' name '.' name '.' name

Of course this assumes that email addresses only consist of letters and name can’t have a period in it (just to keep things simple here)
Summary

• Regular expressions describe many useful languages

• Regular languages are a language specification
  – We still need an implementation

• Next time: Given a string $s$ and a rexp $R$, is $s \in L(R)$?
Implementation of Lexical Analysis

Lecture 4
Outline

• Specifying lexical structure using regular expressions

• Finite automata
  - Deterministic Finite Automata (DFAs)
  - Non-deterministic Finite Automata (NFAs)

• Implementation of regular expressions
  RegExp => NFA => DFA => Tables
Notation

• There is variation in regular expression notation

• Union: \( A | B \) \( \equiv \) \( A + B \)

• Option: \( A + \varepsilon \) \( \equiv \) \( A? \)

• Range: ‘a’+’b’+…+’z’ \( \equiv \) [a-z]

• At least one: \( A^+ \) \( \equiv \) \( AA^* \)

• Excluded range:

  \text{complement of} \ [a-z] \equiv \ [\^a-z]
Regular Expressions in Lexical Specification

• Earlier lecture: a specification for the predicate

\[ s \in L(R) \]

Note: we can do this just by looking at reg. exp. R

• But a yes/no answer is not enough!
  – Because we need to know not just whether the string is a valid program, but also...

• How to partition the input into tokens

• We adapt regular expressions to this goal
  – I.e., there are some small required extensions
1. Write a regexp for the lexemes of each token class
   • Number = digit +
   • Keyword = ‘if’ + ‘else’ + ...
   • Identifier = letter (letter + digit)*
   • OpenPar = ‘(‘
   • ...

• So, we write down a whole list of regular expressions, one for each syntactic category in language
2. Construct \( R \), matching all lexemes for all token classes

\[
R = \text{Keyword} + \text{Identifier} + \text{Number} + \ldots = R_1 + R_2 + \ldots
\]
What follows is the key to how we use the regular expression specification to perform lexical analysis
Regular Expressions => Lexical Spec. (3)

3. Let input be $x_1...x_n$
   For $1 \leq i \leq n$ check whether the prefix
   $x_1...x_i \in L(R)$

4. If success, then we know that
   $x_1...x_i \in L(R_j)$ for some $j$

5. Remove $x_1...x_i$ from input and go to (3)

Continue removing pieces until we have tokenized the entire string
Ambiguities (1)

- There are ambiguities in the algorithm
  - Some things are under specified (and these turn out to be interesting)
- How much input is used? What if
  - $x_1 \ldots x_i \in L(R)$ and also
  - $x_1 \ldots x_k \in L(R)$ (of course $i \neq k$)
  - Ex. We've got $==$
- Rule: Pick longest possible string in $L(R)$
  - The “maximal munch”
    - Yes, this is really what this rule is called
Ambiguities (2)

• Which token is used? What if
  • $x_1...x_i \in L(R_j)$ and also
  • $x_1...x_i \in L(R_k)$

• Ex. Recall our specifications for keywords and identifiers
  • Keyword = ‘if’ + ‘else’ + ...
  • Identifier = letter (letter + digit)*
    - ‘if’ satisfies both
Ambiguities (2)

• Which token is used? What if
  • \(x_1...x_i \in L(R_j)\) and also
  • \(x_1...x_i \in L(R_k)\)

• Note: in most languages, if it’s a keyword, it’s not an identifier
  - But: changing RE for Identifier to explicitly exclude keywords is a real pain (current rule much more natural)
Ambiguities (2)

• Which token is used? What if
  • $x_1 \ldots x_i \in L(R_j)$ and also
  • $x_1 \ldots x_i \in L(R_k)$

• Rule: use rule listed first ($j$ if $j < k$)
  - Treats “if” as a keyword, not an identifier
  - Bottom line: in file defining our lexical specification, put Keywords before the Identifiers
Error Handling

• What if
  No rule matches a prefix of input?
  Note: This comes up quite a bit

• Solution:
  - Don’t let it ever happen that a string isn’t in $L(R)$
  - Write a rule matching all “bad” strings
    • Create an Error token class
  - Put it last (lowest priority)
    • Putting it last also allows us to be a little bit sloppy - can include strings in the RE that ARE valid
    • Earlier rules will have caught these
    • Action for this rule is to print an error string
Summary

• Regular expressions provide a concise notation for string patterns

• Use in lexical analysis requires small extensions
  - To resolve ambiguities
  - To handle errors

• Good algorithms known
  - Require only single pass over the input
  - Few operations per character (table lookup)
  - These algorithms are the subject of the following slides
Finite Automata

• Regular expressions = specification
• Finite automata = implementation

Closely related: they can specify exactly the same languages -- the regular languages
Finite Automata

• Regular expressions = specification
• Finite automata = implementation

Closely related: they can specify exactly the same languages -- the regular languages
Finite Automata

- Regular expressions = specification
- Finite automata = implementation

- A finite automaton consists of
  - An input alphabet $\Sigma$ (characters the FA can read)
  - A finite set of states $S$ (thus “finite” automata)
  - A start state $n$
  - A set of accepting states $F \subseteq S$
  - A set of transitions $\text{state } \rightarrow^{\text{input}} \text{state}$
    - I.e., if it’s in a given state, it can read some input and move to another specified state
Finite Automata

• Transition

\[ s_1 \rightarrow^a s_2 \]

• Is read

In state \( s_1 \) on input “a” go to state \( s_2 \)

• If end of input and in accepting state => accept
  - That is, “yes, this string was in the language of this machine”

• Otherwise => reject
So

- Start in start state
- Repeat (until end of input string):
  - Read one character of input string
  - Move to appropriate state
- If after last character read you end up in accepting state, string is in language of the FA
- Else, string not in language of FA
  - E.g., Terminates in state not in F or
  - Machine gets stuck: finds itself in a state and there is no transition of that state on the input (note that it does not read “out of”)
Alternative Notation: Finite Automata State Graphs

- A state
- The start state
- An accepting state
- A transition
A Simple Example

- A finite automaton that accepts only “1”
The Language of a Finite Automata

• Is the set consisting of accepted strings
Another Simple Example

- A finite automaton accepting any number of 1’s followed by a single 0
- Alphabet: \{0,1\}
And Another Example

- Alphabet \( \{0,1\} \)
- What language does this recognize?
Epsilon Moves

- Another kind of transition: $\varepsilon$-moves

- Machine can move from state \textbf{A} to state \textbf{B} without reading input
Deterministic and Nondeterministic Automata

• Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No $\varepsilon$-moves

• Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have $\varepsilon$-moves
Execution of Finite Automata

• A DFA can take only one path through the state graph
  - Completely determined by input

• NFAs can choose
  - Whether to make $\varepsilon$-moves
  - Which of multiple transitions for a single input to take
Acceptance of NFAs

- An NFA can get into multiple states

- Input: 1 0 0 0

Rule: NFA accepts if it can get to a final state
NFA vs. DFA (1)

• NFAs and DFAs recognize the same set of languages (regular languages)
  - Think of NFAs as parallel processors

• DFAs are faster to execute
  - There are no choices to consider
NFA vs. DFA (2)

• For a given language NFA can be simpler than DFA

\[
\begin{array}{c}
\text{NFA} \\
\begin{array}{c}
\hspace{1cm} 1 \\
\hspace{1cm} 0 \\
\hspace{1cm} 0 \\
\hspace{1cm} 0 \\
\hspace{1cm} 0 \\
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\text{DFA} \\
\begin{array}{c}
\hspace{1cm} 1 \\
\hspace{1cm} 0 \\
\hspace{1cm} 0 \\
\hspace{1cm} 0 \\
\hspace{1cm} 0 \\
\hspace{1cm} 0 \\
\end{array}
\end{array}
\]

• DFA can be exponentially larger than NFA (why would you expect that to be the case?)
Regular Expressions to Finite Automata

- High-level sketch

- Regular expressions
  - Lexical Specification
  - NFA
  - DFA
  - Table-driven Implementation of DFA (Lookup tables + code for traversing those tables)
Regular Expressions to NFA (1)

- For each kind of rexp, define an equivalent NFA
  - Notation: NFA for rexp $M$

- For $\varepsilon$

- For input $a$
Regular Expressions to NFA: Atomic REs

- For each kind of rexp, define an equivalent NFA
  - Notation: NFA for rexp \( M \)

- For \( \varepsilon \)

- For input \( a \)

Notion here is that we will be building our overall machine up from smaller machines
Regular Expressions to NFA: Compound REs

- For $AB$

- For $A + B$
Regular Expressions to NFA: Compound REs

- For $AB$
  - Note modification of final state

- For $A + B$
  - And addition of $\varepsilon$ transition
  - Note addition of new start and final states
Regular Expressions to NFA: Compound REs

- For $AB$

- For $A + B$

Remember: With NFA if ANY choice works, string is accepted
Regular Expressions to NFA (3)

- For $A^*$
Regular Expressions to NFA (3)

• For $A^*$
Let’s remember one very important thing in all of this: it is done this way so that the process can be automated!

You might see easier DFAs or NFAs, but a computer needs to have an algorithm to create these, which is why we go through this process.
Example of RegExp $\rightarrow$ NFA conversion

- Consider the regular expression $(1+0)^*1$
- The NFA is

![NFA Diagram]

A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D $\rightarrow$ E $\rightarrow$ F $\rightarrow$ G $\rightarrow$ H $\rightarrow$ I $\rightarrow$ J
**ε-closure**

- An idea that helps with transition from NFA to DFA
- ε-closure of state B is all states that can be reached from B via epsilon moves
  - ε-closure(B) = {B,C,D}; ε-closure(G) = {A,B,C,D,G,H,I}
NFA to DFA. Remark

- An NFA may be in many states at any time

- How many different states?

- If there are $N$ states, the NFA must be in some subset of those $N$ states

- How many subsets are there?
  - $2^N - 1$ = finitely many
NFA to DFA: The Trick

- Simulate the NFA
- Each state of DFA is a non-empty subset of states of the NFA
  - Though not all subsets of states of the NFA will have links to/from them
  - So a large number of possible states (but finite!)
Let's do this!

• We won't follow exact steps: too many possible states to list them all (and most not involved in final DFA)
• Start with start state, then work out which additional states required
NFA -> DFA Example
The Final Step: Implementation of DFA

• First, note that our original diagram was somewhat misleading. Some systems go straight from NFA to implementation
  - More on this in a bit
Implementation

- A DFA can be implemented by a 2D table $T$
  - One dimension is “states”
  - Other dimension is “input symbol”
  - For every transition $S_i \rightarrow^a S_k$ define $T[i,a] = k$

- DFA “execution”
  - If in state $S_i$ and input $a$, read $T[i,a] = k$ and skip to state $S_k$
  - Very efficient
Table Implementation of a DFA

<table>
<thead>
<tr>
<th>State</th>
<th>Input Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>0</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>U</td>
<td>T</td>
</tr>
</tbody>
</table>

input symbol

0

1
The code

```c
char[] input;
int i = 0;
int state = 0;

while ( input[i] != '\0' ) {
    state = transitionTable[ state, input[i] ];
    ++i;
}
```

Note how compact and efficient this is
Table Implementation of a DFA

- Much more compact
- Duplicate rows common in DFAs for LA
Compaction of Table

• Turns out that this technique can make tables much more compact

• Moreover, duplicate rows show up quite a bit in the FAs that arise in lexical analysis

• Considering large number of potential states, this can lead to considerable compaction of the table
  - While blowup in states from NFA to DFA is often not the worst case, it can be substantial
  - 2D table can be quite large

• Disadvantage: pointer dereferences
What if we don’t want a DFA?

• Why? Table that results might be huge, so may want to use NFA directly
Directly from NFA to Table

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>ε</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>{B,H}</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>{C,D}</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td>ε</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td>ε</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Good and Bad of Going Directly from NFA to Table

• **Good**: Table guaranteed to be relatively small
  - limited by size of NFA and size of input alphabet

• **Bad**: Even with compression tricks, inner loop runs much more slowly because dealing with sets of states, rather than states themselves
  - So on each move, we need to keep track of all possible states to which we could go (and associated $\varepsilon$-moves)

• **Bottom line**: Can save a lot of space, but cost a lot in time
Implementation (Cont.)

• NFA -> DFA conversion is at the heart of tools such as flex

• But, DFAs can be huge

• In practice, flex-like tools trade off speed for space in the choice of NFA and DFA representations
  - User chooses via configuration, whether they want to be closer to full DFA or full NFA
VTAB = \x0b
NEWLINE = \n
WHITESPACE = [ \t\f\r\{VTAB\}]
TYPESYM = [A-Z][A-Za-z0-9]*
OBJECTSYM = [a-z][A-Za-z0-9]*
SINGLE = [\+/-=*/<,.>,;::@\}\{
ILLEGAL = [^\n \t\f\{VTAB\},rA-Za-z0-9+/\\-<,.>,;::@\}\{

%state COMMENT, LINE_COMMENT, STRING, STRING_RECOVER

%{

<YYINITIAL>{NEWLINE} { curr_lineno++; }
<YYINITIAL>{WHITESPACE}+ {}

<YYINITIAL>"--" { yybegin(LINE_COMMENT); }
<LINE_COMMENT>.* {}
<LINE_COMMENT>\r {}
<LINE_COMMENT>\n { curr_lineno++; yybegin(YYINITIAL); }

<YYINITIAL>"(*" { yybegin(COMMENT);
 comment_nesting++; }

<YYINITIAL>"*)" { /* error - unmatched comment */
 return new Symbol( TokenConstants.ERROR, "Unmatched *" ); }

<COMMENT>"(*)" { comment_nesting++; }
<COMMENT>"*" { comment_nesting--;
 if ( comment_nesting == 0 ) yybegin(YYINITIAL); }
<COMMENT>"([\#\n]+ {}
<COMMENT>"(" {}
<COMMENT>"=*" { curr_lineno++; }

<YYINITIAL>"=>" { return new Symbol( TokenConstants.DARROW ); }
<YYINITIAL>"<=" { return new Symbol( TokenConstants.LE ); }
<YYINITIAL>"<" { return new Symbol( TokenConstants.ASSIGN ); }
<YYINITIAL>[0-9][0-9]* { return new Symbol( TokenConstants.INT_CONST, 

AbstractTable.inttable.addString(yytext())); }

<YYINITIAL>\[Cc][Aa][Ss][Ee] { return new Symbol( TokenConstants.CASE ); }
<YYINITIAL>\[Cc][Ll][Ll][Aa][Ss][Ss] { return new Symbol( TokenConstants.CLASS ); }
<YYINITIAL>\[Cc][Ee][Ee][Ss][Aa][Cc] { return new Symbol( TokenConstants.ELSE ); }
<YYINITIAL>\[Cc][Ff][Ii][Ff] { return new Symbol( TokenConstants.FALSE ); }
<YYINITIAL>\[Cc][Gg][Ee][Ee][Ss][Aa][Cc] { return new Symbol( TokenConstants.GOTO ); }
<YYINITIAL>\[Cc][Hh][Hh][Ee][Ee][Rr][Ii][Ii][Ii] { return new Symbol( TokenConstants.ENTRY ); }
<YYINITIAL>\[Cc][Hh][Ii][Ss][Ss][Ww][Ww][Dd] { return new Symbol( TokenConstants.ISVOID ); }
<YYINITIAL>\[Cc][Oo][Oo][Pp][Pp] { return new Symbol( TokenConstants.LOOP ); }
<YYINITIAL>\[Cc][Oo][Oo][Pp][Pp] { return new Symbol( TokenConstants.NEW ); }

%}