Implementing Automata

We can also encode the transitions directly into a transition table:

<table>
<thead>
<tr>
<th>state</th>
<th>char1</th>
<th>char2</th>
<th>other</th>
<th>Accepting</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>[3]</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>√</td>
</tr>
</tbody>
</table>

States in brackets don’t consume their inputs. Accepting states are indicated by a √. Empty entries represent error states.

Given the table, we can write an interpreter to perform lexical analysis of any DFA:

```java
state := 1
c := first char
while not ACCEPT[state] do {
    newstate := NEXTSTATE[state, c]
    if ADVANCE[state, c] then
        c := nextChar()
        state := newstate
    }
if ACCEPT[state] then accept;
```
Table-driven C Comments

static String input;
static int current = -1;

static boolean[] ACCEPT =
{false, false, false, false, true};

static boolean[][] ADVANCE = {
  // /* */ other
  {true, true, true},
  {true, true, true},
  {true, true, true},
  {true, true, true},
  {true, true, true}
};
public static boolean interpret () {
    int state = 0;
    int c = nextChar();
    while ((c != END) && (state>=0) && !ACCEPT[state])
    {
        int newstate = NEXTSTATE[state][c];
        if (ADVANCE[state][c])
            c = nextChar();
        state = newstate;
    }
    return (state>=0) && ACCEPT[state];
}

public static void main (String[] args) {
    input = args[0];
    boolean result = interpret();
}

Let's do the same thing again, but this time we will hard-code the interpreter using switch-statements.

nextChar and the constant declarations are the same as for the previous program.

case 1 :
    switch (ch) {
        case STAR: ch=nextChar(); state=2;
            break;
        default : return false;
    }
    break;
case 2 :
    switch (ch) {
        case SLASH: ch=nextChar(); state=2;
            break;
        case STAR : ch=nextChar(); state=3;
            break;
        case OTHER: ch=nextChar(); state=2;
            break;
        default : return false;
    }
    break;
We will describe our tokens using REs, convert these to an NFA, convert this to a DFA, and finally code this into a program or a table to be interpreted:

- Each piece of a regular expression is turned into a part of an NFA.
- Each part is glued together (using ǫ-transitions) into a complete automaton.
- An RE matching the character a translates into
  ![a transition diagram](image)
- An RE matching ǫ translates into
  ![epsilon transition diagram](image)

We will next show how to construct an NFA from a regular expression. This algorithm is called Thompson’s Construction (after Ken Thompson of Bell Labs).
Thompson’s Construction – Concatenation

- We represent an RE component $r$ by the figure:

- An RE matching the regular expression $r$ followed by the regular expression $s$ ($rs$) translates into

---

Thompson’s Construction – Alternation

- The regular expression $r|s$ translates into

---

Thompson’s Construction – Repetition

- The regular expression $r^*$ translates into

---

Thompson’s Construction – Example I

- The regular expression $ab|a$ translates into

The regular expression `letter(letter|digit)*` translates into

```
letter
digit
epsilon
epsilon
epsilon
epsilon
epsilon
epsilon
```

From NFA to DFA

We now know how to translate a regular expression into an NFA, and how to translate a DFA into code. The missing piece is how to translate an NFA into a DFA.

Each state in the DFA corresponds to a set of states in the NFA.

The DFA will be in state $\{2, 3, 4\}$ if the NFA could have been in any of the states $\{2, 3, 4\}$.

After reading $a_1a_2\cdots a_n$ the DFA is in a state that represents the states the NFA could be in after seeing the input $a_1a_2\cdots a_n$. 
We need three functions:

1. **ε-closure** is the set of NFA states reachable from some NFA state \( s \) in \( T \) on \( ε \)-transitions alone. This is essentially a graph exploration algorithm that finds the nodes in a graph reachable from a given node.
2. **move** is the set of NFA states to which there is a transition on input symbol \( a \) from some NFA state \( s \in T \).
3. **SubsetConstruction** returns a DFA \( D = (D_{states}, D_{trans}) \) corresponding to NFA \( N \).

### ɛ-closure(T)

**procedure ɛ-closure(T)**
- push all states in \( T \) onto stack
- \( C := T \)
- while stack is not empty do
  - \( t := \text{pop(stack)} \)
  - for each edge \( t \xrightarrow{ε} u \) do
    - if \( u \) is not in \( C \) then
      - \( C := C \cup u \)
      - push(stack, \( u \))
  - return \( C \)

**ε-closure(T) – Example**

- **ε-closure(①)** = \{①, ②, ④\}
- **ε-closure(②)** = \{②\}
- **ε-closure(④)** = \{②, ④\}
- **ε-closure(③, ④)** = \{②, ③, ④\}
move\( (T,a) \) – Example

\[
\begin{align*}
\text{move}(\{1\}, a) &= \{2, 3\} \\
\text{move}(\{2, 3\}, b) &= \{4\}
\end{align*}
\]

SubsetConstruction\((N)\)

procedure SubsetConstruction\((NFA N)\)

\[
\begin{align*}
\text{Dstates} &:= \{\epsilon\text{-closure}(s_0)\} \\
\text{Dtrans} &:= \{} \\
\text{repeat} \\
T &:= \text{an unexplored state in Dstates} \\
\text{for each input symbol } a \text{ do} \\
U &:= \epsilon\text{-closure} (\text{move}(T, a)) \\
\text{if } U \text{ is not in Dstates then} \\
\text{Dstates} &:= \text{Dstates} \cup U \\
\text{Dtrans} &:= \text{Dtrans} \cup (T \xrightarrow{a} U) \\
\text{until all states have been explored} \\
\text{return } (\text{Dstates}, \text{Dtrans})
\end{align*}
\]

NFAT\(\Rightarrow\)DFA

\(\epsilon\text{-closure}(1) = \{1, 2, 4\} = A\)

\(A\) will be the DFA's start state.
Example...

\[ \varepsilon - \text{closure}(\text{move}(A, a)) = \varepsilon - \text{closure}(\text{move}(\{1, 2, 4\}, a)) = \varepsilon - \text{closure}(\{2, 3\}) = \{2, 3, 4\} = B \]

- We add the transition \( A \xrightarrow{a} B \)

Example...

\[ \varepsilon - \text{closure}(\text{move}(A, b)) = \varepsilon - \text{closure}(\text{move}(\{1, 2, 4\}, b)) = \varepsilon - \text{closure}(\{4\}) = \{2, 4\} = C \]

- We add the transition \( A \xrightarrow{b} C \)

Example...

\[ \varepsilon - \text{closure}(\text{move}(B, b)) = \varepsilon - \text{closure}(\text{move}(\{2, 3, 4\}, b)) = \varepsilon - \text{closure}(\{2\}) = \{2, 4\} = C \]

- We add the transition \( B \xrightarrow{b} C \)

Example...

\[ \varepsilon - \text{closure}(\text{move}(C, b)) = \varepsilon - \text{closure}(\text{move}(\{2, 4\}, b)) = \varepsilon - \text{closure}(\{2, 4\}) = \{2, 4\} = C \]

- We add the transition \( C \xrightarrow{b} C \)
A slightly different approach is to generate the power-set of the set of NFA states, and then add all the edges we get from $\epsilon$-closure().

From states 1, 2, 4 we can go to states 2, 3, 4 on an a.

From states 1, 2, 4 we can go to states 2, 4 on a b.
From states \(2, 3, 4\) we can go to states \(2, 4\) on \(a b\).

Finally, removing unreachable states gives us our DFA.

Keywords
Keywords revisited

For a language with many keywords (Ada-95 has 98, COBOL has hundreds), the transition table can be large. We can remove all keywords from the transition table and instead analyze them as IDENTs. When an IDENT is found we look it up in a special table to see if it is, in fact, a reserved word.

We can use a regular hash-table, of course, but if we're concerned about speed we can use a minimal perfect hash-table. This is a static table and related lookup routines that have been optimized for a particular static set of words.

For example, we could build this perfect hash-table for the words LUCA, MODULA-2, OBERON:

<table>
<thead>
<tr>
<th>Numerical Value</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LUCA</td>
</tr>
<tr>
<td>1</td>
<td>MODULA-2</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>OBERON</td>
</tr>
</tbody>
</table>

int hash(String s) {return s[0]-'L';}

boolean member(String s) {return table[hash(s)] = s;}

In this case we use the first character of the string as the hash-value.

This is not a minimal table, there's one wasted entry.

Using Unix gperf

- gperf (http://www.gnu.org/manual/gperf-2.7) is a Unix program that takes a list of keywords as input and returns a perfect hash-table (and related search routines) as output.
- From the gperf manual:

  The perfect hash function generator gperf reads a set of "keywords" from a keyfile. It attempts to derive a perfect hashing function that recognizes a member of the static keyword set with at most a single probe into the lookup table. If gperf succeeds in generating such a function it produces a pair of C source code routines that perform hashing and table lookup recognition.

- The following command

  > echo "BEGIN\n  END" | gperf -L ANSI-C

  generates the C program below.

  /* ANSI-C code produced by gperf version 2.7 */

  #define TOTAL_KEYWORDS 2
  #define MIN WORD LENGTH 3
  #define MAX WORD LENGTH 5
  #define MIN HASH VALUE 3
  #define MAX HASH VALUE 5
Using Unix gperf...

static unsigned int hash (register const char *str, register unsigned int len) {
    static unsigned char asso_values[] = {
        6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
        6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
        6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
        6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
        6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
        6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
        6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
    };
    return len + asso_values[(unsigned char)str[len - 1]] +
    asso_values[(unsigned char)str[0]];
}

const char * in_word_set (register const char *str,
    register unsigned int len) {
    static const char * wordlist[] = {
        "", "", "", "END", ",", "BEGIN";
    }
    if (len<=MAX_WORD_LENGTH && len>=MIN_WORD_LENGTH) {
        register int key = hash (str, len);
        if (key <= MAX_HASH_VALUE && key >= 0) {
            register const char *s = wordlist[key];
            if (*str == *s && !strcmp (str + 1, s + 1)) return
        }
    return 0;
}

- In this particular case, the hash function only looks at the first
  and last characters of the string, as well as the string length.

Summary

- The problem with table-driven methods is that the tables can
  easily get huge. Much work has gone into constructing
  table-compression algorithms, and data structures for sparse
  tables. See the Dragon book for details.
- There are also many algorithms for minimizing the number of
  states in a DFA. See Louden, pp. 72–74.
Read Louden, pp. 31–80.

Or, read the Dragon book, pp. 83–140.

An interview with Ken Thompson:

His Turing award lecture (*Reflections on Trusting Trust*):
http://www.acm.org/classics/sep95/.

The next slide shows how you insert a Trojan Horse in the C compiler.