CSc 453 — Compilers and Systems Software

4 : Lexical Analysis II

Christian Collberg
Department of Computer Science
University of Arizona
collberg@gmail.com

Copyright © 2009 Christian Collberg

August 23, 2009

1 Implementing Automata

NFAs and DFAs can be hard-coded using this pattern:

```
state := start state
c := first char
while (true) {
   case state of {
      1:    case c of {
               char1 :  {
                  c := nextChar();
                  state := new state; }
      2:    case c of {
               char2 :  {
                  c := nextChar();
                  state := new state; }
               char3 :  {
                  return; /* accept */}}}
```

2 Implementing Automata...

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

	next state			
state	$\texttt{char}_1$	$\texttt{char}_2$	other	**Accepting**
1	2			
2	2	2	[3]	
3				$\surd$

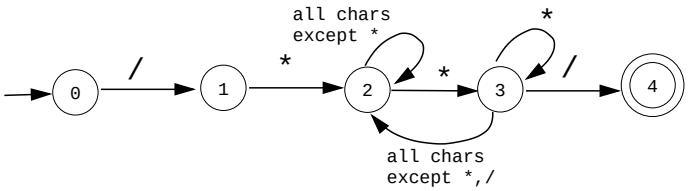
- States in brackets don't consume their inputs. Accepting states are indicated by a $\surd$. Empty entries represent error states.

3 Implementing Automata...

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

```
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;
```

4 Table-driven C Comments



state	/	*	other	Accepting
0	1			
1		2		
2	2	3	2	
3	4	3	2	
4				√

5 Table-driven C Comments...

```
class Comments {
    public static final int SLASH = 0;
    public static final int STAR  = 1;
    public static final int OTHER = 2;
    public static final int END   = 3;

    static int[][] NEXTSTATE = {
    //   "/"   "*"     other
        { 1,     -1,     -1},
        {-1,      2,     -1},
        { 2,      3,      2},
        { 4,      3,      2},
        {-1,     -1,     -1}
    };
```

6 Table-driven C Comments...

```
    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}
    };
```

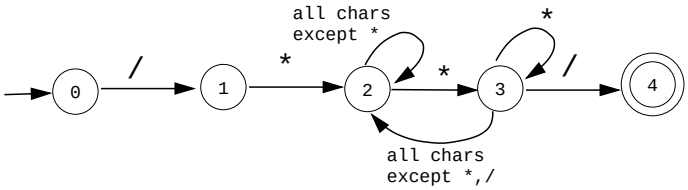
7 Table-driven C Comments...

```
    static String input;
    static int current = -1;

    static int nextChar() {
        int ch;
        current++;
        if (current >= input.length()) return END;
        switch (input.charAt(current)) {
           case '/' : { ch = SLASH; break;}
           case '*' : { ch = STAR; break;}
           default  : { ch = OTHER; break;}
        }
        return ch;
    }
```

8 Table-driven C Comments...

```
    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();
    }
}
```

9 Hard-coded C Comments



- 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.

10 Hard-coded C Comments...

```
class Comments {
// Declarations of SLASH,STAR,OTHER,END, and nextChar().
   public static boolean interpret() {
      int state = 0;
      int ch = nextChar();
      while(true) {
         switch (state) {
             case -1 :
                 return false;
             case 0  :
                switch (ch) {
                   case SLASH:ch=nextChar();state=1;break;
                   default   :return false;
                }
                break;
```

11

```
             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;
```

12 Hard-coded C Comments...

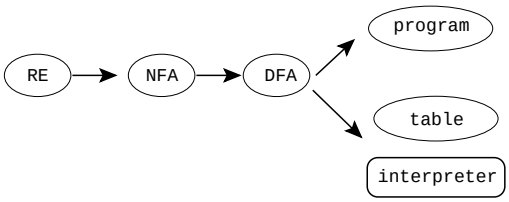
```
            case 3 :
               switch (ch) {
                  case SLASH: ch=nextChar(); state=4;
                              break;
                  case STAR : ch=nextChar(); state=3;
                              break;
                  case OTHER: ch=nextChar(); state=2;
                              break;
                  default   : return false;
               }
               break;
            case 4 :
               return (ch == END);
         }
      }
   }
}
```

13

From REs to NFAs

14 From REs to NFAs

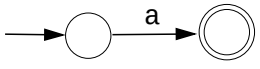
- We will describe our tokens using REs, convert these to an NFA, convert this to a DFA, and finally code this into a program or a table to be interpreted:



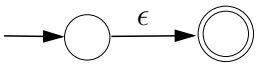
- We will next show how to construct an NFA from a regular expression. This algorithm is called Thompson's Construction (after Ken Thompson of Bell Labs).

15 Thompson's Construction

- Each piece of a regular expression is turned into a part of an NFA.
- Each part is glued together (using ϵ-transitions) into a complete automaton.
- An RE matching the character `a` translates into

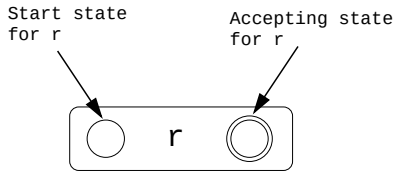


- An RE matching ϵ translates into

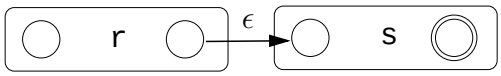


16 Thompson's Construction – Concatenation

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

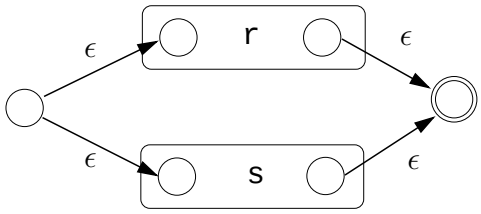


- An RE matching the regular expression `r` followed by the regular expression `s` (`rs`) translates into



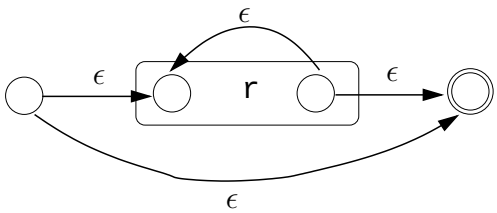
17 Thompson's Construction – Alternation

- The regular expression `r|s` translates into



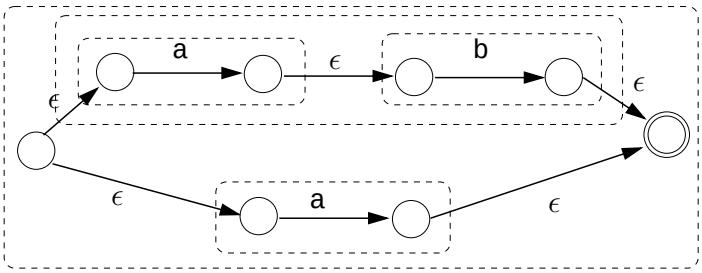
18 Thompson's Construction – Repetition

- The regular expression `r*` translates into



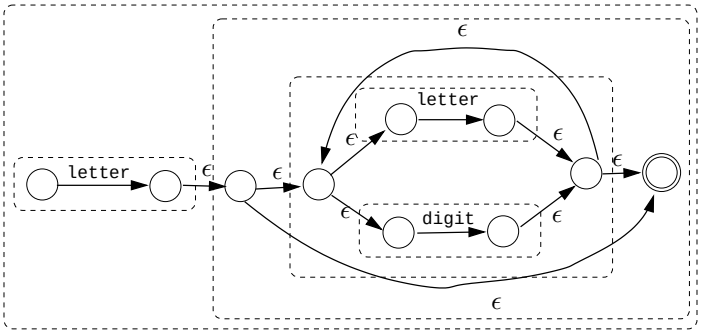
19 Thompson's Construction – Example I

- The regular expression `ab|a` translates into



20 Thompson's Construction – Example II

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

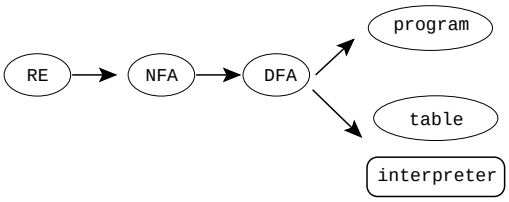


21

From NFA to DFA

22 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.



23 From NFA to DFA...

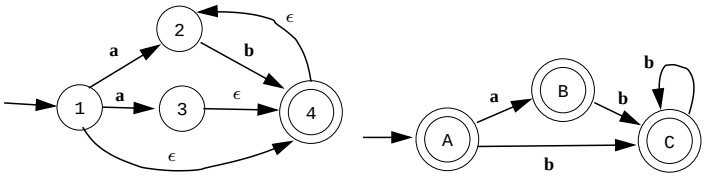
- Each state in the DFA corresponds to a set of states in the NFA.
- The DFA will be in state

  if the NFA could have been in any of the states

  ②, ③, ④.

- After reading $a_1 a_2 \cdots a_n$ the DFA is in a a state that represents the states the NFA could be in after seeing the input $a_1 a_2 \cdots a_n$.

24 From NFA to DFA...



- Ⓐ in the DFA represents the set of states {①, ②, ④} in the NFA. These are the states the FAs could be in before any input is consumed (the start states).
- Ⓑ in the DFA represents the set of states {②, ③, ④} in the NFA. These are the states we can get to on the symbol `a` from Ⓐ.

25 From NFA to DFA...

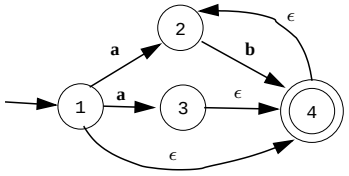
We need three functions:

1. ϵ-closure(T) 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 griven node.
2. move(T,a) 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(N) returns a DFA D=(Dstates,Dtrans) corresponding to NFA N.

26 ϵ-closure(T)

```
procedure ϵ-closure(T)
    push all states in T onto stack
    C := T
    while stack is not empty do
        t := pop(stack)
        for each edge t --ϵ--> u do
            if u is not in C then
                C := C ∪ u
                push(stack, u)
    return C
```

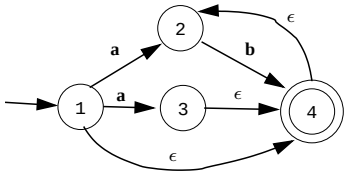
27 ϵ-closure(T) – Example



- ϵ-closure(①) = {①, ②, ④}
- ϵ-closure(②) = {②}
- ϵ-closure(④) = {②, ④}

- ϵ-closure($\{③, ④\}$) = $\{②, ③, ④\}$

28 move(T,a) – Example

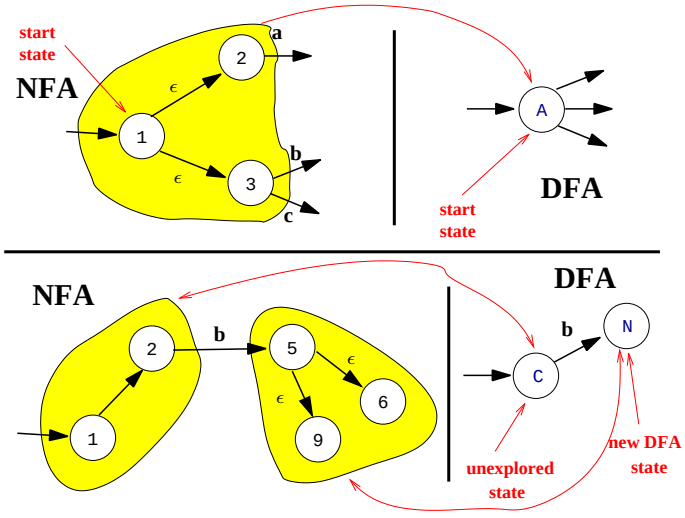


- move($\{①\}, a$) = $\{②, ③\}$
- move($\{②, ③\}, b$) = $\{④\}$

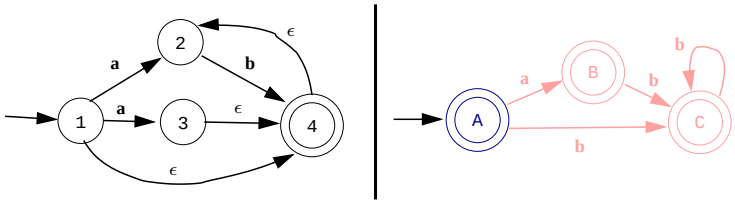
29 SubsetConstruction(N)

```
procedure SubsetConstruction(NFA N)
   Dstates := {ϵ-closure(s0)}
   Dtrans := {}
   repeat
      T := an unexplored state in Dstates
      for each input symbol a do
         U := ϵ-closure(move(T,a))
         if U is not in Dstates then
            Dstates := Dstates ∪ U
         Dtrans := Dtrans ∪ (T -a-> U)
   until all states have been explored
   return (Dstates,Dtrans)
```

30 NFA⇒DFA

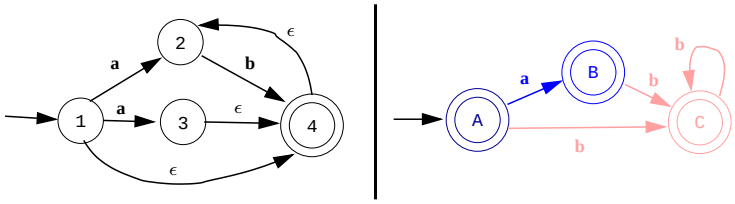


31 `SubsetConstruction(N)` – Example



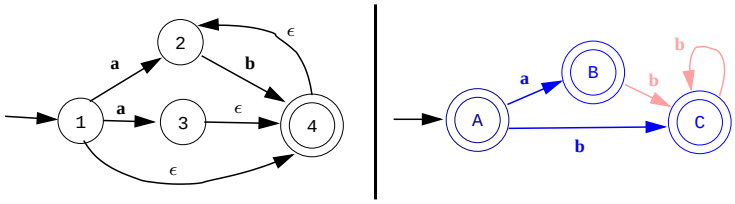
1. ϵ-`closure`(①) = {①, ②, ④} = Ⓐ
   - Ⓐ will be the DFA's start state.

32 Example. . .



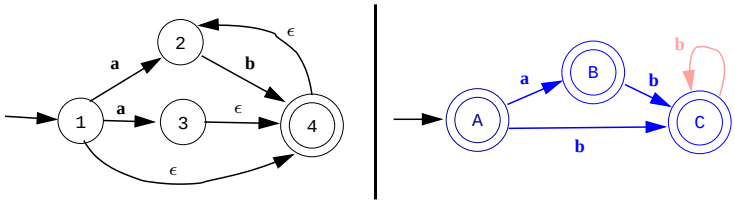
2. ϵ-`closure`(`move`(Ⓐ, a)) = ϵ-`closure`(`move`({①, ②, ④}, a)) = ϵ-`closure`({②, ③}) = {②, ③, ④} = Ⓑ
   - We add the transition Ⓐ $\xrightarrow{a}$ Ⓑ

33 Example. . .



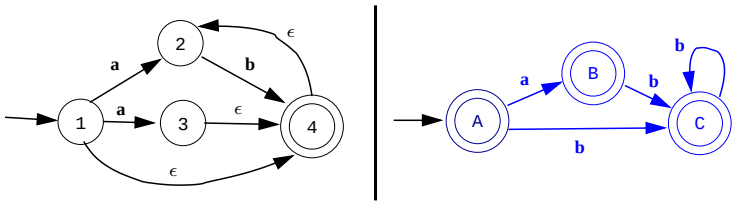
3. ϵ-`closure`(`move`(Ⓐ, b)) = ϵ-`closure`(`move`({①, ②, ④}, b)) = ϵ-`closure`({④}) = {②, ④} = Ⓒ
   - We add the transition Ⓐ $\xrightarrow{b}$ Ⓒ

34 Example. . .



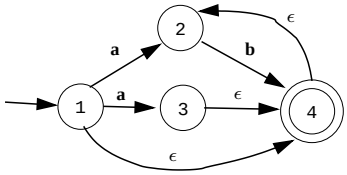
4. ϵ-`closure`(`move`(Ⓑ, b)) = ϵ-`closure`(`move`({②, ③, ④}, b)) = ϵ-`closure`({④}) = {②, ④} = Ⓒ
   - We add the transition Ⓑ $\xrightarrow{b}$ Ⓒ

35 Example...

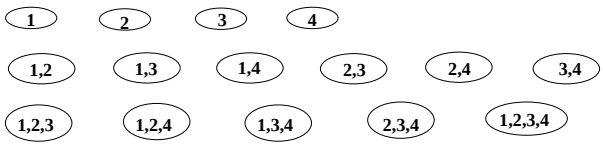


5. ϵ-closure(move(Ⓒ, b)) = ϵ-closure(move({②, ④}, b)) = ϵ-closure({②, ④}) = {②, ④} = Ⓒ
   - We add the transition Ⓒ $\xrightarrow{b}$ Ⓒ

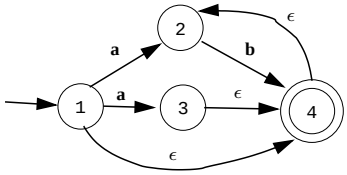
36 Example, Take 2



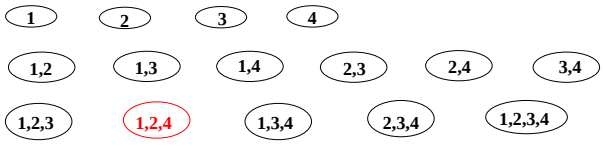
- A slightly different approach is to generate the power-set of the set of NFA states, and then add all the edges we get from ϵ-closure().



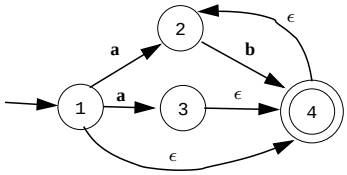
37 Example, Take 2...



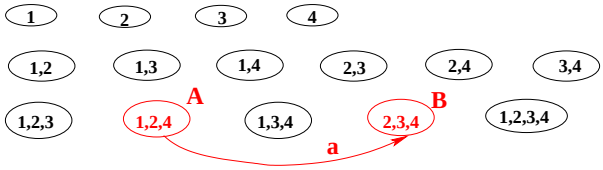
- On ϵ we can go to states ①, ②, ④ which becomes our start state, Ⓐ.



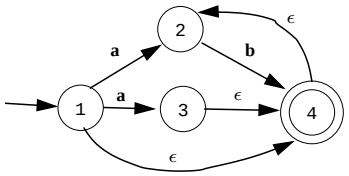
38 Example, Take 2...



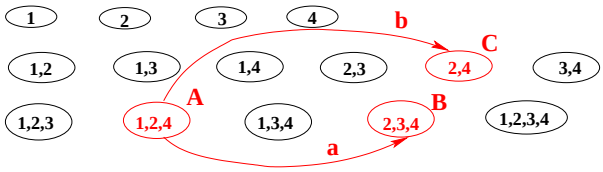
- From states ①, ②, ④ we can go to states ②, ③, ④ on an **a**.



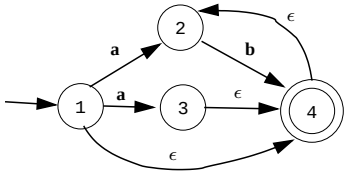
39 Example, Take 2...



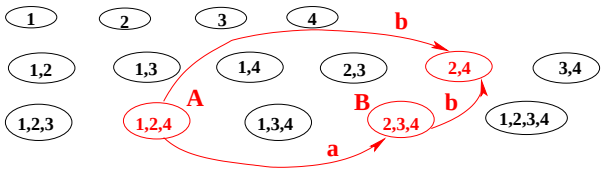
- From states ①, ②, ④ we can go to states ②, ④ on a **b**.



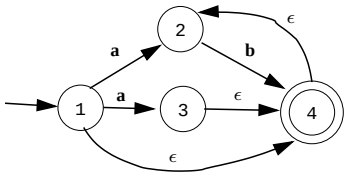
40 Example, Take 2...



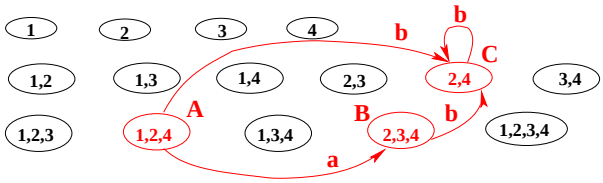
- From states ②, ③, ④ we can go to states ②, ④ on a **b**.



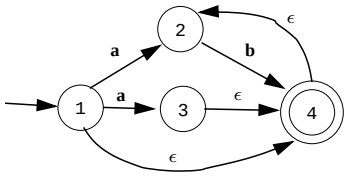
41 Example, Take 2...



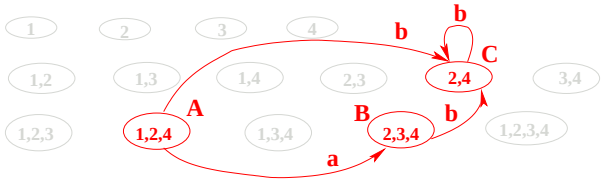
- From states ②, ④ we can go to states ②, ④ on a **b**.



42 Example, Take 2...



- Finally, removing unreachable states gives us our DFA.



43

Keywords

44 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 `IDENT`s.
- 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.

45 Keywords revisited...

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

0	`LUCA`
1	`MODULA-2`
2	
3	`OBERON`

```
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.

46 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.
  ```

47 Using Unix gperf...

- The following command

  ```
  > echo "BEGIN\nEND" | 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
```

48 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, 0, 6, 0, 0,
    <--- Lots more stuff like this --->
    };
  return len + asso_values[(unsigned char)str[len - 1]] +
         asso_values[(unsigned char)str[0]];
}
```

49

```
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 s;
      }
  }
  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.

50

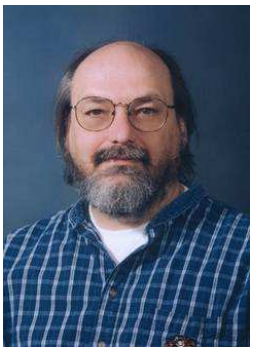
Summary

51 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.

52 Readings and References

- Read Louden, pp. 31–80.
- Or, read the Dragon book, pp. 83–140.
- An interview with Ken Thompson: `http://www.computer.org/computer/thompson.htm`.
- 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.



53 Reflections on Trusting Trust

```
compile (String S)
   if (we're compiling "login.c")
      GENERATE_CODE(
         if (user=="collberg" && passwd="D. Troi")
            login_ok = true
      )
   if (we're compiling "gcc.c")
      GENERATE_CODE(
         if (we're compiling "login.c")
            GENERATE_CODE(
               if (user=="collberg" && passwd="D. Troi")
                  login_ok = true
            )
      )
```