

# Lexical Analysis Wrapup Comp 412

Notes on the Lab 1 Report  
(due Friday) are posted on  
the class web site.

Copyright 2009, Keith D. Cooper & Linda Torczon, all rights reserved.  
Students enrolled in Comp 412 at Rice University have explicit permission to make copies  
of these materials for their personal use.  
Faculty from other educational institutions may use these materials for nonprofit  
educational purposes, provided this copyright notice is preserved.

## Table-Driven Scanners

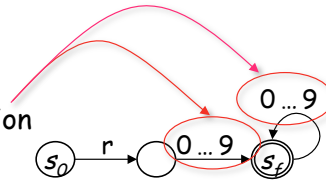


Common strategy is to simulate DFA execution

- Table + Skeleton Scanner
  - So far, we have used a simplified skeleton

```
state ← s0;  
while (state ≠ exit) do  
  char ← NextChar()           // read next character  
  state ← δ(state, char);     // take the transition
```

- In practice, the skeleton is more complex
  - Character classification for table compression
  - Building the lexeme
  - Recognizing subexpressions
    - Practice is to combine all the REs into one DFA
    - Must recognize individual words without hitting EOF





## Table-Driven Scanners

### Character Classification

- Group together characters by their actions in the DFA
  - Combine identical columns in the transition table,  $\delta$
  - Indexing  $\delta$  by class shrinks the table

```

state ← s0;
while (state ≠ exit) do
  char ← NextChar( )           // read next character
  cat ← CharCat(char)         // classify character
  state ← δ(state,cat)        // take the transition

```

- Idea works well in ASCII (or EBCDIC)
  - compact, byte-oriented character sets
  - limited range of values
- Not clear how it extends to larger character sets (unicode)

Obvious algorithm is  $O(|\Sigma|^2 \cdot |S|)$ .  
Can you do better?



## Table-Driven Scanners

### Building the Lexeme

- Scanner produces syntactic category *(part of speech)*
  - Most applications want the lexeme (word), too

```

state ← s0
lexeme ← empty string
while (state ≠ exit) do
  char ← NextChar( )           // read next character
  lexeme ← lexeme + char       // concatenate onto lexeme
  cat ← CharCat(char)         // classify character
  state ← δ(state,cat)        // take the transition

```

- This problem is trivial
  - Save the characters

## Table-Driven Scanners



### Choosing a Category from an Ambiguous RE

- We want one DFA, so we combine all the REs into one
  - Some strings may fit RE for more than 1 syntactic category
    - Keywords versus general identifiers
    - Would like to encode them into the RE & recognize them
  - Scanner must choose a category for ambiguous final states
    - Classic answer: specify priority by order of REs (*return 1<sup>st</sup>*)

### Alternate Implementation Strategy (Quite popular)

- Build hash table of keywords & fold keywords into *identifiers*
- Preload keywords into hash table
- Makes sense if
  - Scanner will enter all *identifiers* in the table
  - Scanner is hand coded
- Otherwise, let the DFA handle them (*O(1) cost per character*)

Separate keyword table can make matters worse

## Table-Driven Scanners



### Scanning a Stream of Words

- Real scanners do not look for 1 word per input stream
  - Want scanner to find all the words in the input stream, in order
  - Want scanner to return one word at a time
  - Syntactic Solution: can insist on delimiters
    - Blank, tab, punctuation, ...
    - Do you want to force blanks everywhere? in expressions?
  - Implementation solution
    - Run DFA to error or EOF, back up to accepting state
- Need the scanner to return *token*, not boolean
  - Token is *<Part of Speech, lexeme>* pair
  - Use a map from DFA's state to Part of Speech (*PoS*)



# Table-Driven Scanners

## Handling a Stream of Words

```
// recognize words
state ← s0
lexeme ← empty string
clear stack
push (bad)
while (state ≠ se) do
  char ← NextChar( )
  lexeme ← lexeme + char
  if state ∈ SA
    then clear stack
  push (state)
  cat ← CharCat(char)
  state ← δ(state,cat)
end;
```

```
// clean up final state
while (state ∉ SA and state ≠ bad) do
  state ← pop()
  truncate lexeme
  roll back the input one character
end;

// report the results
if (state ∈ SA)
  then return ⟨PoS(state), lexeme⟩
else return invalid
```

PoS: state → part of speech

Need a clever buffering scheme, such as double buffering to support roll back

Comp 412, Fall 2009

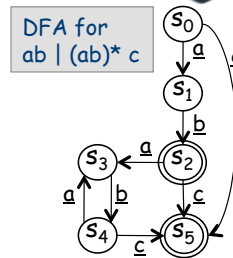
This scanner differs significantly from the ones shown in Chapter 2 of Eac 1e.

## Avoiding Excess Rollback



- Some REs can produce quadratic rollback
  - Consider  $ab / (ab)^* c$  and its DFA
  - Input "ababababc"
    - s<sub>0</sub>, s<sub>1</sub>, s<sub>3</sub>, s<sub>4</sub>, s<sub>3</sub>, s<sub>4</sub>, s<sub>3</sub>, s<sub>4</sub>, s<sub>5</sub>
  - Input "abababab"
    - s<sub>0</sub>, s<sub>1</sub>, s<sub>3</sub>, s<sub>4</sub>, s<sub>3</sub>, s<sub>4</sub>, s<sub>3</sub>, s<sub>4</sub>, rollback 6 characters
    - s<sub>0</sub>, s<sub>1</sub>, s<sub>3</sub>, s<sub>4</sub>, s<sub>3</sub>, s<sub>4</sub>, rollback 4 characters
    - s<sub>0</sub>, s<sub>1</sub>, s<sub>3</sub>, s<sub>4</sub>, rollback 2 characters
    - s<sub>0</sub>, s<sub>1</sub>, s<sub>3</sub>

Not too pretty



Need transition on c

- This behavior is preventable
  - Have the scanner remember paths that fail on particular inputs
  - Simple modification creates the "maximal munch scanner"

Comp 412, Fall 2009



## Maximal Munch Scanner

```

// recognize words
state ← s0
lexeme ← empty string
clear stack
push (bad,bad)
while (state ≠ se) do
  char ← NextChar( )
  InputPos ← InputPos + 1
  lexeme ← lexeme + char
  if Failed[state,InputPos]
    then break;
  if state ∈ SA
    then clear stack
  push (state,InputPos)
  cat ← CharCat(char)
  state ← δ(state,cat)
end

// clean up final state
while (state ∈ SA and state ≠ bad) do
  Failed[state,InputPos] ← true
  ⟨state,InputPos⟩ ← pop()
  truncate lexeme
  roll back the input one character
end

// report the results
if (state ∈ SA)
  then return ⟨Pos(state), lexeme⟩
else return invalid

InitializeScanner()
InputPos ← 0
for each state s in the DFA do
  for i ← 0 to |input| do
    Failed[s,i] ← false
  end;
end;

```

Comp 412, Fall 2009

8



## Maximal Munch Scanner

- Uses a bit array *Failed* to track dead-end paths
  - Initialize both *InputPos* & *Failed* in *InitializeScanner()*
  - *Failed* requires space  $\propto$  |input stream|
    - Can reduce the space requirement with clever implementation
- Avoids quadratic rollback
  - Produces an efficient scanner
  - Can your favorite language cause quadratic rollback?
    - If so, the solution is inexpensive
    - If not, you might encounter the problem in other applications of these technologies

Comp 412, Fall 2009

Thomas Reps, "Maximal munch' tokenization in linear time", ACM TOPLAS, 20(2), March 1998, pp 259-273.

## Table-Driven Versus Direct-Coded Scanners



Table-driven scanners make heavy use of indexing

- Read the next character
- index* • Classify it
- index* • Find the next state
- Branch back to the top

```
state ← s0;  
while (state ≠ exit) do  
  char ← NextChar( )  
  cat ← CharCat(char)  
  state ← δ(state, cat);
```

Alternative strategy: direct coding

- Encode state in the program counter
  - Each state is a separate piece of code
- Do transition tests locally and directly branch
- Generate ugly, spaghetti-like code
- More efficient than table driven strategy
  - Fewer memory operations, might have more branches

Code locality as opposed to random access in  $\delta$

## Table-Driven Versus Direct-Coded Scanners



Overhead of Table Lookup

- Each lookup in CharCat or  $\delta$  involves an address calculation and a memory operation
  - CharCat(char) becomes  $@CharCat_0 + char \times w$  w is sizeof(el't of CharCat)
  - $\delta(state, cat)$  becomes  $@\delta_0 + (state \times cols + cat) \times w$  cols is # of columns in  $\delta$   
w is sizeof(el't of  $\delta$ )
- The references to CharCat and  $\delta$  expand into multiple ops
- Fair amount of overhead work per character
- Avoid the table lookups and the scanner will run faster



## Building Faster Scanners from the DFA

### A direct-coded recognizer for `r Digit Digit`

```

start: accept ← se
      lexeme ← ""
      count ← 0
      goto s0
s0: char ← NextChar
     lexeme ← lexeme + char
     count++
     if (char = 'r')
       then goto s1
       else goto sout
s1: char ← NextChar
     lexeme ← lexeme + char
     count++
     if ('0' ≤ char ≤ '9')
       then goto s2
       else goto sout
s2: char ← NextChar
     lexeme ← lexeme + char
     count ← 0
     accept ← s2
     if ('0' ≤ char ≤ '9')
       then goto s2
       else goto sout
sout: if (accept ≠ se)
      then begin
        for i ← 1 to count
          RollBack()
        report success
      end
      else report failure

```

Fewer (complex) memory operations  
 No character classifier  
 Use multiple strategies for test & branch

Comp 412, Fall 2009



## Building Faster Scanners from the DFA

### A direct-coded recognizer for `r Digit Digit`

```

start: accept ← se
      lexeme ← ""
      count ← 0
      goto s0
s0: char ← NextChar
     lexeme ← lexeme + char
     count++
     if (char = 'r')
       then goto s1
       else goto sout
s1: char ← NextChar
     lexeme ← lexeme + char
     count++
     if ('0' ≤ char ≤ '9')
       then goto s2
       else goto sout
s2: char ← NextChar
     lexeme ← lexeme + char
     count ← 1
     accept ← s2
     if ('0' ≤ char ≤ '9')
       then goto s2
       else goto sout
sout: if (accept ≠ se)
      then begin
        for i ← 1 to count

```

If end of state test is complex (e.g., many cases), scanner generator should consider other schemes

- Table lookup (with classification?)
- Binary search

Comp 412, Fall 2009

Direct coding the maximal munch scanner is easy, too.

13

## What About Hand-Coded Scanners?



Many (most?) modern compilers use hand-coded scanners

- Starting from a DFA simplifies design & understanding
- Avoiding straight-jacket of a tool allows flexibility
  - Computing the value of an integer
    - In LEX or FLEX, many folks use `sscanf()` & touch chars many times
    - Can use old assembly trick and compute value as it appears
  - Combine similar states *(serial or parallel)*
- Scanners are fun to write
  - Compact, comprehensible, easy to debug, ...
  - Don't get too cute *(e.g., perfect hashing for keywords)*

## Building Scanners



The point

- All this technology lets us automate scanner construction
- Implementer writes down the regular expressions
- Scanner generator builds NFA, DFA, minimal DFA, and then writes out the (table-driven or direct-coded) code
- This reliably produces fast, robust scanners

For most modern language features, this works

- You should think twice before introducing a feature that defeats a DFA-based scanner
- The ones we've seen (e.g., insignificant blanks, non-reserved keywords) have not proven particularly useful or long lasting

*Of course, not everything fits into a regular language ...*

## Limits of Regular Languages



Not all languages are regular

$$RL's \subset CFL's \subset CSL's$$

You cannot construct DFA's to recognize these languages

- $L = \{p^kq^k\}$  *(parenthesis languages)*
- $L = \{wcw^r \mid w \in \Sigma^*\}$

Neither of these is a regular language *(nor an RE)*

But, this is a little subtle. You can construct DFA's for

- Strings with alternating 0's and 1's  
 $(\epsilon \mid 1)(01)^*(\epsilon \mid 0)$
  - Strings with an even number of 0's and 1's
- RE's can count bounded sets and bounded differences

## Limits of Regular Languages



Advantages of Regular Expressions

- Simple & powerful notation for specifying patterns
- Automatic construction of fast recognizers
- Many kinds of syntax can be specified with REs

Example — an expression grammar

$$Term \rightarrow [a-zA-Z]([a-zA-Z] \mid [0-9])^*$$

$$Op \rightarrow + \mid - \mid * \mid /$$

$$Expr \rightarrow (Term Op)^* Term$$

Of course, this would generate a DFA ...

If REs are so useful ...

*Why not use them for everything?*

## What can be so hard?



Poor language design can complicate scanning

- Reserved words are important  
if then then then = else; else else = then (PL/I)
- Insignificant blanks (Fortran & Algol68)  
do 10 i = 1,25  
do 10 i = 1.25
- String constants with special characters (C, C++, Java, ...)  
newline, tab, quote, comment delimiters, ...
- Finite closures (Fortran 66 & Basic)
  - Limited identifier length
  - Adds states to count length

## What can be so hard?

(Fortran 66/77)



```
INTEGERFUNCTIONA
PARAMETER(A=6,B=2)
IMPLICIT CHARACTER*(A-B)(A-B)
INTEGER FORMAT(10), IF(10), DO9E1
100 FORMAT(4H)=(3)
200 FORMAT(4 )=(3)
   DO9E1=1
   DO9E1=1,2
   9 IF(X)=1
     IF(X)H=1
     IF(X)300,200
300 CONTINUE
   END
C   THIS IS A "COMMENT CARD"
$ FILE(1)
   END
```

How does a compiler scan this?

- First pass finds & inserts blanks
- Can add extra words or tags to create a scanable language
- Second pass is normal scanner