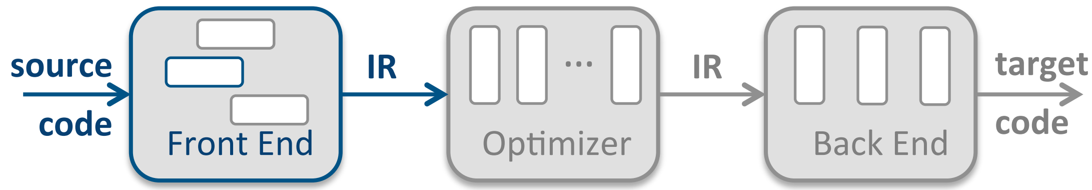
RICE

COMP 412
FALL 2017

Lexical Analysis, V

Implementing Scanners

Comp 412



Section 2.5 in EaC2e

Table-Driven Scanners

A 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
  - Forming a string from the characters
    - → Recording the "lexeme"
  - Recognizing sub-expressions
    - → Practice is to combine all the REs into one DFA
    - → Must recognize individual words without hitting EOF

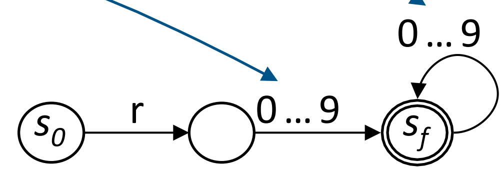


Table-Driven Scanners

Character Classification

- Group together characters by their actions in the **DFA**
  - Combine identical columns in the transition table, δ
  - Indexing δ by a character's class shrinks the table

state $\leftarrow s_0$ *;*
while (state $\neq$ *exit) do*
  char $\leftarrow$ *NextChar( )*   *// read next character*
  cat $\leftarrow$ *CharCat(char)*   *// classify character*
  state $\leftarrow$ 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 $\mathbf{O}(|\Sigma|^2 \bullet |S|)$.
Can you do better?

Table-Driven Scanners

Forming a String with the Lexeme

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

state $\leftarrow$ *s_0*
lexeme $\leftarrow$ *empty string*
while (state $\neq$ *exit) do*
  char $\leftarrow$ *NextChar( )* // *read next character*
  lexeme $\leftarrow$ *lexeme || char* // *concatenate onto lexeme*
  cat $\leftarrow$ *CharCat(char)* // *classify character*
  state $\leftarrow$ *δ(state,cat)* // *take the transition*

Choose syntactic category based on the final state (see Lecture 4, Slide 16)

- 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 **RE**s into one
  - Some strings may fit **RE** for more than 1 syntactic category
    - → Keywords versus general identifiers

      [a-z]([a-z]|[0-9])* vs for|while
    - → Would like to encode all of these distinctions into the **RE** & recognize them in a single **DFA**
  - Scanner must choose a category for ambiguous final states
    - → Classic answer: specify priority by order of **RE**s (*return 1st one*)

[a-z] is a lex notation for the letters 'a' through 'z' in ASCII collating sequence order.

Keywords Folded into the RE

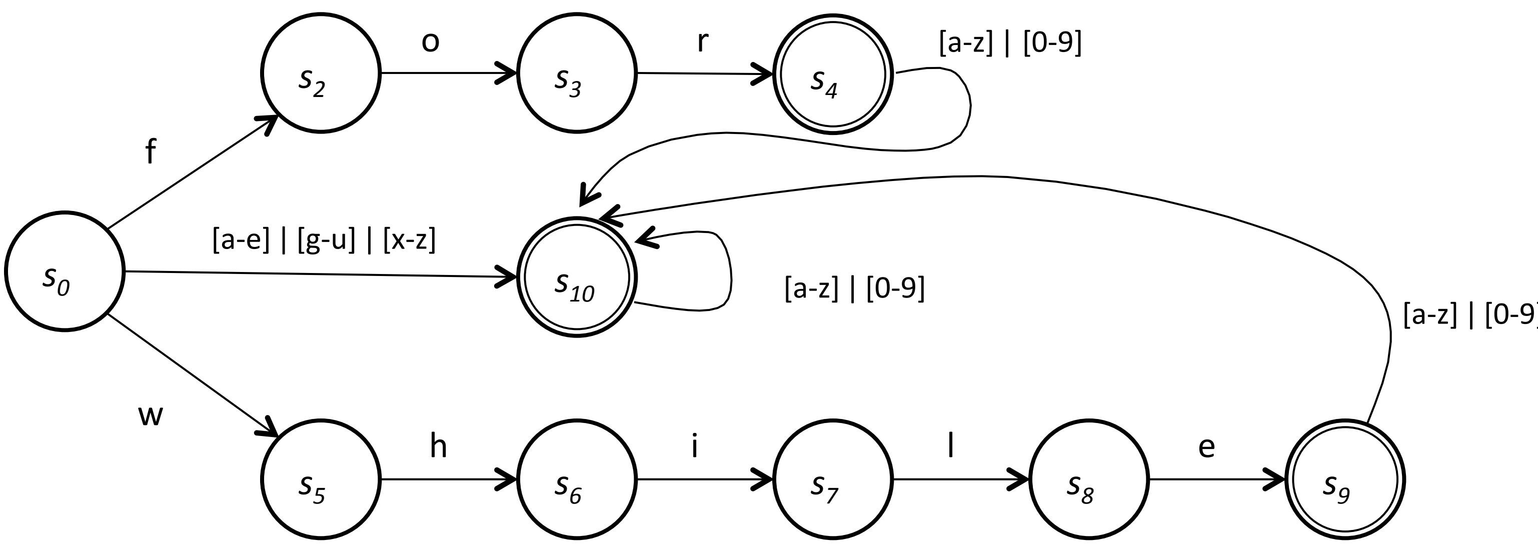
Example

The regular expression

for | while | ([a-z] ([a-z] | [0-9])*)

leads to a **DFA** such as

Need transitions from for and while into s_{10}



s_4 ⇒ "for"
s_9 ⇒ "while"
s_{10} ⇒ general identifier

Table-Driven Scanners

Choosing a Category from an Ambiguous RE

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

Alternate Implementation Strategy (*Quite popular*)

- Build hash table of all identifiers & fold keywords into **RE** for *identifier*
- Preload keywords into hash table & set their categories
- 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 an error or EOF, back up to accepting state
- Need the scanner to return *token*, not boolean
  - *Token* is <*lexeme, Part of Speech*> pair
  - Use a map from **DFA**'s final state to Part of Speech (***PoS)***
    - → *"for" ends in a final state that maps to <u>for</u> while "ford" ends in a final state that maps to <u>identifier</u>*

EaC1e: This scanner differs significantly from the discussion in the 1st Edition of EaC.

Table-Driven Scanners

Handling a Stream of Words

// recognize words
state $\leftarrow s_0$
lexeme $\leftarrow$ *empty string*
clear stack
push (<u>bad</u>)

while (state $\neq s_e$*) do*
  char $\leftarrow$ *NextChar( )*
  lexeme $\leftarrow$ *lexeme + char*
  if state $\in S_A$
    then clear stack
  push (state)
  cat $\leftarrow$ *CharCat(char)*
  state $\leftarrow \delta$*(state,cat)*

  end;

// clean up final state
while (state $\notin S_A$ *and state* $\neq$ *<u>bad</u>) do*
  state $\leftarrow$ *pop()*
  truncate lexeme
  roll back the input one character
  end;

// report the results
if (state $\in S_A$*)*
  then return <PoS(state), lexeme>
  else return <u>invalid</u>

S_A is the set of accepting (e.g., final) states

PoS: state $\rightarrow$ *part of speech*

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

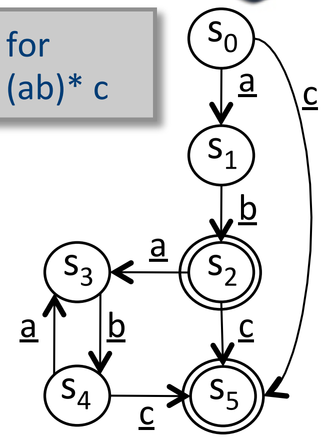
Avoiding Excess Rollback

Some REs can produce quadratic rollback

- Consider *ab | (ab)* c* and its **DFA**
- Input "ababababc"
  - $s_0, s_1, s_2, s_3, s_4, s_3, s_4, s_3, s_4, s_5$
- Input "abababab"
  - s_0, s_1, s_2, s3, s_4, s_3, s_4, s_3, s_4, rollback 6 characters
  - $s_0, s_1, s_2, s_3, s_4, s_3, s_4$, rollback 4 characters
  - s_0, s_1, s_2, s_3, s_4, rollback 2 characters
  - s_0, s_1, s_2

Not too pretty

DFA for
ab | (ab)* c



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

Note that Exercise 2.16 on page 82 of EaC2e is worded incorrectly. You can do better than the scheme shown in Figure 2.15, but cannot avoid, in the worst case, space proportional to the input string. (Alternative: fixed upper bound on token length)

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;
```

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

Maximal Munch Scanner

- Uses a bit array *Failed* to track dead-end paths
  - Initialize both *InputPos* & *Failed* in *InitializeScanner()*
  - *Failed* requires space ∝ |*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*

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 $\leftarrow s_0$;
while (state $\neq$ *exit) do*
  char $\leftarrow$ *NextChar( )*
  cat $\leftarrow$ *CharCat(char)*
  state $\leftarrow \delta$*(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 δ

Table-Driven Versus Direct-Coded Scanners

Overhead of Table Lookup

- Each lookup in CharCat or δ involves an address calculation and a memory operation
  - *CharCat(char) becomes*

    $@CharCat_0$ + char x w — w is sizeof(el't of CharCat)
  - *δ(state,cat) becomes*

    $@\delta_0$ + (state x cols + cat) x w — cols is # of columns in δ; w is sizeof(el't of δ)

- The references to *CharCat* and δ expand into multiple ops
- Fair amount of overhead work per character
- Avoid the table lookups and the scanner will run faster

We will see these expansions in Ch. 7.

Reference to an array or vector is almost always more expensive than to a scalar variable.

Building Faster Scanners from the DFA

A direct-coded recognizer for r *Digit Digit*

```
start:  accept ← s_e
        lexeme ← “”
        count ← 0
        goto s_0

s_0:    char ← NextChar
        lexeme ← lexeme + char
        count++
        if (char = ‘r’)
          then goto s_1
          else goto s_out

s_1:    char ← NextChar
        lexeme ← lexeme + char
        count++
        if (‘0’ ≤ char ≤ ‘9’)
          then goto s_2
          else goto s_out

s_2:    char ← NextChar
        lexeme ← lexeme + char
        count ← 0
        accept ← s_2
        if (‘0’ ≤ char ≤ ‘9’)
          then goto s_2
          else goto s_out

s_out:  if (accept ≠ s_e)
          then {
            for i ← 1 to count {
                RollBack()
                return <PoS,lexeme>
                }
          }
          else return <invalid,invalid>
```

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

Direct coding the maximal munch scanner is easy, too.

Building Faster Scanners from the DFA

A direct-coded recognizer for r *Digit Digit*

start: *accept* $\leftarrow s_e$
lexeme $\leftarrow$ “”
count $\leftarrow 0$
goto s_0

s_0: *char* $\leftarrow$ *NextChar*
lexeme $\leftarrow$ *lexeme* + *char*
count++
if (char = ‘r’)
then goto s_1
else goto s_{out}

s_1: *char* $\leftarrow$ *NextChar*
lexeme $\leftarrow$ *lexeme* + *char*
count++
if (‘0’ $\leq$ *char* $\leq$ ‘9’)
then goto s_2
else goto s_{out}

s_2: *char* $\leftarrow$ *NextChar*
lexeme $\leftarrow$ *lexeme* + *char*
count $\leftarrow 1$
accept $\leftarrow s_2$
if (‘0’ $\leq$ *char* $\leq$ ‘9’)
then goto s_2
else goto s_{out}

s_{out}: *if* (*accept* $\neq s_e$)
then begin
for i $\leftarrow$ *1 to count*
RollBack()
report success
end
else report failure

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

- Table lookup (with classification?)
- Binary search

What About Hand-Coded Scanners?

Many modern compilers use hand-coded scanners

- Starting from a **DFA** simplifies design & understanding
- There are things you can do that don't fit well into tool-based scanner
  - 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

      *value = value * 10 + digit – '0';*
  - 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 and works well

- 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 …

⇒ which leads to parsing …