COMP 412
FALL 2010

Lexical Analysis: Building Scanners

Comp 412

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

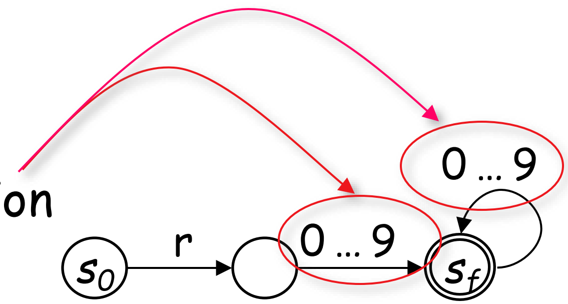


Table-Driven Scanners

Character Classification

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

state $\leftarrow s_0$;
while (*state* $\neq$ <u>*exit*</u>) *do*
  char $\leftarrow$ *NextChar*( ) *// read next character*
  cat $\leftarrow$ *CharCat*(*char*) *// classify character*
  state $\leftarrow \delta$(*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 1st*)

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
- Othersise, 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

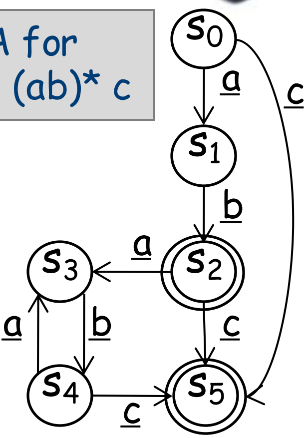
This scanner differs significantly from the ones shown in Chapter 2 of EaC 1e.

Avoiding Excess Rollback

- Some REs can produce quadratic rollback
  - Consider *ab | (ab)* c* and its DFA
  - Input "abababc"
    - → $s_0, s_1, s_3, s_4, s_3, s_4, s_3, s_4, s_5$
  - Input "ababab"
    - → $s_0, s_1, s_3, s_4, s_3, s_4, s_3, s_4$, rollback 6 characters
    - → $s_0, s_1, s_3, s_4, s_3, s_4$, rollback 4 characters
    - → s_0, s_1, s_3, s_4, rollback 2 characters
    - → s_0, s_1, s_3

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"

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;
```

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*

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 $\leftarrow s_0$;
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*
    $@CharCat_0 + char \times w$ w is sizeof(el't of CharCat)
  - *δ(state,cat) becomes*
    $@\delta_0 + (state \times cols + cat) \times w$ 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

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

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

Direct coding the maximal munch scanner is easy, too.

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 …