Lexical Analysis, IV

Implementing Scanners

Comp 412
Building a Scanner from a Collection of REs

We can build an RE for each kind of word in the input language

• We can build an RE for the language by combining the individual REs
  – \( RE_1 | RE_2 | RE_3 | \ldots | RE_k \)

• We can build a single DFA for this new RE
  – Thompson’s construction \( \rightarrow \) subset construction \( \rightarrow \) DFA minimization

Can we build a scanner \((\text{automatically})\) from this new RE?

• In practice, several important issues arise that make scanner construction harder than it might seem from an automata theory book
A **DFA** is not a Scanner

**A DFA recognizes a single word**

A scanner takes the entire input stream, breaks it into individual words, and classifies them.

**Finding all of the words**

- **DFA** stops at **EOF**
- Scanner needs to find a word & return it
- Scanner needs to start next call at the point where it stopped
  - Incremental, but continuous, pass over the input stream

**Classifying words**

- Scanner needs to return both syntactic category and lexeme
- Mapping from final state to syntactic category
  - Scanner must preserve enough final states to preserve this mapping
  - Can use a simple table lookup (vector with one entry per state)
A DFA is not a Scanner

The REs may be ambiguous

Individual REs are well formed, but the collection of them is not

• Is “then” a keyword or an identifier?
  – Typical approach is to let the compiler writer assign priorities
  – Lex and flex take priority from order in the specification
• In a given accepting state, highest priority wins
  – Mapping from $s \in S_A$ to category contains highest priority match

A given string of characters may match at multiple points

• In “donut”, does the scanner match “do”, “don”, “donu”, or “donut”
• Correct choice is typically defined to be the longest match

Scanner simulates the DFA until it makes an error transition

• From $s_e$, the scanner backs up to find an accepting state
  – Needs ability to back up in both the DFA and the input stream

Need to preserve that separate final state. See comments on minimization in last lecture.
A “Run to Error” Scanner

The body of `NextToken()`

```plaintext
// 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 (bad)
        push (state)
        state ← σ(state, char)

// back up to an accepting state
while (state ∉ Sₐ and state ≠ bad) do
    state ← pop()
    truncate lexeme by one character
    roll back the input one character

// report the results
if (state ∈ Sₐ) then
    return <PoS(state), lexeme>
else
    return <invalid, invalid>
```

Amount of rollback depends on the language
Typical rollback is 1 or 2 characters

```plaintext
PoS: state → part of speech

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

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

Plus a test for EOF and that latches to EOF
```
Recognizing Keywords in Hand-Coded Scanner

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
    → It is going to re-process the lexeme, anyway
  – Scanner is hand coded
• Otherwise, let the DFA handle them (O(1) cost per character)

This strategy processes the lexeme of an identifier twice

• Unavoidable if scanner is creating a table of identifiers (typical case)
  – Design should minimize number of times a character is touched
• Some programmers make it worse by using a separate keyword table
  – Classical use for “perfect” hashing, which makes it worse again
The Role of Whitespace

**What does the scanner do with whitespace?**

• This issue is language-specific

**Most Algol-like languages**

• Whitespace terminates a word, if you have it
  – “x = y + z” versus “x=y+z”
  – Both should (and do) scan the same

• Whitespace is not necessary *in most cases*

**Python**

– Whitespace is significant; blanks define block structure
– Scanner might count blanks and insert tokens for start block and end block

  ➔ *Scanner spots change in indentation & inserts either <start> or <end>*

**What about comments?**

• In many situations, they can be discarded
The Role of Whitespace

Fortran 66 was the poster child for problems with whitespace

By definition, whitespace was not significant in Fortran 66 or 77

• This simple issue caused a host of problems

```fortran
  do 10 k = 1, 100, 1
  ...
  10 continue
```

Fortran’s Do Loop

• Necessitated large (*but bounded*) lookahead
  – “do10k =1” versus “do10k=1,100” (and there are other examples)
  – Scanner needs to look for the comma in the right-hand side before it can disambiguate a “do” from an “identifier”

• Cannot express this with a regular expression-based scanner

I know of no other language that has followed this path

‡ Bounded because statement was limited to 1,320 characters.
The transition table, $\delta$, can grow quite large

Transition-table size can become a problem when it approaches the size of the L1 data cache (remember ELEC 220?)

Can we shrink the transition table?

<table>
<thead>
<tr>
<th></th>
<th>r</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_0$</td>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td></td>
</tr>
<tr>
<td>$S_1$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_e$</td>
<td></td>
</tr>
<tr>
<td>$S_2$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_e$</td>
<td></td>
</tr>
<tr>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td></td>
</tr>
</tbody>
</table>

Transition Table for $r[0...9]^+$

DFA for $r[0...9]^+$
The transition table, $\delta$, can grow quite large

Transition-table size can become a problem when it approaches the size of the L1 data cache

(remember ELEC 220 ?)

Can we shrink the transition table?

<table>
<thead>
<tr>
<th></th>
<th>r</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_e$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_2$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_e$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Transition Table for $r[0...9]^+$

These columns are identical
Represent them once
The transition table, $\delta$, can grow quite large

Transition-table size can become a problem when it approaches the size of the L1 data cache

(remember ELEC 220?)

Can we shrink the transition table?

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>$r$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td></td>
</tr>
<tr>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_e$</td>
<td></td>
</tr>
<tr>
<td>$s_2$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_e$</td>
<td></td>
</tr>
<tr>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td></td>
</tr>
</tbody>
</table>

Transition Table for $r[0...9]^+$

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>$r$</th>
<th>0...9</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
</tbody>
</table>

Compressed Table

Of course, we need to make the scanner work with this table ...
Table Size in a Table-Driven Scanner

Character Classification

• Group together characters by their actions in the DFA
  – Combine identical columns in the transition table, \( \delta \)
  – Using class rather than character to index \( \delta \) shrinks the table

\[
\begin{align*}
  state & \leftarrow s_0; \\
  while \ (state \neq \text{exit}) \ do & \\
  char & \leftarrow \text{NextChar}( ) \quad \text{// read next character} \\
  col & \leftarrow \text{CharClass}(char) \quad \text{// classify character} \\
  state & \leftarrow \delta(state, col) \quad \text{// take the transition}
\end{align*}
\]

• Idea works well in ASCII (or EBCDIC)
  – compact, byte-oriented character sets
  – limited range of values means compact \textit{CharClass} vector

• Not clear how it extends to larger character sets (16 bit or larger unicode)

Obvious algorithm is \( O(|\Sigma|^2 \cdot |S|) \). Can you do better?
Table Size in a Table-Driven Scanner

Compressing the Table
- Scanner generator must identify identical columns (see ⇒)
- Given MapTo, scanner generator can construct CharClass and the compressed table

Tricks to speed up identity test
- Keep a population count of each column’s non-error entries
- Radix sort columns by pop count & only compare equivalent cols
- Compute a more complex signature (such as sum of state #s)

Finding identical columns in $\delta$

```
for i ← 1 to NumCols
    MapTo[i] ← i

for i ← 1 to NumCols
    if MapTo[i] = i then
        for j ← i to NumCols
            if MapTo[j] = j then
                same ← true
        for k ← 1 to NumRows
            if $\delta(I,k) \neq \delta(j,k)$ then
                same ← false
                break
        if same then
            MapTo[j] ← I
```
Avoiding Excess Rollback

• Some REs can produce quadratic rollback
  – Consider \( ab \mid (ab)^* \) \( c \) and its DFA
  – Input “ababababc”
    \[ \rightarrow s_0, s_1, s_2, s_3, s_4, s_3, s_4, s_3, s_4, s_5 \]
  – Input “abababab”
    \[ \rightarrow s_0, s_1, s_2, s_3, s_4, s_3, s_4, s_3, s_4, \text{rollback 6 characters} \]
    \[ \rightarrow s_0, s_1, s_2, s_3, s_4, s_3, s_4, \text{rollback 4 characters} \]
    \[ \rightarrow s_0, s_1, s_2, s_3, s_4, \text{rollback 2 characters} \]
    \[ \rightarrow s_0, s_1, s_2 \]

• This behavior is preventable
  – Have the scanner **remember** paths that fail on particular inputs
    \[ \rightarrow \text{Keep a position-by-position map of the input string} \]
    \[ \rightarrow \text{Roll forward as before, but mark failed states on rollback from error state} \]
    \[ \rightarrow \text{Truncate the lexeme when it hits a “failed” state} \]
  – 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.
Maximal Munch Scanner

// recognize words
state ← s₀
lexeme ← empty string
clear stack
push (bad, 1)
while (state ≠ sₑ) do
  if Failed[state, InputPos] then
    ⟨state, InputPos⟩ ← pop()
    truncate lexeme
    break;
  char ← Input[InputPos]
  lexeme ← lexeme + char
  if state ∈ Sᴬ then
    clear stack
    push(<bad, 1>)
  push (state, InputPos)
col ← CharClass(char)
state ← δ(state, col)
InputPos ← InputPos + 1

// clean up final state
while (state ∉ Sᴬ and state ≠ bad) do
  if state ≠ sₑ then
    Failed[state, InputPos] ← true
    ⟨state, InputPos⟩ ← pop()
    truncate lexeme
  // report the results
  if (state ∈ Sᴬ) then
    return ⟨PoS(state), lexeme⟩
  else
    return ⟨invalid, invalid⟩

InitializeScanner()
InputPos ← 0
for each state s in the DFA do
  for i ← 0 to |input| do
    Failed[s, i] ← false
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
  – Design languages that do not cause quadratic rollback

Table-Driven Versus Direct-Coded Scanners

Table-driven scanners make heavy use of indexing

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

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

```plaintext
state ← s_0;
while (state ≠ exit) do
    char ← NextChar()
    cat ← CharCat(char)
    state ← δ(state, cat);
```

Code locality as opposed to random access in δ
Table-Driven Versus Direct-Coded Scanners

**Overhead of Table Lookup**

- Each lookup in CharCat or $\delta$ involves an address calculation and a memory operation
  
  - $\text{CharCat}(\text{char})$ becomes
    
    $\text{@CharCat}_0 + \text{char} \times w$
    
    $w$ is sizeof(elt of CharCat)
  
  - $\delta(\text{state,cat})$ becomes
    
    $\text{@}\delta_0 + (\text{state} \times \text{cols} + \text{cat}) \times w$
    
    $\text{cols}$ is # of columns in $\delta$
    
    $w$ is sizeof(elt of $\delta$)

- The references to $\text{CharCat}$ and $\delta$ 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 _Digit Digit_

```plaintext
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 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 _Digit Digit_

\[
\begin{align*}
\text{start:} & \quad \text{accept } \leftarrow s_e \\
& \quad \text{lexeme } \leftarrow "" \\
& \quad \text{count } \leftarrow 0 \\
& \quad \text{goto } s_0 \\
\text{s}_0: & \quad \text{char } \leftarrow \text{NextChar} \\
& \quad \text{lexeme } \leftarrow \text{lexeme } + \text{char} \\
& \quad \text{count }++ \\
& \quad \text{if (char } = \text{’r’)} \\
& \quad \quad \text{then goto } s_1 \\
& \quad \quad \text{else goto } s_{out} \\
\text{s}_1: & \quad \text{char } \leftarrow \text{NextChar} \\
& \quad \text{lexeme } \leftarrow \text{lexeme } + \text{char} \\
& \quad \text{count }++ \\
& \quad \text{if (’0’ } \leq \text{char } \leq \text{’9’)} \\
& \quad \quad \text{then goto } s_2 \\
& \quad \quad \text{else goto } s_{out} \\
\text{s}_2: & \quad \text{char } \leftarrow \text{NextChar} \\
& \quad \text{lexeme } \leftarrow \text{lexeme } + \text{char} \\
& \quad \text{count } \leftarrow 1 \\
& \quad \text{accept } \leftarrow s_2 \\
& \quad \text{if (’0’ } \leq \text{char } \leq \text{’9’)} \\
& \quad \quad \text{then goto } s_2 \\
& \quad \quad \text{else goto } s_{out} \\
\text{s}_{out}: & \quad \text{if (accept } \neq s_e) \\
& \quad \quad \text{then begin} \\
& \quad \quad \quad \text{for } i \leftarrow 1 \text{ to count} \\
& \quad \quad \quad \quad \text{RollBack}() \\
& \quad \quad \quad \quad \text{report success} \\
& \quad \quad \quad \text{end} \\
& \quad \quad \text{else report failure} 
\end{align*}
\]

Direct coding the maximal munch scanner is easy, too.

If end of state test is complex (e.g., many cases), scanner generator should consider other schemes
- Table lookup (with classification?)
- Binary search
What does it take to write your own scanner generator?

• Parser for the regular expressions
  – Top-down, recursive descent parser
  – Can perform Thompson’s construction during the parse

• Implementation of the subset construction
  – Tedious, but not difficult

• DFA Minimization
  – Hopcroft or Brozowski

• Code to compress and emit the table
  – Map final states to token types
Building a Scanner Generator

When the compiler writer joins together the REs for the words in the language, the final states determine which syntactic category is found.

• Thompson’s construction creates new final states at each step
  – Each NFA produced by Thompson’s construction has exactly 1 final state
  – Oops.

• Need to build the NFAs word-by-word, then join them to a new start state with $\varepsilon$-transitions
Building a Scanner Generator

When the compiler writer joins together the REs for the words in the language, the final states determine which syntactic category is found.

- Thompson’s construction creates new final states at each step
  - Each NFA produced by Thompson’s construction has exactly 1 final state
  - Oops.

- Need to build the NFAs word-by-word, then join them to a new start state with \( \varepsilon \)-transitions

- Now, scanner generator can apply the subset construction to the NFA
  - Subset construction preserves final states \((compresses prefixes)\)

- Next, scanner generator can minimize the NFA
  - Both Hopcroft and Brzozowski combine final states
Building a Scanner Generator

Preserving final states with minimization

Two choices

• Build, determinize, and minimize DFAs for each word; then combine them and determinize
  – Run more passes, but on smaller DFAs
  – Final DFA is not minimal, but it does have distinct final states

• Use Hopcroft’s algorithm, but change the initialization
  – Rather than clustering all final states in one set for the initial partition, only cluster together final states that have the same syntactic category
  – Hopcroft’s algorithm splits sets in the partition; it never combines them

What does flex do?

• flex builds RE by RE DFAs; next it combines them and determinizes them
• It does not perform minimization
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
      \[
      \text{value} = \text{value} \times 10 + \text{digit} - '0';
      \]
  – Combine similar states \( \text{(serial or parallel)} \)
• Scanners are fun to write
  – Compact, comprehensible, easy to debug, ...
  – Don’t get too cute \( \text{(e.g., perfect hashing for keywords)} \)
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 ...
Kleene’s Construction

for i ← 0 to |D| - 1;  // label each immediate path
    for j ← 0 to |D| - 1;
        R^0_{0j} ← \{ a | \delta(d_i, a) = d_j \};
        if (i = j) then
            R^0_{ii} = R^0_{0i} \cup \{ \varepsilon \};

for k ← 0 to |D| - 1;  // label nontrivial paths
    for i ← 0 to |D| - 1;
        for j ← 0 to |D| - 1;
            R^k_{ij} ← R^{k-1}_{ik} (R^{k-1}_{kk})^* R^{k-1}_{kj} \cup R^{k-1}_{ij}

L ← \{ \}  // union labels of paths from
For each final state s_i  // s_0 to a final state s_i
    L ← L \cup R^{(|D|-1)}_{0i}

R^k_{ij} is the set of paths from i to j that include no state higher than k

The Wikipedia page on “Kleene’s algorithm” is pretty good. It also contains a link to Kleene’s 1956 paper. This form of the algorithm is usually attributed to McNaughton and Yamada in 1960.