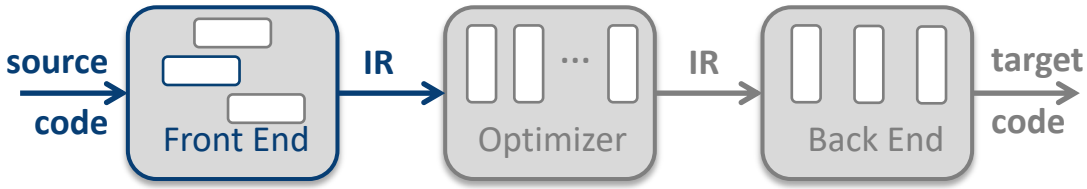
COMP 412
FALL 2018

Lexical Analysis, IV

Implementing Scanners

Comp 412



Section 2.5 in EaC2e

Building a Scanner from a Collection of **RE**s

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 **RE**s
  - $RE_1 \mid RE_2 \mid RE_3 \mid \ldots \mid RE_k$
- We can build a single **DFA** for this new **RE**
  - Thompson's construction → subset construction → **DFA** minimization

Can we build a scanner (*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

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

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

A "Run to Error" Scanner

The body of *NextToken()*

```
// 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 ← δ(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>
```

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

PoS: state → part of speech

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*)
- 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? (*blanks, tabs*)

- This issue is language-specific

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

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

```
      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.

Table Size in a Table-Driven Scanner

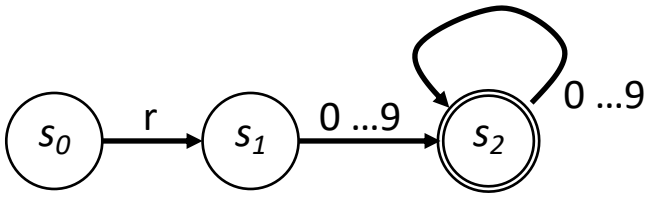
The transition table, δ, can grow quite large

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?

	r	0	1	2	3	4	5	6	7	8	9	Other
s_0	s_1	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e
s_1	s_e	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_e
s_2	s_e	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_e
s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e

Transition Table for r[0…9]$^+$



DFA for r [0…9]$^+$

Table Size in a Table-Driven Scanner

The transition table, δ, can grow quite large

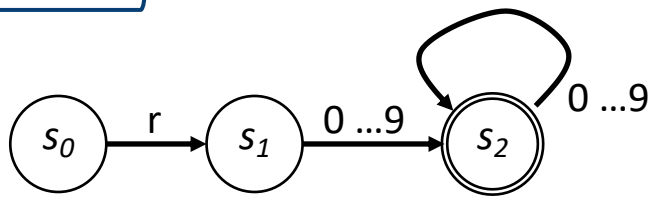
Transition-table size can become a problem when it approaches the size of the **L1** data cache (*remember ELEC 220 ?*)

Can we shrink the transition table?

	r	0	1	2	3	4	5	6	7	8	9	Other
s_0	s_1	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e
s_1	s_e	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_e
s_2	s_e	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_e
s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e

Transition Table for r[0…9]+

These columns are identical
Represent them once



DFA for r [0…9]+

Table Size in a Table-Driven Scanner

The transition table, δ, 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?

	r	0	1	2	3	4	5	6	7	8	9	Other
s_0	s_1	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e
s_1	s_e	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_e
s_2	s_e	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_e
s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e

Transition Table for r[0…9]⁺

	r	0	Other
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

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, δ
  - Indexing δ by a character's class shrinks the table

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

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

Obvious algorithm is **O**($|\Sigma|^2 \bullet |S|$). 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 compression

- 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

```
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 δ(I,k) = δ(j,k) then
                        same ← false
                        break
                if same then
                    MapTo[j] ← i
```

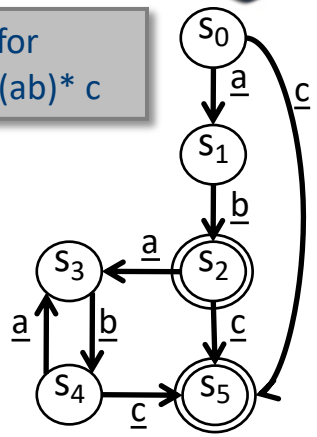
Finding identical columns in δ

Avoiding Excess Rollback

- Some **RE**s can produce quadratic rollback
  - Consider *ab | (ab)* c* and its **DFA**
  - Input "abababab**c**"
    - → $s_0, s_1, s_2, s_3, s_4, s_3, s_4, s_3, s_4, s_5$
  - Input "abababa**b**"
    - → $s_0, s_1, s_2, s3, s_4, s_3, s_4, s_3, s_4$, 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,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
          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*

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

Building a Scanner Generator

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 **NFA**s word-by-word, then join them to a new start state with ε-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 **NFA**s word-by-word, then join them to a new start state with ε-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 **DFA**s for each word; then combine them and determinize
  - Run more passes, but on smaller **DFA**s
  - 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 DFA**s; 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

      *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 …