Compiling Techniques
Lecture 4: Constructing a Scanner from Regular Expressions

Christophe Dubach
Overview

- Regular Expressions to NFAs
- NFAs to DFAs
- Minimisation of DFAs
- Limits of Regular Languages
Quick Review

- Previous class:
  - The scanner is the first stage in the front end
  - Specifications can be expressed using regular expressions
  - Build tables and code from a DFA
  - Regular expressions, NFAs and DFAs
Goal

- Construct a finite state automaton to recognise any RE

Overview:
- Direct construction of a nondeterministic finite automaton (NFA) to recognise a given RE
  - Requires $\epsilon$-transitions to combine regular subexpressions
- Construct a deterministic finite automaton (DFA) to simulate the NFA
  - Use a set-of-states construction
- Minimise the number of states
  - Hopcroft state minimisation algorithm
- Generate the scanner code
  - Additional specifications needed for details
RE→NFA

Key idea:
× NFA pattern for each symbol & each operator
× Join them with ε moves in precedence order
Example

1. $a, b, \& c$

2. $b | c$

3. $(b | c)^*$
Example continued

\[ a(b | c)^* \]

\[
\begin{array}{c}
S_0 \xrightarrow{a} S_1 \xrightarrow{\varepsilon} S_2 \xrightarrow{\varepsilon} S_3 \\
S_4 \xrightarrow{\varepsilon} S_5 \xrightarrow{\varepsilon} S_6 \xrightarrow{c} S_7 \xrightarrow{\varepsilon} S_8 \\
S_9
\end{array}
\]

Of course, a human would design something simpler ...

But, we can automate production of the more complex one ...
NFA → DFA (Subset Constr.)

- Need to build a simulation of the NFA

- Two key functions
  - \( \text{Move}(s_i, a) \) is set of states reachable from \( s_i \) by \( a \)
  - \( \varepsilon\)-closure\((s_i) \) is set of states reachable from \( s_i \) by \( \varepsilon \)

- The algorithm:
  - Start state derived from \( s_0 \) of the NFA
  - Take its \( \varepsilon\)-closure \( S_0 = \varepsilon\)-closure\((s_0) \)
  - Take the image of \( S_0 \), \( \text{Move}(S_0, \alpha) \) for each \( \alpha \in \Sigma \), and take its \( \varepsilon\)-closure
  - Iterate until no more states are added
NFA→DFA (Subset Constr.)

The algorithm:

\[ s_0 \leftarrow \varepsilon\text{-closure}(q_{0n}) \]

while ( S is still changing )

for each \( s_i \in S \)

for each \( \alpha \in \Sigma \)

\[ s_? \leftarrow \varepsilon\text{-closure}(\text{Move}(s, \alpha)) \]

if ( \( s_? \notin S \) ) then

\( s_? \) to \( S \) as \( s_j \)

\[ T[s, \alpha] \leftarrow s_j \]

Let's think about why this works

The algorithm halts:

1. \( S \) contains no duplicates (test before adding)
2. \( 2^{Q_0} \) is finite
3. while loop adds to \( S \), but does not remove from \( S \) (monotone)

\[ \Rightarrow \text{the loop halts} \]

\( S \) contains all the reachable NFA states

It tries each character in each \( s_r \)

It builds every possible NFA configuration.

\[ \Rightarrow S \text{ and } T \text{ form the DFA} \]
Example of a fixed-point computation
- Monotone construction of some finite set
- Halts when it stops adding to the set
- Proofs of halting & correctness are similar
- These computations arise in many contexts

Other fixed-point computations
- Canonical construction of sets of LR(1) items
  - Quite similar to the subset construction
- Classic data-flow analysis (& Gaussian Elimination)
  - Solving sets of simultaneous set equations
NFA→DFA (Subset Constr.)

Applying the subset construction:

<table>
<thead>
<tr>
<th></th>
<th>NFA states</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>s₀</td>
<td>q₀</td>
<td>q₁, q₂, q₃, q₄, q₅, q₆, q₉</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>s₁</td>
<td>q₁, q₂, q₃, q₄, q₅, q₆, q₉</td>
<td>none</td>
<td>q₅, q₈, q₉, q₃, q₄, q₆</td>
</tr>
<tr>
<td></td>
<td>s₂</td>
<td>q₅, q₈, q₉, q₃, q₄, q₆</td>
<td>none</td>
<td>s₂</td>
</tr>
<tr>
<td></td>
<td>s₃</td>
<td>q₇, q₈, q₉, q₃, q₄, q₆</td>
<td>none</td>
<td>s₂</td>
</tr>
</tbody>
</table>

Final states
- DFA for a ( b | c )*
  - Ends up smaller than the NFA
  - All transitions are deterministic
  - Use same code skeleton as before
Where are we?

- **RE → NFA** (Thompson’s construction) \(\checkmark\)
  - Build an NFA for each term
  - Combine them with \(\varepsilon\)-moves
- **NFA → DFA** (subset construction) \(\checkmark\)
  - Build the simulation
- **DFA → Minimal DFA**
  - Hopcroft’s algorithm
DFA Minimisation

- The Big Picture
- Discover sets of equivalent states
- Represent each such set with just one state
DFA Minimisation

Two states are equivalent if and only if:

- The set of paths leading to them are equivalent
- $\forall \alpha \in \Sigma$, transitions on $\alpha$ lead to equivalent states (DFA)
- $\alpha$-transitions to distinct sets $\Rightarrow$ states must be in distinct sets
DFA Minimisation

- A partition $P$ of $S$
  - Each $s \in S$ is in exactly one set $p_i \in P$
  - The algorithm iteratively partitions the DFA’s states
DFA Minimisation

- Key idea of the algorithm
  - Group states into maximal size sets, optimistically
  - Iteratively subdivide those sets, as needed
  - States that remain grouped together are equivalent
DFA Minimisation

- Initial partition, $P_0$, has two sets:
  - $\{F\}$ & $\{Q-F\}$  ($D = (Q, \Sigma, \delta, q_0, F)$)

- Splitting a set (“partitioning a set by $a$”)
  - Assume $q_a \in s$, and $\delta(q_a, a) = q_x$ & $\delta(q_b, a) = q_y$
  - If $q_x$ & $q_y$ are not in the same set, then $s$ must be split
    - one state in the final DFA cannot have two transitions on $a$
  - If $q_a$ has transition on $a$ and $q_b$ does not $\Rightarrow$ $a$ splits $s
DFA Minimisation

The algorithm

- \( P \leftarrow \{ F, \{Q-F\}\} \)
- while (\( P \) is still changing)
  - \( T \leftarrow \{ \} \)
  - for each set \( S \in P \)
    - for each \( \alpha \in \Sigma \)
      - partition \( S \) by \( \alpha \)
        - into \( S_1 \) and \( S_2 \)
      - \( T \leftarrow T \cup S_1 \cup S_2 \)
    - if \( T \neq P \) then
      - \( P \leftarrow T \)

Why does this work?

- Partition \( P \in 2^Q \)
- Start off with 2 subsets of \( Q \)
  - \( \{F\} \) and \( \{Q-F\} \)
- While loop takes \( P_i \rightarrow P_{i+1} \) by
  - splitting 1 or more sets
- \( P_{i+1} \) is at least one step closer
  - to the partition with \( |Q| \) sets
- Maximum of \( |Q| \) splits

Note that

- Partitions are never combined
- Initial partition ensures that
  - final states are intact

This is a fixed-point algorithm!
DFA Minimisation

- Refining the algorithm
  - As written, it examines every $S \in P$ on each iteration
    - This does a lot of unnecessary work
    - Only examine $S$ if some $T$, reachable from $S$, has split

- Reformulate the algorithm using a “worklist”
  - Start worklist with initial partition, $F$ and $\{Q-F\}$
  - When it splits $S$ into $S_1$ and $S_2$, place $S_2$ on worklist

- This version looks at each $S \in P$ many fewer times ⇒ Well-known, widely used algorithm due to John Hopcroft
Example RE to Min-DFA

Start with a regular expression

```
| r0 | r1 | r2 | r3 | r4 | r5 | r6 | r7 | r8 | r9 |
```
Example RE to Min-DFA

Thompson’s construction produces

To make it fit, we’ve eliminated the \( \varepsilon \)-transition between “r” and “0”.
Example RE to Min-DFA

The subset construction builds

This is a DFA, but it has a lot of states ...
Example RE to Min-DFA

The DFA minimization algorithm builds

This looks like what a skilled compiler writer would do!
Limits of Regular Languages

* Advantages of Regular Expressions
  * Simple & powerful notation for specifying patterns
  * Automatic construction of fast recognisers
  * Many kinds of syntax can be specified with REs
Limits of Regular Languages

- Not all languages are regular RL’s ⊂ CFL’s ⊂ CSL’s  
  (R=regular, CF=context-free, CS=context-sensitive)

- Cannot construct DFA’s to recognise:
  - L= \{wcw | w \in \Sigma^*\}
  - This is not a regular language (cannot be expressed with RE)

- But, this is a little subtle. You can construct DFA’s for
  - Strings with alternating 0’s and 1’s: ( \epsilon | 1 ) ( 01 )^* ( \epsilon | 0 )
  - Strings with even number of 0’s or 1’s
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
  do 10 i = 1,25 DO 10 I = 1,25 (LOOP)
do 10 i = 1.25 DO10I = 1.25 (ASSIGNMENT)

- String constants with special characters
  newline, tab, quote, comment delimiters, ...
  (C, C++, Java, ...)

- Finite closures
  → Limited identifier length
  → Adds states to count length
  (Fortran 66 & Basic)
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
Preview

- Context-Free Grammars
- Introduction to Parsing