Compiling Techniques
Lecture 3: Introduction to Lexical Analysis
Christophe Dubach
Overview

- Tutorials
- The Big Picture
- Regular Expressions
- DFAs and NFAs
- Automating Scanner Construction
Tutorials

- Monday 1:10pm - AT 4.07 (Christophe Dubach)
- Monday 1:10pm - AT 4.14A (Björn Franke)
- Thursday 1:10pm - AT 4.07 (Christophe Dubach)

- Tutorials start next week
- Group allocation on course website
- Online Forum: https://piazza.com/
  - action: enrol today!
Maps character stream into words—the basic unit of syntax

- Produces pairs — a word & its part of speech
  - $x=x+y$; becomes $<id,x>=<id,x>+<id,y>$;
  - word $\equiv$ lexeme, part of speech $\equiv$ token type
  - In casual speech, we call the pair a token
- Typical tokens include number, identifier, +, -, new, while, if
- Scanner eliminates white space (including comments)
- Speed is important
The Big Picture

- Why study lexical analysis?
  - We want to avoid writing scanners by hand
  - We want to harness the theory from other classes

- Goals:
  - To simplify specification & implementation of scanners
  - To understand the underlying techniques and technologies
The Big Picture

Source code → Scanner → parts of speech & words

Specifications → Scanner Generator

- Represent words as indices into a global table
- Specifications written as "regular expressions"
- tables or code
Regular Expressions

- Lexical patterns form a regular language
- Any finite language is regular
- Regular expressions (REs) describe regular languages

- Regular Expression (over alphabet $\Sigma$)
  - $\epsilon$ is a RE denoting the set \{\epsilon\}
  - If $a$ is in $\Sigma$, then $a$ is a RE denoting \{a\}
  - If $x$ and $y$ are REs denoting $L(x)$ and $L(y)$ then
    - $x|y$ is an RE denoting $L(x) \cup L(y)$
    - $xy$ is an RE denoting $L(x)L(y)$
    - $x^*$ is an RE denoting $L(x)^*$

- Precedence is closure, then concatenation, then alternation
# Set Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Union of ( L ) and ( M )</strong></td>
<td>( L \cup M = { s \mid s \in L \text{ or } s \in M } )</td>
</tr>
<tr>
<td>Written ( L \cup M )</td>
<td></td>
</tr>
<tr>
<td><strong>Concatenation of ( L ) and ( M )</strong></td>
<td>( LM = { st \mid s \in L \text{ and } t \in M } )</td>
</tr>
<tr>
<td>Written ( LM )</td>
<td></td>
</tr>
<tr>
<td><strong>Kleene closure of ( L )</strong></td>
<td>( L^* = \bigcup_{0 \leq i \leq \infty} L^i )</td>
</tr>
<tr>
<td>Written ( L^* )</td>
<td></td>
</tr>
<tr>
<td><strong>Positive Closure of ( L )</strong></td>
<td>( L^+ = \bigcup_{1 \leq i \leq \infty} L^i )</td>
</tr>
<tr>
<td>Written ( L^+ )</td>
<td></td>
</tr>
</tbody>
</table>
Example

**Identifiers:**

- \( \text{Letter} \rightarrow (a|b|c|\ldots|z|A|B|C|\ldots|Z) \)
- \( \text{Digit} \rightarrow (0|1|2|\ldots|9) \)
- \( \text{Identifier} \rightarrow \text{Letter} (\text{Letter} | \text{Digit})^* \)

**Numbers:**

- \( \text{Integer} \rightarrow (\pm|\varepsilon)(0|1|2|3|\ldots|9)(\text{Digit}^*) \)
- \( \text{Decimal} \rightarrow \text{Integer} \cdot \text{Digit}^* \)
- \( \text{Real} \rightarrow (\text{Integer} | \text{Decimal}) \cdot (\pm|\varepsilon) \text{Digit}^* \)
- \( \text{Complex} \rightarrow (\text{Real} \cdot \text{Real}) \)

**NUMBERS CAN GET MORE COMPLICATED!**
Scanners & Regular Expressions

- Regular expressions can be used to specify the words to be translated to parts of speech by a lexical analyser.
- Using results from automata theory and theory of algorithms, we can automatically build recognisers from regular expressions.
- Some of you may have seen this construction for string pattern matching.
- We study REs and associated theory to automate scanner construction!
Example

- Consider the problem of recognising register names
  - Register → r (0|1|2| ... | 9) (0|1|2| ... | 9)*
  - Allows registers of arbitrary number
  - Requires at least one digit

RE corresponds to a recogniser (or DFA, Deterministic Finite Automaton)
Example (continued)

- DFA operation
  - Start in state s0 & take transitions on each input character
  - DFA accepts a word $x$ iff $x$ leaves it in a final state (s2)
- So,
  - $r17$ takes it through s0, s1, s2 and accepts
  - $r$ takes it through s0, s1 and fails
  - $a$ takes it straight to failure
To be useful, recogniser must turn into code

<table>
<thead>
<tr>
<th>State</th>
<th>Char</th>
<th>r</th>
<th>0,1,2,3,4,5,6,7,8,9</th>
<th>All others</th>
</tr>
</thead>
<tbody>
<tr>
<td>(s_0)</td>
<td>(s)</td>
<td>(s)</td>
<td>(s)</td>
<td>(s)</td>
</tr>
<tr>
<td>(s_1)</td>
<td>(s)</td>
<td>(s)</td>
<td>(s_2)</td>
<td>(s)</td>
</tr>
<tr>
<td>(s_2)</td>
<td>(s)</td>
<td>(s)</td>
<td>(s_2)</td>
<td>(s)</td>
</tr>
<tr>
<td>(s_e)</td>
<td>(s)</td>
<td>(s)</td>
<td>(s)</td>
<td>(s)</td>
</tr>
</tbody>
</table>

**Skeleton recognizer**

**Table encoding RE**
Example (continued)

- To be useful, recogniser must turn into code

<table>
<thead>
<tr>
<th>δ</th>
<th>r</th>
<th>0,1,2,3,4,5,6,7,8,9</th>
<th>All others</th>
</tr>
</thead>
<tbody>
<tr>
<td>s₀</td>
<td>s₁</td>
<td>sₑ</td>
<td>sₑ</td>
</tr>
<tr>
<td></td>
<td>start</td>
<td>error</td>
<td>error</td>
</tr>
<tr>
<td>s₁</td>
<td>sₑ</td>
<td>s₂</td>
<td>sₑ</td>
</tr>
<tr>
<td></td>
<td>error</td>
<td>add</td>
<td>error</td>
</tr>
<tr>
<td>s₂</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
</tr>
<tr>
<td></td>
<td>error</td>
<td>add</td>
<td>error</td>
</tr>
<tr>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
</tr>
<tr>
<td></td>
<td>error</td>
<td>error</td>
<td>error</td>
</tr>
</tbody>
</table>
Extended Example

- What if we need a tighter specification?
- \( r \) Digit Digit* allows arbitrary numbers
  - Accepts r00000
  - Accepts r99999
- What if we want to limit it to r0 through r31?
- Write a tighter regular expression
  - Register → r ( (0|1|2) (Digit | \( \varepsilon \)) | (4|5|6|7|8|9) | (3|30|31) )
  - Register → r0|r1|r2| ... |r31|r00|r01|r02| ... |r09
- Produces a more complex DFA
  - Has more states
  - Same cost per transition
  - Same basic implementation
Extended Example (continued)

- The DFA for Register → r ( (0|1|2) (Digit | ε) | (4|5|6|7|8|9) | (3|30|31) )

- Accepts a more constrained set of registers
- Same set of actions, more states
Extended Example (continued)

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>$r$</th>
<th>0,1</th>
<th>2</th>
<th>3</th>
<th>4-9</th>
<th>All others</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>
</tr>
<tr>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_5$</td>
<td>$s_4$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$s_e$</td>
<td>$s_3$</td>
<td>$s_3$</td>
<td>$s_3$</td>
<td>$s_3$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_3$</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>
</tr>
<tr>
<td>$s_4$</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>
</tr>
<tr>
<td>$s_5$</td>
<td>$s_e$</td>
<td>$s_6$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_6$</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>
</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>
</tr>
</tbody>
</table>

Table encoding RE for the tighter register specification

RUNS IN THE SAME SKELETON RECOGNISER!
Goal

- We will show how to construct a finite state automaton to recognise any RE
- Overview:
  - Direct construction of a **nondeterministic finite automaton (NFA)** to recognise a given RE
    - Requires $\varepsilon$-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
Non-Deterministic Finite Automata

- Each RE corresponds to a deterministic finite automaton (DFA)
- May be hard to directly construct the right DFA
- What about an RE such as (a|b)*abb?

- This is a little different
  - S0 has a transition on ε
  - S1 has two transitions on a
  - This is a non-deterministic finite automaton (NFA)
Non-Deterministic Finite Automata

- An NFA accepts a string $x$ iff $\exists$ a path through the transition graph from $s_0$ to a final state such that the edge labels spell $x$.
- Transitions on $\varepsilon$ consume no input.
- To “run” the NFA, start in $s_0$ and guess the right transition at each step.
  - Always guess correctly.
  - If some sequence of correct guesses accepts $x$ then accept.
- Why study NFAs?
  - They are the key to automating the RE→DFA construction.
  - We can paste together NFAs with $\varepsilon$-transitions.
Relationship between NFAs and DFAs

- DFA is a special case of an NFA
  - DFA has no $\varepsilon$ transitions
  - DFA’s transition function is single-valued
  - Same rules will work
- DFA can be simulated with an NFA
  - Obviously
- NFA can be simulated with a DFA
  - Less obvious
  - Simulate sets of possible states
  - Possible exponential blowup in the state space
  - Still, one state per character in the input stream
Automating Scanner Construction

To convert a specification into code:
1. Write down the RE for the input language
2. Build a big NFA
3. Build the DFA that simulates the NFA
4. Systematically shrink the DFA
5. Turn it into code

Scanner generators
- Lex and Flex work along these lines
- Algorithms are well-known and well-understood
- Key issue is interface to parser (define all parts of speech)
- You could build one in a weekend!
Automating Scanner Construction

- RE → NFA (Thompson’s construction)
  - Build an NFA for each term
  - Combine them with ε-moves
- NFA → DFA (subset construction)
  - Build the simulation
- DFA → Minimal DFA
  - Hopcroft’s algorithm
- DFA → RE (Not part of the scanner construction)
  - All pairs, all paths problem
Constructing a Scanner from Regular Expressions