2. Lexical Analysis

Oscar Nierstrasz

Thanks to Jens Palsberg and Tony Hosking for their kind permission to reuse and adapt the CS132 and CS502 lecture notes.
http://www.cs.ucla.edu/~palsberg/
http://www.cs.purdue.edu/homes/hosking/
Roadmap

> Introduction
> Regular languages
> Finite automata recognizers
> From RE to DFAs and back again
> Limits of regular languages

See, Modern compiler implementation in Java (Second edition), chapter 2.
Roadmap

> **Introduction**
> **Regular languages**
> **Finite automata recognizers**
> **From RE to DFAs and back again**
> **Limits of regular languages**
Lexical Analysis

1. Maps sequences of characters to tokens
2. Eliminates white space (tabs, blanks, comments etc.)

The string value of a token is a lexeme.

Source Code → **Scanner** → Tokens → **Parser** → IR

\[ x = x + y \] → \[ <ID,x> <EQ> <ID,x> <PLUS> <ID,y> \]
How to specify rules for token classification?

A scanner must recognize various parts of the language’s syntax.

Some parts are easy:

**White space**

\[
\begin{align*}
<\text{ws}> & ::= <\text{ws}> \ ' ' \\
& \quad | <\text{ws}> \ \backslash t' \\
& \quad | ' ' \\
& \quad | \ ' t'
\end{align*}
\]

**Keywords and operators**

specified as literal patterns: do, end

**Comments**

opening and closing delimiters: /* ... */
Specifying patterns

Other parts are harder:

**Identifiers**
- alphabetic followed by \( k \) alphanumerics (\_, \$, \&, \ldots)\)

**Numbers**
- integers: 0 or digit from 1–9 followed by digits from 0–9
- decimals: integer '.' digits from 0–9
- reals: (integer or decimal) \( 'E' \) (+ or –) digits from 0–9
- complex: \( '(' \text{ real }, 'real ')' \)

*We need an expressive notation to specify these patterns!*

why don’t we write it by hand?
Roadmap

- Introduction
- **Regular languages**
  - Finite automata recognizers
  - From RE to DFAs and back again
  - Limits of regular languages
A *language* is a set of strings

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

How do you define a language?
Recognizer.
Production grammar.
Production Grammars

> Powerful formalism for language description
  — Non-terminals (A, B)
  — Terminals (a,b)
  — Production rules (A→abA)
  — Start symbol (S0)
> Rewriting
**Detail: The Chomsky Hierarchy**

> **Type 0**: $\alpha \rightarrow \beta$
> — Unrestricted grammars generate recursively enumerable languages. Minimal requirement for recognizer: Turing machine.

> **Type 1**: $\alpha A \beta \rightarrow \alpha \gamma \beta$
> — Context-sensitive grammars generate context-sensitive languages, recognizable by linear bounded automata

> **Type 2**: $A \rightarrow \gamma$
> — Context-free grammars generate context-free languages, recognizable by non-deterministic push-down automata

> **Type 3**: $A \rightarrow a$ and $A \rightarrow aB$
> — Regular grammars generate regular languages, recognizable by finite state automata

*NB: A is a non-terminal; $\alpha$, $\beta$, $\gamma$ are strings of terminals and non-terminals*

**Individual identifiers** in a classical programming language form a regular language.
The language is on the other hand **context free** most of the time.
Grammars for regular languages

*Regular grammars generate regular languages*

**Definition:**
In a regular grammar, all productions have one of two forms:
1. $A \rightarrow aA$
2. $A \rightarrow a$
   where $A$ is any non-terminal and $a$ is any terminal symbol

These are also called type 3 grammars (Chomsky)
Regular languages can be described by *Regular Expressions*

*Regular expressions (RE) over an alphabet* $\Sigma$:

1. $\varepsilon$ is a RE denoting the set $\{\varepsilon\}$
2. If $a \in \Sigma$, then $a$ is a RE denoting $\{a\}$
3. If $r$ and $s$ are REs denoting $L(r)$ and $L(s)$, then:
   1. $(r) \mid (s)$ is a RE denoting $L(r) \cup L(s)$
   2. $(r)(s)$ is a RE denoting $L(r)L(s)$
   3. $(r)^*$ is a RE denoting $L(r)^*$

We adopt a *precedence* for operators: *Kleene closure*, then *concatenation*, then *alternation* as the order of precedence.

For any RE $r$, there exists a grammar $g$ such that $L(r) = L(g)$

---

Epsilon (the set with the “empty” string)
As you can see, we don’t define $a^+$ or $[a]$
Patterns are often specified as *regular languages*.
Notations used to describe a regular language (or a regular set) include both *regular expressions* and *regular grammars*
Examples

Let $\Sigma = \{a,b\}$

- $a \mid b$ denotes $\{a,b\}$
- $(a \mid b) (a \mid b)$ denotes $\{aa,ab,ba,bb\}$
- $a^*$ denotes $\{\varepsilon,a,aa,aaa,\ldots\}$
- $(a \mid b)^*$ denotes the set of all strings of a’s and b’s (including $\varepsilon$)
- Universi(ä | ae)t Bern(e |) ...
### Algebraic properties of REs

<table>
<thead>
<tr>
<th>Operation</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r \mid s = s \mid r$</td>
<td>is commutative</td>
</tr>
<tr>
<td>$r \mid (s \mid t) = (r \mid s) \mid t$</td>
<td>is associative</td>
</tr>
<tr>
<td>$r (st) = (rs)t$</td>
<td>concatenation is associative</td>
</tr>
<tr>
<td>$r(s \mid t) = rs \mid rt$</td>
<td>concatenation distributes over $</td>
</tr>
<tr>
<td>$(s \mid t)r = sr \mid tr$</td>
<td></td>
</tr>
<tr>
<td>$\varepsilon r = r$</td>
<td>$\varepsilon$ is the identity for concatenation</td>
</tr>
<tr>
<td>$r \varepsilon = r$</td>
<td></td>
</tr>
<tr>
<td>$r^* = (r \mid \varepsilon)^*$</td>
<td>$\varepsilon$ is contained in $*$</td>
</tr>
<tr>
<td>$r^{**} = r^*$</td>
<td>$*$ is idempotent</td>
</tr>
</tbody>
</table>
### Examples of using REs to specify lexical patterns

<table>
<thead>
<tr>
<th><strong>Identifiers</strong></th>
<th><strong>Numbers</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>letter → (a</td>
<td>b</td>
</tr>
<tr>
<td>digit → (0</td>
<td>1</td>
</tr>
<tr>
<td>id → letter ( letter</td>
<td>digit )*</td>
</tr>
<tr>
<td>integer → (+</td>
<td>—</td>
</tr>
<tr>
<td>decimal → integer . ( digit )*</td>
<td></td>
</tr>
<tr>
<td>real → ( integer</td>
<td>decimal ) E (+</td>
</tr>
<tr>
<td>complex → ’( real , real )’</td>
<td></td>
</tr>
</tbody>
</table>

Numbers can get much more complicated.
Most programming language tokens can be described with REs.
REs are cool for specifying.
FAs are good for implementing REs.
Recognizers

| letter  | → (a | b | c | ... | z | A | B | C | ... | Z) |
|--------|----------------------------------|
| digit  | → (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9) |
| id     | → letter ( letter | digit )* |

From a regular expression we can construct a **deterministic finite automaton** (DFA)

DFA for recognizing an identifier.
why D? why F? why A?
I.e., encode the transitions in the next_state matrix
Two tables control the recognizer

<table>
<thead>
<tr>
<th>char_class</th>
<th>char</th>
<th>a-z</th>
<th>A-Z</th>
<th>0-9</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>letter</td>
<td>letter</td>
<td>digit</td>
<td>other</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>next_state</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>letter</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>digit</td>
<td>3</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>other</td>
<td>3</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

To change languages, we can just change tables
## Automatic construction

> **Scanner generators** automatically construct code from regular expression-like descriptions
  > — construct a DFA
  > — use *state minimization* techniques
  > — emit code for the scanner (table driven or direct code)

> A key issue in automation is an interface to the parser

> *lex* is a scanner generator supplied with UNIX
  > — emits C code for scanner
  > — provides macro definitions for each token (used in the parser)
  > — nowadays JavaCC is more popular
What about the RE \((a | b)^*abb\) ?

State \(s_0\) has multiple transitions on \(a\)!

*This is a non-deterministic finite automaton*
Review: Finite Automata

A non-deterministic finite automaton (NFA) consists of:
1. a set of states \( S = \{ s_0, \ldots, s_n \} \)
2. a set of input symbols \( \Sigma \) (the alphabet)
3. a transition function \( \delta \) mapping state-symbol pairs to sets of states
4. a distinguished start state \( s_0 \)
5. a set of distinguished accepting (final) states \( F \)

A Deterministic Finite Automaton (DFA) is a special case of an NFA:
1. no state has a \( \epsilon \)-transition, and
2. for each state \( s \) and input symbol \( a \), there is at most one edge labeled \( a \) leaving \( s \).

A DFA accepts \( x \) iff there exists a unique path through the transition graph from the \( s_0 \) to an accepting state such that the labels along the edges spell \( x \).
Example: the set of strings containing an even number of zeros and an even number of ones

The RE is \((00 | 11)^*((01 | 10)(00 | 11)^*(01 | 10)(00 | 11))^*)^*

Note how the RE walks through the DFA.

NB: The states capture whether there are an even number of 0s or 1s => 4 possible states.
DFAs and NFAs are equivalent

1. DFAs are a subset of NFAs

2. Any NFA can be converted into a DFA, by *simulating sets of simultaneous states*:
   — each DFA state corresponds to a set of NFA states
   — NB: possible exponential blowup
NFA to DFA using the subset construction
Roadmap

> Introduction
> Regular languages
> Finite automata recognizers
> From RE to DFAs and back again
> Limits of regular languages
Constructing a DFA from a RE

> RE $\rightarrow$ NFA
  — Build NFA for each term; connect with $\varepsilon$ moves

> NFA $\rightarrow$ DFA
  — Simulate the NFA using the subset construction

> DFA $\rightarrow$ minimized DFA
  — Merge equivalent states

> DFA $\rightarrow$ RE
  — Construct $R^k_{ij} = R^{k-1}_{ik}(R^{k-1}_{kk})^* R^{k-1}_{kj} \cup R^{k-1}_{ij}$
  — Or convert via Generalized NFA (GNFA)
RE to NFA example: \((a \mid b)^*abb\)
NFA to DFA: the subset construction

**Input:** NFA $N$

**Output:** DFA $D$ with states $S_D$ and transitions $T_D$ such that $L(D) = L(N)$

**Method:** Let $s$ be a state in $N$ and $P$ be a set of states. Use the following operations:

- $\varepsilon$-closure($s$) — set of states of $N$ reachable from $s$ by $\varepsilon$ transitions alone
- $\varepsilon$-closure($P$) — set of states of $N$ reachable from some $s$ in $P$ by $\varepsilon$ transitions alone
- $\text{move}(T,a)$ — set of states of $N$ to which there is a transition on input $a$ from some $s$ in $P$

add state $P = \varepsilon$-closure($s_0$) unmarked to $S_D$

while $\exists$ unmarked state $P$ in $S_D$

mark $P$

for each input symbol $a$

$U = \varepsilon$-closure($\text{move}(P,a)$)

if $U \notin S_D$

then add $U$ unmarked to $S_D$

$T_D[P,a] = U$

end for

end while

$\varepsilon$-closure($s_0$) is the start state of $D$
A state of $D$ is accepting if it contains an accepting state of $N$

Renamed some terms from Palsberg/Hosking slide
NFA to DFA using subset construction: example

A = \{0, 1, 2, 4, 7\}
B = \{1, 2, 3, 4, 6, 7, 8\}
C = \{1, 2, 4, 5, 6, 7\}
D = \{1, 2, 4, 5, 6, 7, 9\}
E = \{1, 2, 4, 5, 6, 7, 10\}

Are NFAs more powerful than DFAs?
Fewer states and easier to construct!
But the transformation is not minimal.
**Theorem:** For each regular language that can be accepted by a DFA, there exists a DFA with a minimum number of states.

**Minimization approach:**
merge *equivalent* states.

States A and C are indistinguishable, so they can be merged!

After \(b^*a\) we always end up in state B.

DFA Minimization algorithm

> Create lower-triangular table DISTINCT, initially blank
> For every pair of states \((p, q)\):
>   — If \(p\) is final and \(q\) is not, or vice versa
>     - \(DISTINCT(p, q) = \varepsilon\)
> > Loop until no change for an iteration:
> >   — For every pair of states \((p, q)\) and each symbol \(\alpha\)
> >     - If \(DISTINCT(p, q)\) is blank and
> >       \(DISTINCT(\delta(p, \alpha), \delta(q, \alpha))\) is not blank
> >     - \(DISTINCT(p, q) = \alpha\)
> > > Combine all states that are not distinct

Distinguish final state from all others. Then take single steps to check what is distinguishable. The intuition:
- if one state is final and the other not, then they are clearly distinct
- otherwise, for every (state, state, symbol) tuple we see whether the \(\delta\) is in \(DISTINCT\)
0. initial state. 1. E is final, so different from others.
2. Only a “b” step from D leads to non-blank space.
3. B can make a “b” step to D, so differs from A and C.
4. A and C are indistinguishable. (An “a” takes both to B and “b” takes both to C.)
It is easy to see that this is in fact the minimal DFA for $(a | b)^* abb …$

Actually it is easy to see that this is the minimal DFA:
Start with the path abb. This gives us 4 states. Now add the missing arrows.
Any a transition brings us to state 1, since we must follow with bb.
Any b not in the path brings us back to state 0, since we must follow with abb.
A Generalized NFA is an NFA where transitions may have any RE as labels.

Conversion algorithm:

1. **Add a new start state and accept state** with $\varepsilon$-transitions to/from the old start/end states.
2. **Merge multiple transitions** between two states to a single RE choice transition.
3. **Add empty $\emptyset$-transitions** between states where missing.
4. **Iteratively "rip out" old states** and replace “dangling transitions” with appropriately labeled transitions between remaining states.
5. **STOP when all old states are gone** and only the new start and accept states remain.
1. Let $k$ be the number of states of $G$, $k \geq 2$
2. If $k=2$, then $RE$ is the label found between $q_s$ and $q_a$ (start and accept states of $G$)
3. While $k>2$, select $q_{rip} \neq q_s$ or $q_a$
   — $Q^* = Q - \{q_{rip}\}$
   — For any $q_i \in Q^* - \{q_{a}\}$ let $\delta^*(q_i,q_j) = R_1 R_2^* R_3 \cup R_4$ where:
     $R_1 = \delta^*(q_i,q_{rip})$, $R_2 = \delta^*(q_{rip},q_{rip})$, $R_2 = \delta^*(q_{rip},q_i)$, $R_4 = \delta^*(q_i,q_j)$
   — Replace $G$ by $G^*$
Add new start and accept states
This means “you can’t get there from here”
Delete an arbitrary state
Fix dangling transitions $s \rightarrow 1$ and $3 \rightarrow 1$
Don't forget to merge the existing transitions!

NB: The path from (3) to (1) merges the old path $bb^*a$ from (3)$\rightarrow$(0)$\rightarrow$(1) and the path $a$ from (3)$\rightarrow$(1).
Simplify the RE
Delete another state

**NB:** \(bb^*a|a = (bb^*|\varepsilon)a = b^*a\)
NB: $aa^*b|bb^*aa^*b = (\varepsilon|bb^*)aa^*b = b^*aa^*b$
Note that $b^*aa^*b = b^*a^*ab$

And so $b^*aa^*b (b^*aa^*b)^* b = (b^*a^*ab)^* b^*a^*abb$

It remains to be shown that $(b^*a^*ab)^* b^*a^* = (alb)^* \ldots$
\[ b^{*}a^{*}b \ (b^{*}a^{*}b)^{*} \ b = (a|b)^{*}abb \ ? \]

> **We can rewrite:**
  - \( b^{*}a^{*}b \ (b^{*}a^{*}b)^{*} \ b \)
  - \( b^{*}a^{*}ab \ (b^{*}a^{*}ab)^{*} \ b \)
  - \( (b^{*}a^{*}ab)^{*} \ b^{*}a^{*} \ abb \)

> **But does this hold?**
  - \( (b^{*}a^{*}ab)^{*} \ b^{*}a^{*} = (a|b)^{*} \)

We can show that the minimal DFAs for these REs are isomorphic …

Proof: Split any string in \((ab)^{*}\) by occurrences of ab. This will match \((Xab)^{*}X\), where \(X\) does not contain ab. \(X\) is clearly \(b^{*}a^{*}\). QED

Proof #2 by @grammarware: \( (b^{*}a^{*}ab)^{*}b^{*}a^{*} = (b^{*}a^{*}b)^{*}b^{*}a^{*} = b^{*}(a^{*}b^{*})a^{*} = b^{*}(b^{*}(a^{*}b^{*})^{*}la^{*})a^{*} = b^{*}(alb)^{*}a^{*} = (alb)^{*}a^{*} = (alb)^{*} \)
Roadmap

> Introduction
> Regular languages
> Finite automata recognizers
> From RE to DFAs and back again
> Limits of regular languages
Limits of regular languages

Not all languages are regular!

One cannot construct DFAs to recognize these languages:

- \[ L = \{ p^k q^k \} \]
- \[ L = \{ wcw^r \mid w \in \Sigma^*, w^r \text{ is } w \text{ reversed} \} \]

In general, DFAs cannot count!

However, one can construct DFAs for:

- Alternating 0’s and 1’s:
  \[ (\varepsilon \mid 1)(01)^+ (\varepsilon \mid 0) \]
- Sets of pairs of 0’s and 1’s:
  \[ (01 \mid 10)^+ \]
So, what is hard?

Certain language features can cause problems:

> Reserved words
  — PL/I had no reserved words
  — if then then then = else; else else = then

> Significant blanks
  — FORTRAN and Algol68 ignore blanks
  — do 10 i = 1,25
  — do 10 i = 1.25

> String constants
  — Special characters in strings
  — Newline, tab, quote, comment delimiter

> Finite limits
  — Some languages limit identifier lengths
  — Add state to count length
  — FORTRAN 66 — 6 characters(!)
How bad can it get?

Example due to Dr. P.K. Zadeck of IBM Corporation

Compiler needs context to distinguish variables from control constructs!
What you should know!

- What are the key responsibilities of a scanner?
- What is a formal language? What are operators over languages?
- What is a regular language?
- Why are regular languages interesting for defining scanners?
- What is the difference between a deterministic and a non-deterministic finite automaton?
- How can you generate a DFA recognizer from a regular expression?
- Why aren’t regular languages expressive enough for parsing?
Can you answer these questions?

- Why do compilers separate scanning from parsing?
- Why doesn’t NFA → DFA translation normally result in an exponential increase in the number of states?
- Why is it necessary to minimize states after translation a NFA to a DFA?
- How would you program a scanner for a language like FORTRAN?
Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

You are free to:

Share — copy and redistribute the material in any medium or format
Adapt — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

http://creativecommons.org/licenses/by-sa/4.0/