Scanning
(Lexical Analysis)

The Front End

- Maps stream of characters into words
- Basic unit of syntax
- Characters that form a word are its **lexeme**
- Its **part of speech** (or **syntactic category**) is called its **token type**
- Scanner discards white space & (often) comments

The Big Picture

- Language syntax is specified with **parts of speech**, not words
- Syntax checking matches parts of speech against a grammar

why study lexical analysis?

- We want to avoid writing scanners by hand

Goals:

- To simplify specification & implementation of scanners
- To understand the underlying techniques and technologies

Represent words as indices into a global table

Specifications written as "regular expressions"
Regular Expressions

Lexical patterns form a regular language

Regular expressions (REs) describe regular languages

Regular Expression (over alphabet Σ)
- ε is a RE denoting the set {ε}
- If a ∈ Σ, then a is a RE denoting {a}
- If x and y are REs denoting L(x) and L(y) then
  - x ∪ y is a RE denoting L(x) ∪ L(y)
  - x·y is a RE denoting L(x)L(y)
  - x* is a RE denoting L(x)*

Precedence is: closure, then concatenation, then alternation

Examples of Regular Expressions

Identifiers:
- Letter → ( { a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z } - | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z )
- Digit → ( { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 } - | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z )
- Identifier → ( Letter | Letter | Digit )

Numbers:
- Integer → ( Digit | Digit | Digit | Digit | Digit | Digit | Digit )
- Decimal → ( Digit | Digit | Digit | Digit | Digit | Digit )
- Real → ( Integer | Decimal | Integer | Decimal | Integer | Decimal )
- Complex → ( Real | Real | Real | Real | Real | Real )

Numbers can get much more complicated!

Example

Consider the problem of recognizing ILOC register names
- Register → ( { A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z } - | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 )
- Allows registers of arbitrary number
- Requires at least one digit

RE corresponds to a recognizer (or DFA)

Transitions on other inputs go to an error state, s_e

Set Operations

(Review)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Union of L and M</td>
<td>L ∪ M = { l ∈ L or m ∈ M }</td>
</tr>
<tr>
<td>Concatenation of L and M</td>
<td>L·M = { l·m</td>
</tr>
<tr>
<td>Kleene closure of L</td>
<td>L* = { l^n</td>
</tr>
<tr>
<td>Positive closure of L</td>
<td>L+ = { l^n</td>
</tr>
</tbody>
</table>

These definitions should be well known

Regular Expressions

(The point)

Regular expressions can be used to specify the words to be translated to parts of speech by a lexical analyzer

Using results from automata theory and theory of algorithms, we can automatically build recognizers from regular expressions

We study REs and associated theory to automate scanner construction!

Example

DFA operation
- Start in state S_0 & take transitions on each input character
- DFA accepts a word if \( S_e \) leaves it in a final state (S_2)

S_0

Recognizer for Register
- \{ A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z \}
- T takes it through S_0 and S_1 and accepts
- A takes it through S_0, S_1, and fails
- \{ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \} takes it straight to S_2
Example (continued)

To be useful, recognizer must turn into code

<table>
<thead>
<tr>
<th>Char ← next character</th>
<th>State ← $s_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>while (Char = EOF)</td>
<td>State ← $\delta$(State,Char)</td>
</tr>
<tr>
<td>Char ← next character</td>
<td>if (State is a final state) then report success else report failure</td>
</tr>
</tbody>
</table>

Skeleton recognizer

Table encoding RE

What if we need a tighter specification?

- Digit Digit* allows arbitrary numbers
  - Accepts 00000
  - Accepts 99999
  - What if we want to limit it to r$^9$ through r$^{31}$?

Write a tighter regular expression

- $\delta$ $\rightarrow$ $\epsilon$ \( (0|1|2) \ (\text{Digit} \ | \ 0) \ | \ (1|2|3|4|5|6|7|8|9) \ | \ (1|2|3|4) \)

- $\delta$ $\rightarrow$ $r$ $| \ (0|1|2|3|4|5|6|7|8|9) \ | \ (1|2|3|4) \)

- $\delta$ $\rightarrow$ $r$ $| \ (0|1|2|3|4|5|6|7|8|9) \ | \ (1|2|3|4) \)

Tighter register specification (continued)

Produces a more complex DFA
  - Has more states
  - Same cost per transition
  - Same basic implementation

Tighter register specification

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>$r$</th>
<th>0</th>
<th>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>$s_2$</td>
<td>$s_3$</td>
<td>$s_4$</td>
<td>$s_5$</td>
<td>$s_6$</td>
<td>$s_7$</td>
</tr>
</tbody>
</table>

Table encoding RE for the tighter register specification

Runs in the same skeleton recognizer

Constructing a Scanner - Quick Review

- The scanner is the first stage in the front end
- Specifications can be expressed using regular expressions
- Build tables and code from a DFA
Goal

- We will show how to construct a finite state automaton to recognize any RE
- Overview:
  → Direct construction of a nondeterministic finite automaton (NFA) to recognize a given RE
  → Requires ε-transitions to combine regular subexpressions
  → Construct a deterministic finite automaton (DFA) to simulate the NFA
  → Use a set-of-states construction
  → Minimize the number of states
  → Hopcroft state minimization algorithm
  → Generate the scanner code
  → Additional specifications needed for details

More Regular Expressions

- All strings of 1s and 0s ending in a 1
  \[(0 | 1)^* \]
- All strings over lowercase letters where the vowels (a, e, i, o, u) occur exactly once, in ascending order

\[\text{Cons} \rightarrow \text{bcldifghijklmnopqrstuvwxyz}\]
\[\text{Cond} \rightarrow \text{Cond} \lor \text{Cond} \lor \text{Cond} \lor \text{Cond} \lor \text{Cond} \]

- All strings of 1s and 0s that do not contain three 0s in a row:
  \[(e | 01 | 001) \lor (e | 0 | 00)\]

Non-deterministic Finite Automata

- An NFA accepts a string x iff ∃ a path through the transition graph from q0 to a final state such that the edge labels spell x
- Transitions on ε consume no input
- To "run" the NFA, start in q0 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 ε-transitions

More Regular Expressions

- All strings of 1s and 0s ending in a 1
  \[(0 | 1)^* \]
- All strings over lowercase letters where the vowels (a, e, i, o, u) occur exactly once, in ascending order
  \[\text{Cons} \rightarrow \text{bcldifghijklmnopqrstuvwxyz}\]
  \[\text{Cond} \rightarrow \text{Cond} \lor \text{Cond} \lor \text{Cond} \lor \text{Cond} \lor \text{Cond} \]
  \[\text{All strings of 1s and 0s that do not contain three 0s in a row:}\]
  \[(e | 01 | 001) \lor (e | 0 | 00)\]

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 \((a | b | \varepsilon)bbb\)?

This is a little different

- \(S_2\) has a transition on \(\varepsilon\)
- \(S_2\) has two transitions on \(b\)
  This is a non-deterministic finite automaton (NFA)

Relationship between NFAs and DFAs

- DFA is a special case of an NFA
- DFA has no ε 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, Flex, JLex 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!

RE $\rightarrow$ NFA using Thompson's Construction

Key idea
- NFA pattern for each symbol and each operator
- Each NFA has a single start and accept state
- Join them with $\epsilon$ moves in precedence order

Example of Thompson's Construction

Let's try $a \ (b \ | \ e) \ ^*$
1. $a$, $b$, $\delta$
2. $b$, $\epsilon$
3. $(b \ | \ e) \ ^*$

NFA $\rightarrow$ DFA with Subset Construction

Need to build a simulation of the NFA

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

The algorithm:
- Start state derived from $s_0$ of the NFA
- Take its $\epsilon$-closure $S_0 \ = \ \epsilon$-closure($s_0$)
- Take the image of $S_0$, move($S_0$, $a$) for each $a \in \Sigma$, and take its $\epsilon$-closure
- Iterate until no more states are added

Sounds more complex than it is.
NFA → DFA with Subset Construction

The algorithm:

\[ s_0 \rightarrow s \text{-closure}(q_0) \]
add \( s \) to \( S \)

while ( \( S \) is still changing )
for each \( s \in S \)
\[ s \leftarrow s \text{-closure}(move(s,a)) \]
if \( \{ s \}, s \in S \) then
add \( s \) to \( S \) as \( s \)
\[ T[s,a] = s \]

Let’s think about why this works.

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
  - Solving sets of simultaneous set equations

We will see more fixed-point computations

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 really part of scanner construction)
- All pairs, all paths problem
- Union together paths from \( s_0 \) to a final state

DFA Minimization

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

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

Two states are equivalent if and only if:
- \( \forall a \in \Sigma, \text{transitions on } a \text{ lead to equivalent states} \) (DFA)
- \( \alpha \)-transitions to distinct sets \( \Rightarrow \) states must be in distinct sets

DFA Minimization

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

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

Splitting a set ("partitioning a set by \( \gamma \)"")
- Assume \( q_s \neq q_s \), and \( \delta(q_s, \gamma) = q_s \) & \( \delta(q_s, \gamma) = q_s \).
- If \( q_s \neq q_s \) are not in the same set, then \( S \) must be split
  \( \rightarrow q_s \) has transition on \( a \), \( q_s \) does not \( \Rightarrow \) \( \gamma \) splits \( S \)

DFA Minimization

Then, apply the minimization algorithm

<table>
<thead>
<tr>
<th>Current Partition</th>
<th>Split on</th>
<th>State</th>
<th>Splitted</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_1 ) ( q_s, q_s, (P_2) )</td>
<td>( a )</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>

To produce the minimal DFA

We observed that a human would design a simpler automaton than Thompson's construction & the subset construction did.

Minimizing that DFA produces the one that a human would design!

DFA Minimization

Why does this work?
- Partition \( P \subseteq 2^Q \)
- Start of with 2 subsets of \( Q \) \( (F) \) and \( (Q-F) \)
- While loop takes \( P \rightarrow P_{s} \) by splitting 1 or more sets
  \( P_{s} \) is at least one step closer to the partition with \( |Q| \) sets
  Maximum of \( |Q| \) splits
  Note that
  - Partitions are never combined

Abbreviated Register Specification

Start with a regular expression

\( r0 | r1 | r2 | r3 | r4 | r5 | r6 | r7 | r8 | r9 \)

Note: "|" is left associative
Abbreviated Register Specification

Thompson's construction produces

The Cycle of Constructions

Abbreviated Register Specification

The DFA minimization algorithm builds

This looks like what a skilled compiler writer would do!

The Cycle of Constructions

Abbreviated Register Specification

The subset construction builds

This is a DFA, but it has a lot of states ...

The Cycle of Constructions

Limits of Regular Languages

Advantages of Regular Expressions
- Simple & powerful notation for specifying patterns
- Automatic construction of fast recognizers
- Many kinds of syntax can be specified with REs

Example — an expression grammar
- Term \( \rightarrow [0-9A-Z][0-9A-Z] \mid \{2, \} \)
- Op \( \rightarrow [\_\_\_\_\_\_\_]\) Term
- Expr \( \rightarrow (\text{Term Op}) \text{Term} \)

Of course, this would generate a DFA ...

If REs are so useful ...

Why not use them for everything?

What can be so hard?

Poor language design can complicate scanning
- Reserved words are important
  - if then then = else; else else = then
  - (Fortran & Algol68)
- Significant blanks
  - do 10 i = 1,25
  - do 10 i = 1,25

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

Limited identifier "length"
  - (Fortran 66 & PL/I)