Prelude
- What happened on August 14, 2003?
  - 2003 North American Blackout
  - 50 million people without power
- How did it happen?
  - Cascading power line failure
  - More power on a line causes it to heat up
  - Heating causes the metal conductor to expand
  - The power line sags, hits a tree, fails
- Why didn’t power co.’s respond more quickly?
  - No alarm sounded on early failures
  - Software bug!

Big picture
- Front end responsibilities
  - Check that the input program is legal
  - Check syntax and semantics
  - Emit meaningful error messages
  - Build IR of the code for the rest of the compiler

Front end design
- Two part design
  - Scanner
    - Reads in characters
    - Classifies sequences into words or tokens
  - Parser
    - Checks sequence of tokens against grammar
    - Creates a representation of the program (AST)

Scanner and parser
- Why separate scanner and parser?
  - Simplifies the implementation
  - Parsing is fundamentally harder
    - Word classification is easier – make it fast
    - Speed up parsing by working with tokens

Lexical analysis
- The input is just a sequence of characters.
  - Example:
    ```
    if (i == j)
    z = 0;
    else
    z = 1;
    ```
- More accurately, the input is:
  ```
  \tif (i \eq j)\\n  \txt z = 0;\\n  \else\\n  \txt z = 1;
  ```
- Goal: Partition input string into substrings
  - And classify them according to their role
Scanner

- Responsibilities
  - Read in characters
  - Produce a stream of tokens
  

```
key.id <op><id> op=<op><id> ... 
```

- Token has a type and a value

Hand-coded scanner

- Explicit test for each token
  - Read in a character at a time
  - Example: recognizing keyword "if"

```c
if (c != 'i')
  error();
else {
  c = readchar();
  if (c != 'f')
    error();
  else
    return IF_TOKEN;
}
```

- What about other tokens?
  - Example: "if" is a keyword, "if0" is an identifier

```c
if (c != 'i') {
  other tokens...}
else {
  c = readchar();
  if (c != 'f') {
    other tokens...}
  else {
    c = readchar();
    if (c not alpha-numeric) {
      putback(c);
      return IF_TOKEN;
    }
    while (c alpha-numeric) {
      build identifier
    }
```

Hand-coded scanner

- Problems:
  - Many different kinds of tokens
    - Fixed strings (keywords)
    - Special character sequences (operators)
    - Tokens defined by rules (identifiers, numbers)
  - Tokens overlap
    - "if" and "if0" example
    - "=" and "=="
  - Coding this by hand is too painful!
    - Getting it right is a serious concern

Outline

Problems we need to solve:
- Scanner specification language
  - How to describe parts of the input language
- The scanning mechanism
  - How to break input string into tokens
- Scanner generator
  - How to translate from (1) to (2)
- Ambiguities
  - The need for lookahead
Problem 1: Describing the scanner

- We want a high-level language \( D \) that
  - Describes lexical components, and
  - Maps them to tokens (determines type)
- But doesn’t describe the scanner algorithm itself!

- Part 3 is important
  - Allows focusing on what, not on how
  - Therefore, \( D \) is sometimes called a specification language, not a programming language
- Part 2 is easy, so let’s focus on Parts 1 and 3

Token examples

- Keyword
  - Exact sequence of characters
- Identifier
  - Sequence of letters or numbers, starting with a letter
- Number
  - Sequence of digits
- Whitespace
  - Sequence of space, tab, carriage-return

Specifying tokens

- Many ways to specify them
- Regular expressions are the most popular
  - REs are a way to specify sets of strings
    - Examples:
      - \( 'a' \) – denotes the set \{“a”\}
      - \( 'a'|'b' \) – denotes the set \{“a”, “b”\}
      - \( 'a''b' \) – denotes the set \{“ab”\}
      - \( 'a''b'* \) – denotes the set \{“a”, “ab”, “abb”, “abbb”, … \}
- Why regular expressions?
  - Easy to understand
  - Strong underlying theory
  - Very efficient implementation

Formal languages

- **Def.** a language is a set of strings
  - Alphabet \( \Sigma \) : the character set
  - Language is a set of strings over alphabet
  - Each regular expression denotes a language
    - If \( A \) is a regular expression, then \( L(A) \) is the set of strings denoted by \( A \)
  - Examples: given \( \Sigma = \{ 'a', 'b' \} \)
    - \( A = 'a' \) \( L(A) = \{ "a" \} \)
    - \( A = 'a'|'b' \) \( L(A) = \{ "a", "b" \} \)
    - \( A = 'a''b' \) \( L(A) = \{ "ab" \} \)
    - \( A = 'a''b'* \) \( L(A) = \{ "a", "ab", "abb", "abbb", … \} \)

Building REs

- Regular expressions over \( \Sigma \)
- Atomic REs
  - \( \varepsilon \) is an RE denoting empty set
  - If \( a \) is in \( \Sigma \), then \( a \) is an RE for \( \{a\} \)
- Compound REs
  - If \( x \) and \( y \) are REs then:
    - \( xy \) is an RE for \( L(x)L(y) \)
    - \( x|y \) is an RE for \( L(x) \cup L(y) \)
    - \( x^* \) is an RE for \( L(x)^* \)

Using regular expressions

- Concatenation: build up words
- Kleene closure: repetition
- Alternation: collect sets of words
- Back to our language problem…
Keywords
- Exact strings: "if" or "for" or "while"
  - Singleton sets
  - Built up using concatenation
    \[ L('i' 'f') = \{ "if" \} \]
    \[ L('f' 'o' 'r') = \{ "for" \} \]
  - We can abbreviate 'i' 'f' as 'if'
    'if' | 'for' | 'else' | ...

Integers
- Integer: non-empty sequence of digits
  \[ \text{digit} = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' \]
  \[ \text{integer} = \text{digit} \text{ digit}^* \]
  - Another abbreviation:
    \[ A+ \cup AA^* \]

Identifiers
- Identifier: string of letters or numbers starting with a letter
  \[ \text{letter} = 'a' | 'b' | ... | 'z' | 'A' | ... | 'Z' \]
  \[ \text{identifier} = \text{letter} ( \text{letter} | \text{digit} )^* \]
  - Is this the same as (letter|digit)+?
  - How about (letter*|digit*)?

Other examples
- Numbers
  \[ \text{int} = ( '+' | '-' | \epsilon ) \text{ digit}^+ \]
  \[ \text{decimal} = \text{int} \ \text{ digit}^+ \]
  \[ \text{real} = (\text{int} | \text{decimal}) ( 'E' ( '+' | '-' | \epsilon ) \text{ digit}^+ ) \]
  - What about IP addresses?
    \[ \text{ip} = \text{digit}^+ . \text{ digit}^+ . \text{ digit}^+ . \text{ digit}^+ \]
    - Is this right?
    - Can we be more precise?

Outline
Problems we need to solve:
- Scanner specification language
  - How to describe parts of the input language
- The scanning mechanism
  - How to break input string into tokens
- Scanner generator
  - How to translate from (1) to (2)
- Ambiguities
  - The need for lookahead

Overview of scanning
- How do we recognize strings in the language?
  Every RE has an equivalent finite state automaton that recognizes its language
  (Actually, more than one)
  - Idea: scanner simulates the automaton
    - Read characters
    - Transition automaton
    - Return a token if automaton accepts the string
Finite Automata

- Regular expressions = specification
- Finite automata = implementation

- A finite automaton consists of
  - An input alphabet \( \Sigma \)
  - A set of states \( S \)
  - A start state \( q_0 \)
  - A set of accepting states \( F \subseteq S \)
  - A set of transitions \( s \rightarrow a \rightarrow s' \)

Finite Automata

- Transition
  - Is read
    - In state \( s_1 \) on input “a” go to state \( s_2 \)
  - If end of input
    - If in accepting state => accept
    - Otherwise => reject

Finite Automata State Graphs

- A state
  - The start state
  - An accepting state
  - A transition

A Simple Example

- A finite automaton that accepts only “1”

Another Simple Example

- FA accepts any number of 1’s followed by a single 0
  - Alphabet: \{0,1\}

- Check that “1110” is accepted but “110…” is not

And Another Example

- Alphabet \{0,1\}
- What language does this recognize?
“Realistic” example

• Consider the problem of recognizing machine register names

\[
\text{Register} \rightarrow r \ (0|1|2| \ldots |9) \ (0|1|2| \ldots |9)^* 
\]


Implementation

• Finite automaton
  • States, characters
  • State transition δ uniquely determines next state
  • Automaton is deterministic

• Next character function
  • Reads next character into buffer
  • (May compute character class by fast table lookup)

• Transitions from state to state
  • Implement δ as a table
  • Access table using current state and character

Example

Turning the recognizer into code

\[
\delta \\
| r | 0,1,2,3,4,5,6,7,8,9 | \text{All others} \\
| s_0 | s_1 | s_2 | s_3 | s_4 | s_5 | s_6 | s_7 | s_8 | s_9 \\
| s_1 | s_2 | s_3 | s_4 | s_5 | s_6 | s_7 | s_8 | s_9 | s_0 \\
| s_2 | s_3 | s_4 | s_5 | s_6 | s_7 | s_8 | s_9 | s_0 | s_1 \\
| s_3 | s_4 | s_5 | s_6 | s_7 | s_8 | s_9 | s_0 | s_1 | s_2 \\

Char ← next character
State ← s_0
while (Char ≠ EOF)
  State ← δ(State, Char)
  perform specified action
  Char ← next character
if (State is a final state)
  then report success
else report failure

Skeleton recognizer

Example

Adding actions

\[
\delta \\
| r | 0,1,2,3,4,5,6,7,8,9 | \text{All others} \\
| s_0 | s_1 | s_2 | s_3 | s_4 | s_5 | s_6 | s_7 | s_8 | s_9 \\
| s_1 | s_2 | s_3 | s_4 | s_5 | s_6 | s_7 | s_8 | s_9 | s_0 \\
| s_2 | s_3 | s_4 | s_5 | s_6 | s_7 | s_8 | s_9 | s_0 | s_1 \\
| s_3 | s_4 | s_5 | s_6 | s_7 | s_8 | s_9 | s_0 | s_1 | s_2 \\
| s_4 | s_5 | s_6 | s_7 | s_8 | s_9 | s_0 | s_1 | s_2 | s_3 \\
| s_5 | s_6 | s_7 | s_8 | s_9 | s_0 | s_1 | s_2 | s_3 | s_4 \\

Char ← next character
State ← s_0
while (Char ≠ EOF)
  State ← δ(State, Char)
  perform specified action
  Char ← next character
if (State is a final state)
  then report success
else report failure

Skeleton recognizer

What if we need a tighter specification?

• Digit Digit* allows arbitrary numbers
  • Accepts 00000
  • Accepts 99999
  • What if we want to limit it to 0 through 31?

• Write a tighter regular expression
  • Register → r (0|1|2) (Digit | ε) | (4|5|6|7|8|9) | (3|30|31)
  • Register → r (0|1|2) (Digit | ε) | (4|5|6|7|8|9) | (3|30|31)

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

Tighter register specification

• The DFA for

\[
\text{Register} \rightarrow r \ (0|1|2) (Digit | ε) | (4|5|6|7|8|9) | (3|30|31) 
\]
Tight register specification

<table>
<thead>
<tr>
<th>( \delta )</th>
<th>0</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_2 )</td>
<td>( s_3 )</td>
<td>( s_4 )</td>
<td>( s_5 )</td>
<td>( s_6 )</td>
</tr>
<tr>
<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_6 )</td>
</tr>
<tr>
<td>( s_2 )</td>
<td>( s_3 )</td>
<td>( s_4 )</td>
<td>( s_5 )</td>
<td>( s_6 )</td>
<td>( s_6 )</td>
<td>( s_6 )</td>
</tr>
<tr>
<td>( s_3 )</td>
<td>( s_4 )</td>
<td>( s_5 )</td>
<td>( s_6 )</td>
<td>( s_6 )</td>
<td>( s_6 )</td>
<td>( s_6 )</td>
</tr>
<tr>
<td>( s_4 )</td>
<td>( s_5 )</td>
<td>( s_6 )</td>
<td>( s_6 )</td>
<td>( s_6 )</td>
<td>( s_6 )</td>
<td>( s_6 )</td>
</tr>
<tr>
<td>( s_5 )</td>
<td>( s_6 )</td>
<td>( s_6 )</td>
<td>( s_6 )</td>
<td>( s_6 )</td>
<td>( s_6 )</td>
<td>( s_6 )</td>
</tr>
<tr>
<td>( s_6 )</td>
<td>( s_6 )</td>
<td>( s_6 )</td>
<td>( s_6 )</td>
<td>( s_6 )</td>
<td>( s_6 )</td>
<td>( s_6 )</td>
</tr>
</tbody>
</table>

Table encoding RE for the tighter register specification

REs and DFAs

- Key idea:
  - Every regular expression has an equivalent DFA that accepts only strings in the language

- Problem:
  - How do we construct the DFA for an arbitrary regular expression?
  - Not always easy

Example

- What is the RE for \( a(a|\varepsilon)b \)?
- Need \( \varepsilon \) moves
- Transition \( A \) to \( B \) without consuming input!

Another example

- Remember this DFA?
- We can simplify it as follows:

A different kind of automaton

- Accepts the same language
  - Actually, it’s easier to understand!

- What’s different about it?
  - Two different transitions on ‘0’
  - This is a non-deterministic finite automaton

DFAs and NFAs

- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No \( \varepsilon \)-moves

- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have \( \varepsilon \)-moves
Execution of Finite Automata

- A DFA can take only one path through the state graph
  - Completely determined by input

- NFAs can choose
  - Whether to make \( \epsilon \)-moves
  - Which of multiple transitions for a single input to take

Acceptance of NFAs

- An NFA can get into multiple states

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 \( \epsilon \) consume no input)

- To "run" the NFA, start in \( s_0 \) and \textit{guess} the right transition at each step
  - Always guess correctly
  - If some sequence of correct guesses accepts \( x \) then accept

Why do we care about NFAs?

- Simpler, smaller than DFAs

- More importantly:
  - Need them to support all RE capabilities
  - Systematic conversion from REs to NFAs
  - Need \( \epsilon \) transitions to connect RE parts

- Problem: how to implement NFAs?
  - How do we guess the right transition?
  - Multiple states: what about memory usage?

Relationship between NFAs and DFAs

- DFA is a special case of an NFA
  - DFA has no \( \epsilon \)-transitions
  - DFA’s transition function is single-valued
  - Same rules will work

- DFA can be simulated with an NFA (obvious)

- 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

Automatic 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!
Automatic scanner construction

**RE → NFA (Thompson’s construction)**
- Build an NFA for each term
- Combine them with $\epsilon$-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
- Take the union of all paths from $s_0$ to an accepting state

Next time...

- RE -to- NFA -to- DFA -to- scanner
- Algorithms (yikes!)
- Programming assignment 1