From characters to tokens

- What is a token?
  - Roughly – a "word" in the source language
  - Identifiers
  - Values
  - Language keywords
  - Really - anything that should appear in the input to syntax analysis

- Technically
  - Usually a pair of (kind, value)
Example Tokens

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>x, y, z, foo, bar</td>
</tr>
<tr>
<td>NUM</td>
<td>42</td>
</tr>
<tr>
<td>FLOATNUM</td>
<td>3.141592654</td>
</tr>
<tr>
<td>STRING</td>
<td>“so long, and thanks for all the fish”</td>
</tr>
<tr>
<td>LPAREN</td>
<td>(</td>
</tr>
<tr>
<td>RPAREN</td>
<td>)</td>
</tr>
<tr>
<td>IF</td>
<td>#</td>
</tr>
</tbody>
</table>

Strings with special handling

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comments</td>
<td>/* Ceci n’est pas un commentaire */</td>
</tr>
<tr>
<td>Preprocessor directives</td>
<td>#include&lt;foo.h&gt;</td>
</tr>
<tr>
<td>Macros</td>
<td>#define THE_ANSWER 42</td>
</tr>
<tr>
<td>White spaces</td>
<td>\ in</td>
</tr>
</tbody>
</table>

From characters to tokens

\[
x = b^2 - 4ac
\]

Token Stream

\[
<ID,"x">, <EQ>, <ID,"b">, <MUL_T>, <ID,"b">, <MINUS>, <INT,4>, <MUL_T>, <ID,"a">, <MUL_T>, <ID,"c">
\]

Errors in lexical analysis

\[
\text{set} \quad \text{illegal token}
\]

\[
\text{set} \quad \text{illegal token}
\]

\[
\text{set} \quad <\text{ID,"pi"}, <\text{EQ}, <\text{ID,"oranges3"}>
\]
How can we define tokens?

- Keywords – easy!
  - if, then, else, for, while, ...
- Identifiers?
- Numerical Values?
- Strings?
- Characterize unbounded sets of values using a bounded description?

Regular Expressions

<table>
<thead>
<tr>
<th>Basic Patterns</th>
<th>Matching</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>The character x</td>
</tr>
<tr>
<td>.</td>
<td>Any character, usually except a new line</td>
</tr>
<tr>
<td>[xyz]</td>
<td>Any of the characters x,y,z</td>
</tr>
<tr>
<td>R?</td>
<td>An R or nothing (=optionally an R)</td>
</tr>
<tr>
<td>R*</td>
<td>Zero or more occurrences of R</td>
</tr>
<tr>
<td>R+</td>
<td>One or more occurrences of R</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Repetition Operators</th>
<th>Composition Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>R?</td>
<td>R1R2</td>
</tr>
<tr>
<td>R*</td>
<td>RsRr</td>
</tr>
<tr>
<td>R+</td>
<td>(R)</td>
</tr>
<tr>
<td>Grouping</td>
<td>R itself</td>
</tr>
</tbody>
</table>

Examples

- ab*|cd? =
- (a|b)* =
- (0|1|2|3|4|5|6|7|8|9)* =

Escape characters

- What is the expression for one or more + symbols?
  - (++) won’t work
  - (+++) will
- backslash \ before an operator turns it to standard character
  - \#, \\
  - \+, ...
Shorthands

- Use names for expressions
  - letter = a | b | ... | z | A | B | ... | Z
  - letter_ = letter | _
  - digit = 0 | 1 | 2 | ... | 9
  - id = letter_ (letter_ | digit)*
- Use hyphen to denote a range
  - letter = a-z | A-Z
  - digit = 0-9

Examples

- digit = 0-9
- digits = digit+
- number = digits (E | .digits (E | e (E|+|) digits ))
- if = if
- then = then
- relop = < | > | <= | >= | = | <>

Ambiguity

- if = if
- id = letter_ (letter_ | digit)*

"if" is a valid word in the language of identifiers... so what should it be?
- How about the identifier "iffy"?

Solution

- Always find longest matching token
- Break ties using order of definitions... first definition wins (=> list rules for keywords before identifiers)

Creating a lexical analyzer

- Input
  - List of token definitions (pattern name, regex)
  - String to be analyzed
- Output
  - List of tokens

- How do we build an analyzer?
Character classification

```c
#define is_end_of_input(ch) ((ch) == '\0');
#define is_uc_letter(ch) ('A' <= (ch) && (ch) <= 'Z')
#define is_lc_letter(ch) ('a' <= (ch) && (ch) <= 'z')
#define is_letter(ch) (is_uc_letter(ch) || is_lc_letter(ch))
#define is_digit(ch) ('0' <= (ch) && (ch) <= '9')
```

Main reading routine

```c
void get_next_token() {
    do {
        char c = getchar();
        switch(c) {
            case is_letter(c) : return recognize_identifier(c);
            case is_digit(c) : return recognize_number(c);
            ... 
        } while (c != EOF);
}
```

But we have a much better way!

- Generate a lexical analyzer automatically from token definitions
- Main idea
  - Use finite-state automata to match regular expressions

Reminder: Finite-State Automaton

- Deterministic automaton
- $M = (\Sigma, Q, \delta, q_0, F)$
  - $\Sigma$ - alphabet
  - $Q$ – finite set of states
  - $q_0 \in Q$ – initial state
  - $F \subseteq Q$ – final states
  - $\delta: Q \times \Sigma \rightarrow Q$ - transition function
Reminder: Finite-State Automaton

- Non-Deterministic automaton
- \( M = (\Sigma, Q, \delta, q_0, F) \)
  - \( \Sigma \) - alphabet
  - \( Q \) - finite set of state
  - \( q_0 \in Q \) - initial state
  - \( F \subseteq Q \) - final states
  - \( \delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q \) - transition function

- Possible \( \varepsilon \)-transitions
- For a word \( w \), \( M \) can reach a number of states or get stuck. If some state reached is final, \( M \) accepts \( w \).

From regular expressions to NFA

- Step 1: assign expression names and obtain pure regular expressions \( R_1 \ldots R_m \)
- Step 2: construct an NFA \( M_i \) for each regular expression \( R_i \)
- Step 3: combine all \( M_i \) into a single NFA

- Ambiguity resolution: prefer longest accepting word

Basic constructs

- \( R = \varepsilon \)  
- \( R = a \)  
- \( R = \emptyset \)  

Composition

- \( R = R_1 | R_2 \)  
- \( R = R_1 R_2 \)
Repetition

\[ R = R_1^* \]

What now?

- Naïve approach: try each automaton separately
- Given a word \( w \):
  - Try \( M_1(w) \)
  - Try \( M_2(w) \)
  - ...  
  - Try \( M_n(w) \)
- Requires resetting after every attempt

Combine automata

Ambiguity resolution

- Recall...
- Longest word
- Tie-breaker based on order of rules when words have same length

- Recipe
  - Turn NFA to DFA
  - Run until stuck, remember last accepting state, this is the token to be returned
**Examples**

- aba: gets stuck after aba in state 12, backs up to state (5 8 11) pattern is a*b+, token is ab abba: stops after second b in (6 8), token is abb because it comes first in spec

**Good News**

- All of this construction is done automatically for you by common tools
- lex is your friend
  - Automatically generates a lexical analyzer from declaration file

**Lex declarations file**

```c
#include "lex.h"

Token_Type Token;
int line_number = 1;

while (1) {
  switch (state) {
  case 0: { 
    {digit}+ {return INTEGER;}
  }
  case 1: { 
    {identifier} {return IDENTIFIER;}
  }
  case 2: { 
    {whitespace} { /* ignore whitespace */}
  }
  case 3: { 
    {identifier} {return IDENTIFIER;}
  }
  case 4: { 
    {identifier} {return IDENTIFIER;}
  }
  case 5: { 
    {identifier} {return IDENTIFIER;}
  }
  case 6: { 
    {identifier} {return IDENTIFIER;}
  }
  case 7: { 
    {identifier} {return IDENTIFIER;}
  }
  case 8: { 
    {identifier} {return IDENTIFIER;}
  }
  case 9: { 
    {identifier} {return IDENTIFIER;}
  }
  case 10: { 
    {identifier} {return IDENTIFIER;}
  }
  case 11: { 
    {identifier} {return IDENTIFIER;}
  }
  case 12: { 
    {identifier} {return IDENTIFIER;}
  }
  case 13: { 
    {identifier} {return IDENTIFIER;}
  }
  }
  goto 1;
}
```

```c
void start_lex(void) {};
void get_next_token(void) {};
```
Summary

- Lexical analyzer
  - Turns character stream into token stream
  - Tokens defined using regular expressions
  - Regular expressions \(\rightarrow\) NFA \(\rightarrow\) DFA construction for identifying tokens
  - Automated constructions of lexical analyzer using `lex`

Coming up next time

- Syntax analysis

### NFA vs. DFA

<table>
<thead>
<tr>
<th>Automation</th>
<th>SPACE</th>
<th>TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>NFA</td>
<td>(O(</td>
<td>r</td>
</tr>
<tr>
<td>DFA</td>
<td>(O(2^{</td>
<td>r</td>
</tr>
</tbody>
</table>

\[(a|b)^*a(a|b)(a|b)...(a|b)\]  
\[\text{\(n\) times}\]