From Characters to Tokens

- What is a token?
  - Roughly – a “word” in the source language
    - Identifiers, Values, Keywords
  - Really – anything that should appear in the input to syntax analysis as a single unit
- Technically
  - Usually a pair of ⟨kind, value⟩
### Example Tokens

<table>
<thead>
<tr>
<th>Kind</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>x, y, z0, 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>&quot;so long, and thanks for all the fish&quot;</td>
</tr>
<tr>
<td>LPAR</td>
<td>(</td>
</tr>
<tr>
<td>RPAR</td>
<td>)</td>
</tr>
<tr>
<td>IF</td>
<td>if</td>
</tr>
<tr>
<td>MINUS</td>
<td>~</td>
</tr>
</tbody>
</table>

### Strings with Special Handling

<table>
<thead>
<tr>
<th>Kind</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>\t \n \t</td>
</tr>
</tbody>
</table>

### Some basic terminology

- **Lexeme** (aka symbol) – a series of characters separated from the rest of the program according to a convention (space, semicolon, word boundary, ...)

- **Pattern** – a rule specifying a set of strings. Example: “an **identifier is a string that starts with a letter, followed by letters and digits**”

- **Token** – a pair of (pattern, attributes)

### From Characters to Tokens

```
let
a = b*b - 4*a*c
```

Token Stream:

```
(ID,"a") : EQ : (ID,"b") : MULT : (ID,"b") : MINUS : (INT,4) : MULT : (ID,"a") : MULT : (ID,"c")
```
Errors in Lexical Analysis

pi = 3.141562

Illegal token

pi = 3oranges

Illegal token

pi = oranges3

⟨ID,”pi”⟩, ⟨EQ⟩, ⟨ID,”oranges3”⟩

Error Handling

• Many errors cannot be identified at this stage
  • Example: should "fi" be "if"? Or is it a routine name?
    – We will discover this later in the analysis
    – At this point, we just create an identifier token

• Sometimes the lexeme does not match any pattern
  – Easiest fix: skip characters until the beginning of a legitimate lexeme
  – Alternatives: eliminate/add/replace one letter, replace order of two adjacent letters, etc.

• Pro: allow the compilation to continue
• Con: errors that spread all over

How can we define tokens?

• Keywords — easy!
  ‣ if, then, else, for, while, ...

• Identifiers?

• Numerical Values?

• Strings?
  • "there are infinitely many", ...

• Characterize unbounded sets of values using a bounded description?

Regular Expressions over \( \Sigma \)

<table>
<thead>
<tr>
<th>Basic Patterns</th>
<th>Matching</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>A single letter 'x' from the alphabet ( \Sigma )</td>
</tr>
<tr>
<td>([xyz])</td>
<td>Any character (usually excluding a new-line)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Repetition Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>R?</td>
</tr>
<tr>
<td>R*</td>
</tr>
<tr>
<td>R+</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Composition Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>R|R|R||R|</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Grouping</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
</tr>
</tbody>
</table>
Examples

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

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

- if = if
- then = then
- relop = $< | > | <= | >= | = | <>$
- digit = 0-9
- digits = digit+
- number = digits ( \. digits ( e (\+|-) digits )\)?
Ambiguity

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

"if" matches both the pattern for reserved words and the pattern for 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);
}
• Generate a lexical analyzer **automatically** from token definitions

• Main idea
  – Use finite-state automata to match regular expressions

---

**Overview**

• Construct a **nondeterministic finite-state automaton** (NFA) from regular expression (automatically)

• Determinize the NFA into a **deterministic finite-state automaton** (DFA)

• DFA can be directly used to identify tokens

---

**Reminder: Finite-State Automaton**

[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
  - \( \delta : Q \times \Sigma \rightarrow Q \) – transition function

[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
  - \( \delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q \) – transition function

- Allows \( \epsilon \)-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 $R_1...R_m$
- 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

$$R = \varepsilon$$

$$R = a$$

$$R = \phi$$

Composition

$$R = R_1 \mid R_2$$

$$R = R_1R_2$$

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 “rewinding” after every attempt

Combine automata

![Combined automata diagram]

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

Corresponding DFA

![Corresponding DFA diagram]
Example Inputs

- **abaa**: 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 \((6,8)\), pattern is \(abb\) because it comes first in spec

Summary of Construction

- ✓ Describe tokens as regular expressions
- ✓ Regular expressions turned into a DFA
- ✓ Lexical analyzer simulates the run of an automata with the given transition table on any input string

Good News

- Construction is done automatically by common tools
- **Flex** is your friend
  - Automatically generates a lexical analyzer from declaration file
- Advantages: short declaration file, easily checked, easily modified and maintained

Flex declarations file

```c
 %{ #include <math.h> int line_number = 1; %}

 WS [ \t ]
 LETTER [a-zA-Z]
 DIGIT [0-9]
 ID (LETTER)(LETTER|DIGIT)*

%%

{DIGIT}+ { printf("number: %d\n", atoi(yytext)); return 1; }
{ID} { return 2; }
{WS} { /* ignore */ }
\n{ line_number++; /* and ignore */ }
{ return -1; /* ERROR */ }

%%
int main() { return yylex(); }
```
Summary

- **Lexical analyzer**
  - ✓ Turns character stream into token stream
  - ✓ Tokens defined using regular expressions
  - ✓ Regex → NFA → DFA construction for identifying tokens
  - ✓ Automated constructions of lexical analyzer using **Flex**

**Coming Up**

[Image of a ladder leading to a sign that reads "Syntax Analysis"]