The Compiler

- Lexical Analysis: Break into tokens (think words, punctuation)
- Syntax Analysis: Parse phrase structure (think document, paragraphs, sentences)
- Semantic Analysis: Calculate meaning

Lexical Analysis

- Lexical Analysis: Breaks stream of ASCII characters (source) into tokens
- Token: Sequence of characters treated as a unit
- Each token has a token type:

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>Token Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo.x</td>
<td>1.0</td>
<td>NUM</td>
</tr>
<tr>
<td>listCount</td>
<td>2.1</td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td>10.45</td>
<td>IF</td>
</tr>
<tr>
<td>SEMI</td>
<td></td>
<td>ASSIGN</td>
</tr>
<tr>
<td>LPAREN</td>
<td></td>
<td>RPAREN</td>
</tr>
</tbody>
</table>

- Some tokens have associated semantic information:

- White space and comments often discarded.
Lexical Analysis Example

\[ x = ( y + 4.0 ); \]

Implementing a Lexer

The first phase of a compiler is called the **Lexical Analyzer** or **Lexer**.

**Implementation Options:**

1. Write Lexer from scratch.
2. Use Lexical Analyzer Generator.

How do we describe the source language tokens to the Lexer Generator?

Regular Expressions

**Some Definitions:**

- Alphabet - a collection of symbols (ASCII is an alphabet)
- String - finite sequence of symbols taken from finite alphabet
- Language - set of strings
- Examples:
  - ML Language - set of all strings representing correct ML programs (INFINITE).
  - Language of ML keywords - set of all strings which are ML keywords (FINITE).
  - Language of ML tokens - set of all strings which map to ML tokens (INFINITE).

**Regular Expressions (REs):**

- REs specify languages (possibly infinite) using finite descriptions.
- REs are good for specifying the language of a language's tokens.
Regular Expressions

**Construction**

**Base Cases:**
- Symbol: for each symbol $a$ in alphabet, $a$ is a RE denoting the language containing only the string $a$.
- Epsilon ($\epsilon$): a language containing only the empty string

**Inductive Cases: (assume $M$ and $N$ are regular expressions)**
- Alternation ($M|N$): a RE denoting strings in $M$ or $N$.
  $$a \mid b \rightarrow \{a, b\}$$
- Concatenation ($MN$): a RE denoting strings in $M$ concatenated with those in $N$.
  $$\{a \mid b\} (a \mid c) \rightarrow \{aa, ac, ba, bc\}$$
- Kleen closure ($M^*$): a RE denoting strings formed by concatenating zero or more strings, all of which are in $M$.
  $$(a \mid b)^* \rightarrow \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}$$

---

Regular Expression Examples

---

Finite Automata

A finite automaton has:
- Finite number of states
- Set of edges, each directed from one state to another, labeled with a single symbol
- A start state
- One or more final states

[Diagram of finite automata and lexer generator]
Finite Automata

- Language recognized by FA is set of strings it accepts.
- Accept orReject
  - Start in start state
  - An edge is traversed for each symbol in input string.
  - After $n$ transitions for $n$-symbol string, if in final state, ACCEPT
  - If in non-final state or no valid edge was found during traversal, REJECT

Classes of Finite Automata

**Deterministic Finite Automata (DFA)**
- Edges leaving a node are uniquely labeled.

**Non-deterministic Finite Automata (NFA)**
- Two or more edges leaving a node can be identically labeled.
- An edge can be labeled with $\epsilon$.

**Implementing Lexer:**
- RE $\rightarrow$ NFA $\rightarrow$ DFA
RE to NFA Rules

\[ a: S \xrightarrow{a} F \]
\[ \varepsilon: S \xrightarrow{\varepsilon} F \]
\[ \varepsilon: S \xrightarrow{\varepsilon} M \]
\[ \varepsilon: S \xrightarrow{\varepsilon} N \]

\[ \text{MN:} \quad S \xrightarrow{\varepsilon} M \xrightarrow{F/S} N \xrightarrow{F} \]

\[ M^*: \quad S \xrightarrow{\varepsilon} M \xrightarrow{\varepsilon} F \]

RE to NFA Example
NFA to DFA Conversion

Idea: Avoid guessing by trying all possibilities simultaneously.

Basic Functions

- $\text{edge}(s, a) =$ All NFA states reachable from state $s$ by traversing label $a$.
- $\text{closure}(S) =$ All reachable NFA states from $s \in S$ by traversing label $\epsilon$.
  \[
  \text{closure}(S) = S \cup (\cup_{s \in S} \text{edge}(s, \epsilon))
  \]
- $\text{DFA}_{\text{edge}}(D, a) =$ All reachable NFA states from $s \in D$ by traversing $a$ and $\epsilon$.
  \[
  \text{DFA}_{\text{edge}}(D, a) = \text{closure} \ (\cup_{s \in D} \text{edge}(s, a))
  \]

NFA to DFA Example

DFA Representation

Coding the DFA: The Transition Matrix and Finality Array
The Longest Token

**Lexer must find longest matching token.**

```plaintext
ifz8       ID not IF, ID
iff        IFF not IF, ID
```

- Save most recent final state and position in stream
- Update when new final state found

---

**Other Useful Techniques**

- Read Chapters 1 and 2.
- **Equivalent states:**
  - Eliminate redundant states, smaller FA.
  - Do Exercise 2.6 (hand in optional).
- **FA → RE:**
  - Useful to confirm correct RE → FA.
  - GNFAs!
  - See: *Introduction to the Theory of Computation* by Michael Sipser

---

**The Compiler**

- **Lexical Analysis:** Break into tokens (think words, punctuation)
- **Syntax Analysis:** Parse phrase structure (think document, paragraphs, sentences)
- **Semantic Analysis:** Calculate meaning
The first phase of a compiler is called the **Lexical Analyzer** or **Lexer**.

**Implementation Options:**

1. Write Lexer from scratch.
2. Use Lexical Analyzer Generator.

- **ml-lex** is a lexical analyzer generator for ML.
- **lex** and **flex** are lexical analyzer generators for C.

---

**ML Lex**

- Input to **ml-lex** is a set of rules specifying a lexical analyzer.
- Output from **ml-lex** is a lexical analyzer in ML.
- A rule consists of a pattern and an *action*:
  - Pattern is a regular expression.
  - Action is a fragment of ordinary ML code. (Typically returns a token type to calling function.)
- Examples:
  
  ```ml
  if -> (print("Found token IF");)
  [0-9]* -> (print("Found token NUM");)
  ```
- General Idea: When prefix of input matches a pattern, the action is executed.

---

**Lexical Specification**

- Lexical specification consists of 3 parts:

  ```plaintext
  User Declarations
  ML-LEX Definitions
  Rules
  ```

- User Declarations:
  - User can define various values that are available to the *action* fragments.
  - Two values **must** be defined in this section:
    - **type lexresult**
      - type of the value returned by each rule action.
    - **fun eof()**
      - called by lexer when end of input stream reached.
Lexical Specification

- Lexical specification consists of 3 parts:
  - User Declarations
  - ML-LEX Definitions
  - Rules

- ML-LEX Definitions:
  - User can define regular expression abbreviations:
    \[\text{DIGITS}=[0-9]+;\]
    \[\text{LETTER}=[a-zA-Z];\]
  - Define \textit{start states} to permit multiple lexers to run together.
    \[\% \text{STATE1 STATE2 STATE3};\]

Lexical Specification

- Lexical specification consists of 3 parts:
  - User Declarations
  - ML-LEX Definitions
  - Rules

- Rules:
  \[\text{<start_state_list> regular_expression} \rightarrow (\text{action_code});\]

- A rule consists of a pattern and an action:
  - \textit{Pattern} is a regular expression.
  - \textit{Action} is a fragment of ordinary ML code. (Typically returns a token type to calling function.)

- Rules may be prefixed with a list of start states (defined in ML-LEX Definition).

Rule Patterns

<table>
<thead>
<tr>
<th>symbol</th>
<th>matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>individual character “a” (not for reserved chars ?,*,+,{,})</td>
</tr>
<tr>
<td>{</td>
<td>reserved character {</td>
</tr>
<tr>
<td>abc</td>
<td>a</td>
</tr>
<tr>
<td>[a-zA-Z]</td>
<td>lowercase and capital letters</td>
</tr>
<tr>
<td>.</td>
<td>any character except new line</td>
</tr>
<tr>
<td>\n</td>
<td>newline</td>
</tr>
<tr>
<td>\t</td>
<td>tab</td>
</tr>
<tr>
<td>“abc?”</td>
<td>abc? taken literally (reserved chars as well)</td>
</tr>
<tr>
<td>{LETTER}</td>
<td>Use abbreviation LETTER defined in ML-LEX Definitions</td>
</tr>
<tr>
<td>a*</td>
<td>0 or more a’s</td>
</tr>
<tr>
<td>a+</td>
<td>1 or more a’s</td>
</tr>
<tr>
<td>a?</td>
<td>0 or 1 a</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>

\[\text{if|iff \rightarrow (print("Found token IF or IFF"));}\]
\[\text{[0-9]+ \rightarrow (print("Found token NUM"));}\]
Rule Actions

- Actions can use various values defined in User Declarations section.
- Two values always available:
  
  ```
  type lexresult
  - type of the value returned by each rule action.
  fun eof()
  - called by lexer when end of input stream reached.
  ```
- Several special variables also available to action fragments.
  - yytext - input substring matched by regular expression.
  - yypos - file position of beginning of matched string.
  - continue() - recursively calls lexing engine.

Start States

- **Start states** permit multiple lexical analyzers to run together.
- Rules prefixed with a start state is matched only when lexer is in that state.
- States are entered with YYBEGIN.
- Example:
  ```
  %
  %# COMMENT
  %
  <INITIAL> if -> {print("Token IF")};
  <INITIAL> [a-z]+ => {print("Token ID")};
  <INITIAL> "*" => {YYBEGIN COMMENT; continue();}
  <COMMENT> "*" => {YYBEGIN INITIAL; continue();}
  <COMMENT> 
  ```

Rule Matching and Start States

- `<start_state_list> regular_expression -> {action_code};`
- Regular expression matched only if lexer is in one of the start states in start state list.
- If no start state list specified, the rule matches in all states.
- Lexer begins in predefined start state: INITIAL

If multiple rules match in current start state, use Rule Disambiguation.
Rule Disambiguation

- **Longest match** - longest initial substring of input that matches regular expression is taken as next token.

  ```
  if8 matches ID("if8"), not IF() and NUM(8).
  ```

- **Rule priority** - for a particular substring which matches more than one regular expression with equal length, choose first regular expression in rules section.

  If we want if to match IF(), not ID("if"), put keyword regular expression before identifier regular expression.

---

Example

```plaintext
(* -. - m l -. - *)

type lexresult = string
fun eof() = (print("End-of-file\n"), *EOP*)

%%
INT=[1-9][0-9]*;
%%

# COMMENT;
%%

<INITIAL>/*

<INITIAL>/*

<INITIAL>"\n"

<INITIAL>-if

<INITIAL>-then

<INITIAL>-[INT]

<INITIAL>-" "\n"\t"

<INITIAL>-.*

<INITIAL>.

== (YyBEGIN COMMENT; continue());

== (YyBEGIN INITIAL; continue());

== (continue());

== (print("Token IF\n"),"IF");

== (print("Token THEN\n"),"THEN");

== (print("Token INT\"\n", "\n");"INT");

== (continue());

== (print("ERR: " yytext " \n\n\n") ; *ERR*) ;
```

---

Example in Action

```plaintext
% cat x.txt
if 999 then 0999
/* This is a comment 099 if */
if 12 then 12

% m1
- CM. m1s();
   (....)
val it = () : unit
- MLexser.tokenset("x.txt");

Token IF
Token INT(999)
Token THEN
ERR: ‘0’.
Token INT(999)
Token IF
Token INT(12)
Token THEN
Token INT(12)
End-of-file
val it = () : unit
```