The Compiler

Lexical Analysis

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

Lexical Analysis Example

```
x = ( y + 4.0 );
```

The Compiler

Lexical Analysis

- Lexical Analysis: Break 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>Token</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>foo, x, listCount</td>
</tr>
<tr>
<td>REAL</td>
<td>10.45, 3.14, -2.1</td>
</tr>
<tr>
<td>SEMI</td>
<td>;</td>
</tr>
<tr>
<td>LPAREN</td>
<td>(</td>
</tr>
<tr>
<td>NUM</td>
<td>50, -100</td>
</tr>
<tr>
<td>IF</td>
<td>if</td>
</tr>
<tr>
<td>ASSIGN</td>
<td>=</td>
</tr>
<tr>
<td>RPAREN</td>
<td>)</td>
</tr>
</tbody>
</table>

- Some tokens have associated semantic information:

<table>
<thead>
<tr>
<th>Value</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>foo</td>
<td>ID(foo)</td>
</tr>
<tr>
<td>-100</td>
<td>NUM(-100)</td>
</tr>
<tr>
<td>10.45</td>
<td>REAL(10.45)</td>
</tr>
</tbody>
</table>

- White space and comments often discarded.
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

**Construction**

**Base Cases:**
- Symbol: for each symbol \( a \) in alphabet, \( a \) is a RE denoting 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

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.

They are also good at specifying a language that can specify the language of a language’s tokens.
Finite Automata

Finite Automata

Finite Automaton: a computational model of a machine with limited memory

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

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 Example

NFA to DFA Conversion

Idea: Avoid guessing by trying all possibilities simultaneously.

Basic Functions

- \textit{edge}(s, a) = \text{All NFA states reachable from state } s \text{ by traversing label } a.
- \textit{closure}(S) = \text{All reachable NFA states from } s \in S \text{ by traversing label } \varepsilon.
  \[
  \text{closure}(S) = S \cup \bigcup_{s \in S} \text{edge}(s, \varepsilon)
  \]
- \textit{DF Aedge}(D, a) = \text{All reachable NFA states from } s \in D \text{ by traversing } a \text{ and } \varepsilon \text{ edges.}
  \[
  \text{DF Aedge}(D, a) = \text{closure} \left( \bigcup_{s \in D} \text{edge}(s, a) \right)
  \]
The Longest Token

Lexer must find longest matching token.

ifz8 ID not IF, ID
tiff 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

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:
  ```
  if => (print("Pound token IF"));
  [0-9]+ => (print("Pound token NUM"));
  ```
- General Idea: When prefix of input matches a pattern, the action is executed.

Lexical Specification

- Lexical specification consists of 3 parts:
  - User Declarations
  ```
  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.

ML Lex, Lex, Flex, ...

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.

RE's

Source

Lex Token Stream

Lexer

Finite Automata

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

- Lexical specification consists of 3 parts:
  - User Declarations
    ```
    %
    ML-LEX Definitions
    %
    Rules
    ```
  - ML-Lex Definitions:
    - User can define regular expression abbreviations:
      ```
      DIGITS=[0-9]+;
      LETTER=[a-zA-Z];
      ```
    - Define `start states` to permit multiple lexers to run together.
      ```
      %s STATE1 STATE2 STATE3;
      ```

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>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>
</tbody>
</table>

- `<start.state.list> regular.expression => (action.code);`
- `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.)
- Rules may be prefixed with a list of `start states` (defined in ML-LEX Definition).

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:

```c
<INITIAL> if => (print("Token IF"));
<INITIAL> [a-z]+ => (print("Token ID"));
<INITIAL> "(*)" => (YYBEGIN COMMENT; continue());
<INITIAL> "*" => (YYBEGIN INITIAL; continue());
<INITIAL> "\n". => (continue());
```

Rule Disambiguation

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

  if6 matches ID(‘if6’), not IF() and NUM(6).

- **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

```c
(* \*\* \n \*)
type lexresult = string
fun eof() = (print("End-of-file\n"); "EOF")

<INITIAL>/\*\* => (YYBEGIN COMMENT; continue());
<INITIAL>/\* => (YYBEGIN INITIAL; continue());
<INITIAL>\n => (continue());
<INITIAL>if => (print("Token IF\n"); "IF");
<INITIAL>then => (print("Token THEN\n"); "THEN");
<INITIAL>[INT] => (print("Token INT(\"\yytext\"\")\n"); "INT");
<INITIAL>\n\t => (continue());
<INITIAL>. => (print("ERR: \"\yytext \n\","ERR");
```
% cat x.txt
if 999 then 0999
/* This is a comment 099 if */
if 12 then 12

% sml
Standard ML of New Jersey, Version 109.33, November 21, 1997 [CM; ...]
- CM.make();
[......]
val it = () : unit
- MyLexer.tokenize("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