# Topic 2: Lexing and Flexing

**COS 320**

**Compiling Techniques**

Princeton University
Spring 2017

Prof. David August

## The Compiler

### 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*:

  | ID     | foo, x, listCount | REAL 10.45, 3.14, –2.1 | SEMI ; | LPAREN ( |
  | NUM 50, –100 | IF if | ASSIGN = | RPAREN |

- Some tokens have associated semantic information:
  - `foo ID(foo)`
  - `–100 NUM(–100)`
  - `10.45 REAL(10.45)`

- White space and comments often discarded.

### Lexical Analysis Example

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

### Syntax Analysis

- **Syntax Analysis**: Parse phrase structure (think document, paragraphs, sentences)

### Semantic Analysis

- **Semantic Analysis**: Calculate meaning

### Intermediate Representation

- Source Program
- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- IR Code Generation

### IR Optimization

- Target Code Generation
- Target Code Optimization

### Target Program

```plaintext
for(i=0; i<20; i++) {
    printf("%d\n", i);
}
```

```plaintext
i = 0
L6: CALL(printf, "%d\n", i)
    i = i + 1
    if(i < 20) GOTO L6
```

```plaintext
.L00: strings="%d\n"
    addl r37 = 0, r0
    addl r36 = @toff(.L00), gp
. L6: br.call.spkt Many b0 = printf$
    adds r37 = 1, r37
    cmpq gp p6, p7 = 9, r37
    (p6) br.cond. optk .L6
```
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?

Using another language of course!

Yeah, but how do we describe the tokens in that language?

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 ($M \mid N$): 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 $M$.
  \[ (a \mid b)^* \rightarrow \{ \epsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots \} \]

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

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

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

- Language recognized by FA is set of strings it accepts.
- Accept or Reject
  - 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

Finite Automata Examples

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 ε.

Implementing Lexer:
- RE → NFA → DFA
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{DFA}edge(D, a) = \text{All reachable NFA states from } s \in D \text{ by traversing } a \text{ and } \varepsilon \text{ edges}.
  \[
  \text{DFA}edge(D, a) = \text{closure} \left( \bigcup_{s \in D} \text{DFA}edge(s, a) \right)
  \]
The Longest Token

Lexer must find longest matching token.

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 $\rightarrow$ RE:
- Useful to confirm correct RE $\rightarrow$ FA.
- GNFAs!
- See: *Introduction to the Theory of Computation* by Michael Sipser

Coding the DFA: The Transition Matrix and Finality Array
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:
  ```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:
  - User Declarations
    ```ml
    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:
      ```ml
      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:
    ```
    DIGITS=[0-9]+;
    LETTER=[a-zA-Z];
    ```
  - Define start states to permit multiple lexers to run together.
    `%% 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>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>

if|iff => (print("Found token IF or IFF");
[0-9]+ => (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> "\n". => (continue());
```

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

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

```%

INT=[1-9][0-9]*;
%
$% COMMENT;
%
<INITIAL>"*/" => (YYBEGIN COMMENT; continue());
<COMMENT>"*/" => (YYBEGIN INITIAL; continue());
<COMMENT>"\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" => (continue());
<INITIAL>. => (print("ERR: " yytext " .\n";"ERR");
```
Example in Action

```plaintext
% 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
- MLexer.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
```