Topic 2: Lexing and Flexing

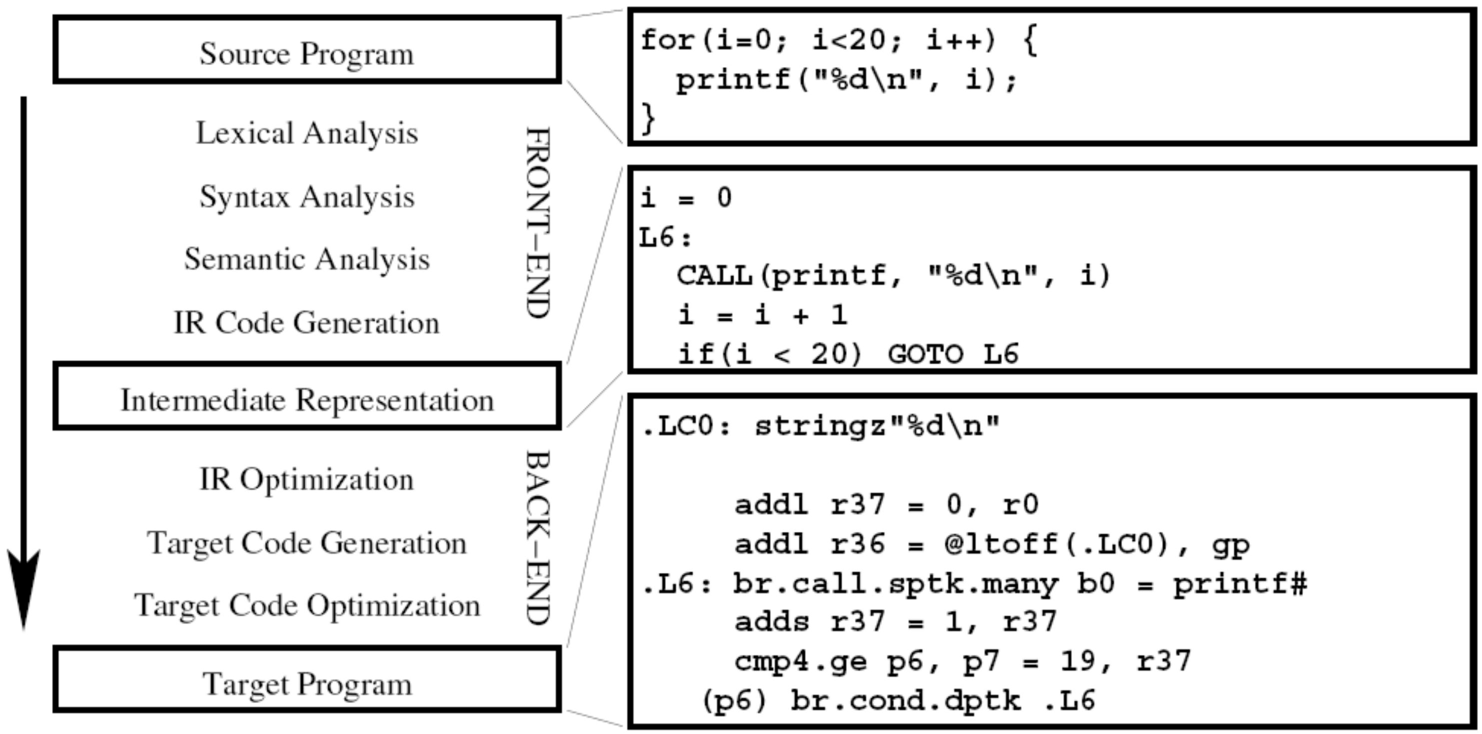
COS 320

Compiling Techniques

Princeton University
Spring 2018

Prof. David August

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*:

ID	`foo`, `x`, `listCount`	NUM	50, -100
$REAL$	10.45, 3.14, -2.1	IF	`if`
$SEMI$	;	$ASSIGN$	=
$LPAREN$	(	$RPAREN$	)

- 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

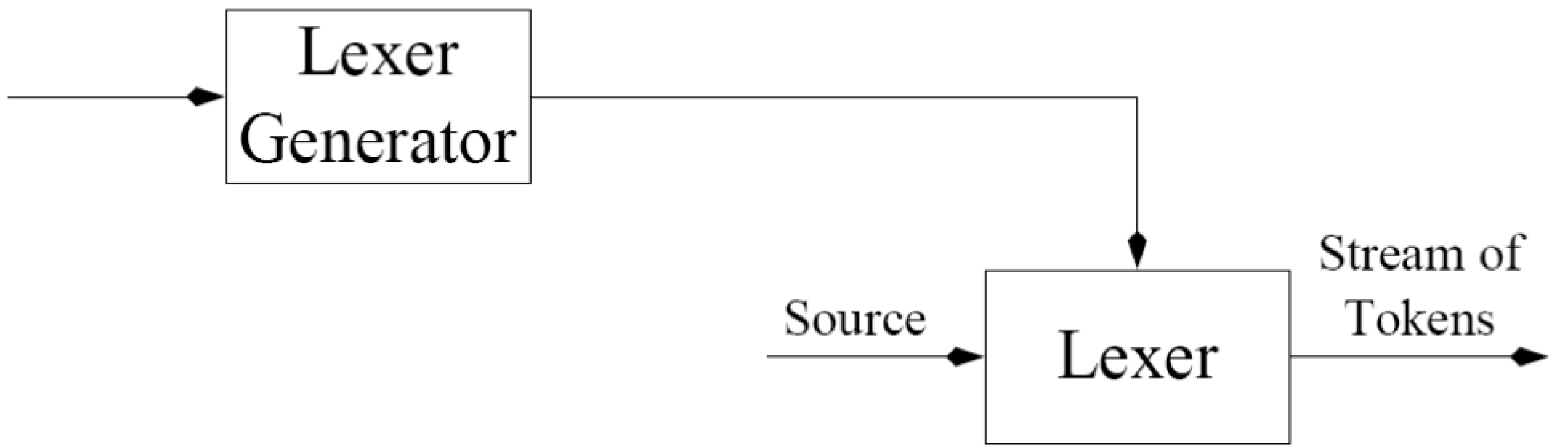
```
x = ( y + 4.0 );
```

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?

Using another language of course!

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

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.

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

Regular Expressions

Construction

Base Cases:

- Symbol: for each symbol a in alphabet, a is a RE denoting language containing only the string a.
- Epsilon (ϵ): 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`.

$$\texttt{a | b} \rightarrow \{\texttt{a, b}\}$$

- Concatenation (`MN`): a RE denoting strings in `M` concatenated with those in `N`.

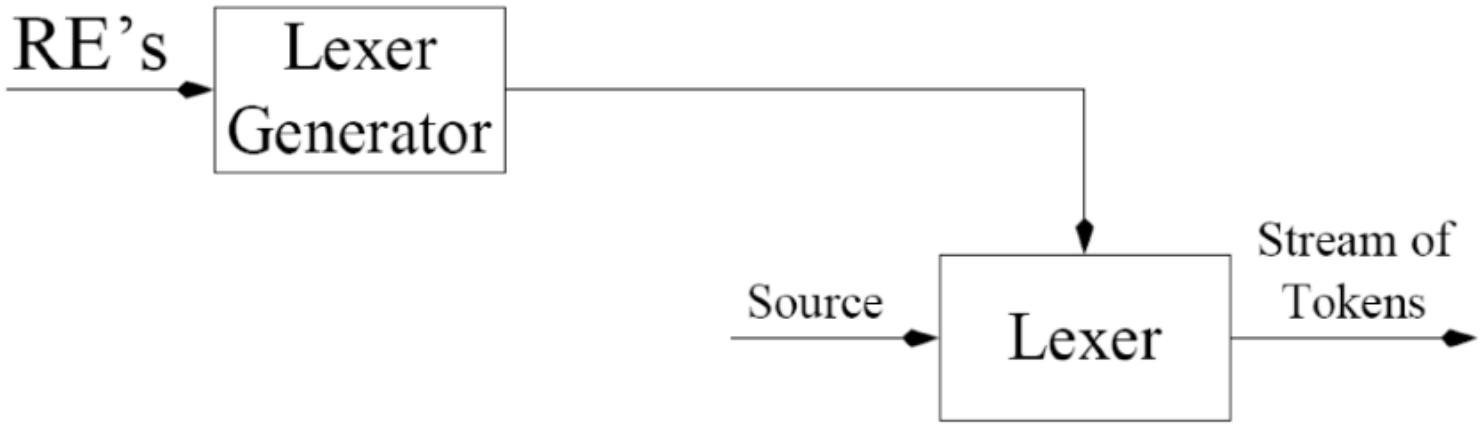
$$\texttt{(a | b)(a | c)} \rightarrow \{\texttt{aa, ac, ba, bc}\}$$

- Kleen closure (`M*`): a RE denoting strings formed by concatenating zero or more strings, all of which are in `M`.

$$\texttt{(a | b)*} \rightarrow \{\epsilon\texttt{, a, b, aa, ab, ba, bb, aaa, aab, ...}\}$$

Regular Expression Examples

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

Finite Automata

- 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)

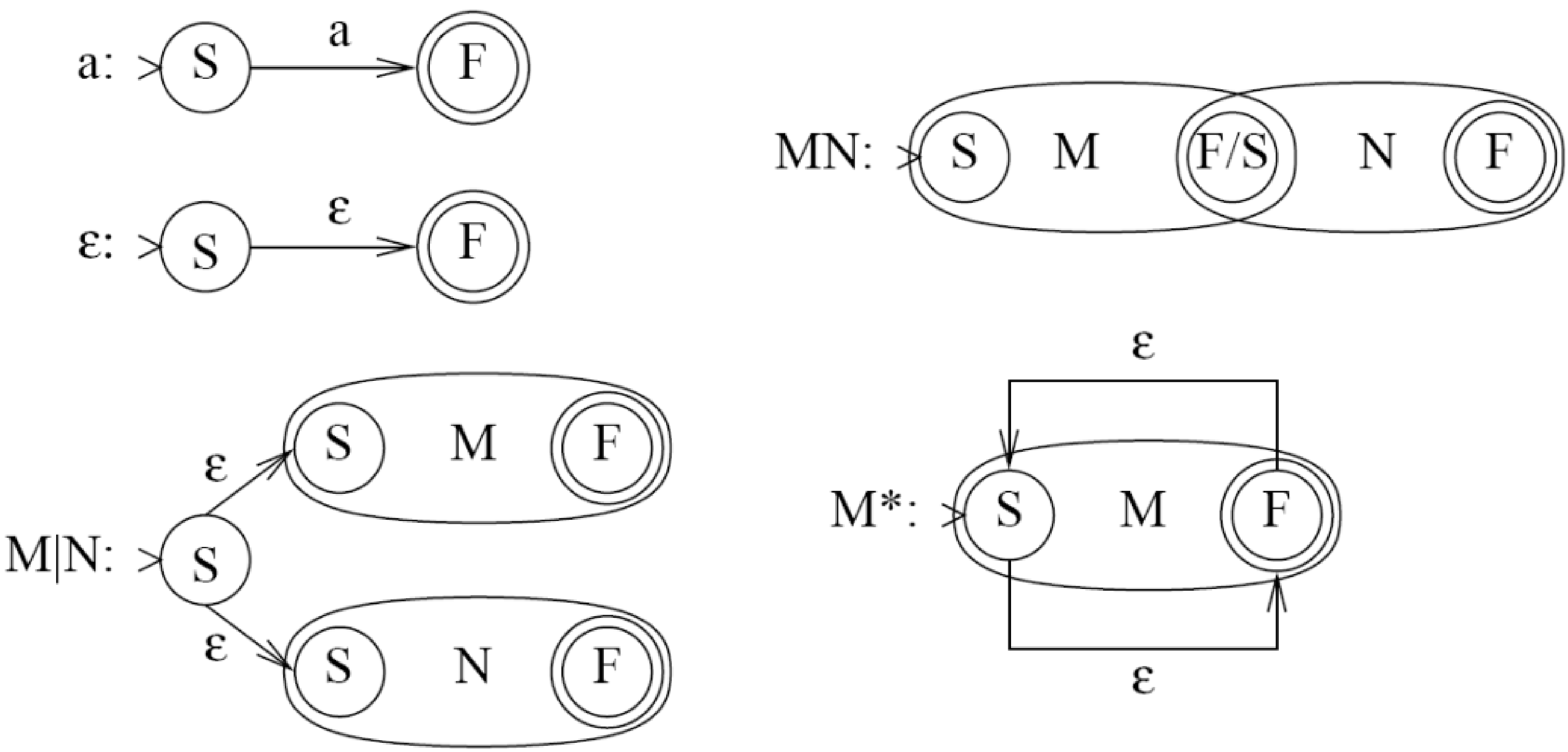
- Two or more edges leaving a node can be identically labeled.
- An edge can be labeled with ϵ.

Implementing Lexer:

- RE $\rightarrow$ NFA $\rightarrow$ DFA

NFA Example

RE to NFA Rules



RE to NFA Example

NFA to DFA Conversion

Idea: Avoid guessing by trying all possibilities simultaneously.

Basic Functions

- $edge(s, a)$ = All NFA states reachable from state ns by traversing label a.
- $closure(S)$ = All reachable NFA states from $s \in S$ by traversing label ϵ.
$$closure(S) = S \cup (\cup_{s \in S} edge(s, \epsilon))$$
- $DFAedge(D, a)$ = All reachable NFA states from $s \in D$ by traversing a and ϵ edges.
$$DFAedge(D, a) = closure\left(\cup_{s \in D} edge(s, a)\right)$$

NFA to DFA Example

DFA Representation

Coding the DFA: The Transition Matrix and Finality Array

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.

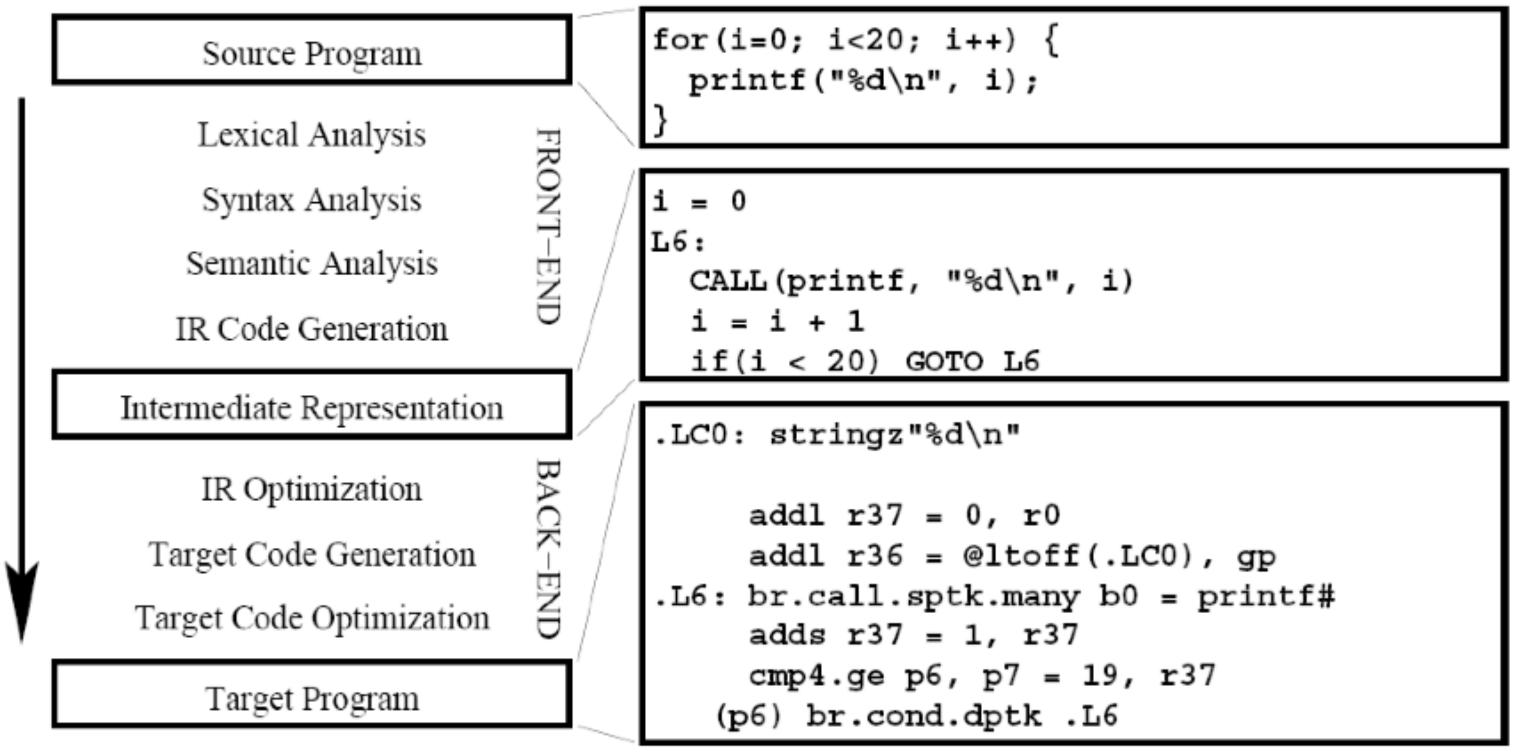
Equivalent states:

- 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

The Compiler



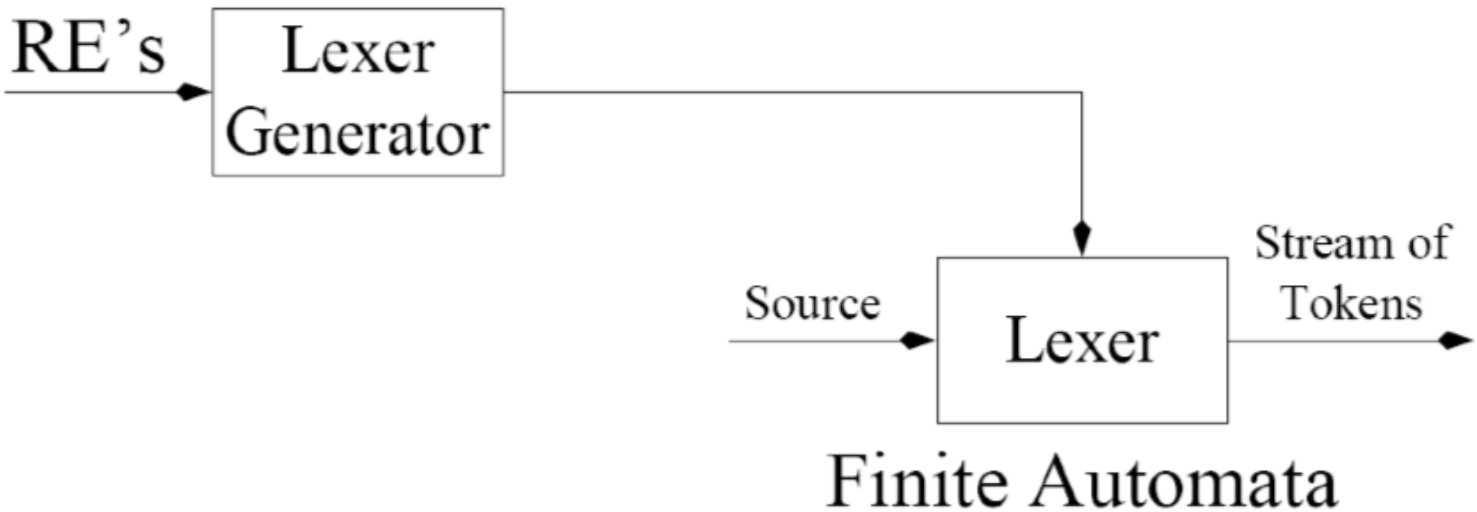
- Lexical Analysis: Break into tokens (think words, punctuation)
- Syntax Analysis: Parse phrase structure (think document, paragraphs, sentences)
- Semantic Analysis: Calculate meaning

ML Lex, 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.



- **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:

```
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-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:

```
DIGITS=[0-9]+;
LETTER=[a-zA-Z];
```

  - Define *start states* to permit multiple lexers to run together.

```
%s STATE1 STATE2 STATE3;
```

Lexical Specification

- Lexical specification consists of 3 parts:

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

- Rules:

```
<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 Patterns

<table>
<tr><th>symbol</th><th>matches</th></tr>
<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 | b | c</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|b</td><td>a or b</td></tr>
</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:

```
%%
%s 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]*;

%s 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"|"\t" => (continue());
<INITIAL>.             => (print("ERR: '" ^ yytext ^ "'.\n");"ERR");
```

Example in Action

```
% 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
```

From Assignment

The below is a very simple example of a flex input file.

```
%{
/* Headers; this part will be copied into lex.yy.c */
#include <stdio.h>
#include <stdlib.h>
void mycode();
%}

 /* Definitions */
letter      [A-Za-z]
digit       [0-9]

%%

 /* Rules (pattern / action) */
({letter})({letter}|{digit})*    printf("WORD\n");
({digit})+                       { printf("NUMBER\N"); mycode(); }

%%

/* Optional user codes; this part will be copied into lex.yy.c */
void mycode() {
  printf("lex example\n");
}
```