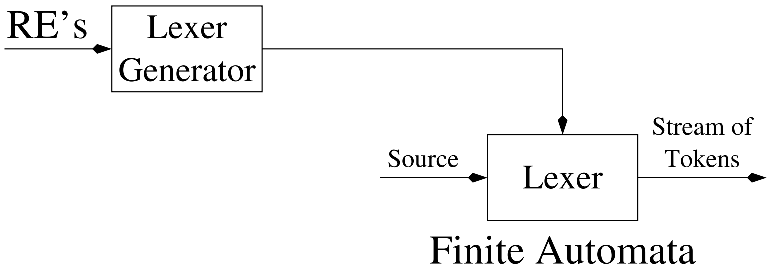
Lexer Implementation

Implementation Options:

1. Write Lexer from scratch. (Last time)
2. Use Lexical Analyzer Generator. (Today)



- **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 multiple lexers to work together. Each lexer is given a unique name.

```
%s LEX1 LEX2 LEX3;
```

Lexical Specification

- Lexical specification consists of 3 parts:

  User Declarations
  %%
  ML-LEX Definitions
  %%
  Rules

- Rules:

```
<lexer_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 the list of lexers that are allowed to use this rule (among those defined in the 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()` - used to recursively call the lexing engine.

Using Multiple Lexers

- Rules prefixed with a lexer name are matched only when that lexer is executing.
- If the rule is not prefixed, it can be matched by any lexer.
- We specify the lexer to use with `YYBEGIN`.
- The primary lexer is called `INITIAL`.
- 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());
```

Example

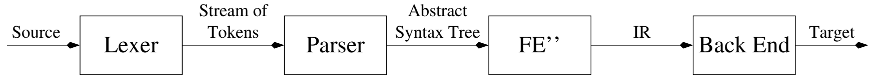
```
(* -*- ml -*- *)
type lexresult = string
fun eof() = (print("End-of-file\n"); "EOF")

%%

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");
```


Syntax Analysis

Front End:

- Lexical Analysis - Break source into *tokens*.
- Syntax Analysis - Parse phrase structure.
- Semantic Analysis - Calculate meaning.

Our Compiler:



Parser Functions:

- Verify that token stream is valid.
- If it is not valid, report syntax error and recover.
- Build Abstract Syntax Tree (AST).

Rule Disambiguation

When multiple rules match, we disambiguate as follows.

- *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.

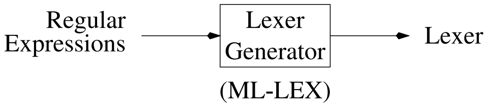
  If we want `if` to match `IF()`, not `ID(``if'')`, put keyword regular expression before identifier regular expression.

Example

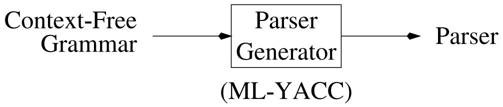
```
% cat x.txt

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
```

Syntactic Analysis

Regular Expressions - describe legal sequences of characters in a token.



Context-Free Grammars - describe legal sequences of tokens in a program.



Definitions

- *Language* - set of strings
- *String* - finite sequence of *symbols* taken from finite *alphabet*
  - Regular Expressions describe a language.
  - Context-Free Grammar also describes a language.

	Lexical Analysis	Syntax Analysis
language	set of tokens	set of source programs
string	token	source program
symbol	ASCII character	token

Context-Free Grammars

- Given by BNF expressions (Backus-Naur Form).
- Context-free grammars are more powerful than regular expressions.
  - Any language that can be generated using regular expressions can be generated by a context-free grammar.
  - There are languages that can be generated by a context-free grammar that cannot be generated by any regular expression.
- Examples:
  - Matching parentheses
  - Nested comments

Context-Free Grammars

- Context-Free Grammars consist of a set of *productions*.

$$symbol \;\longrightarrow\; symbol\ symbol\ ...\ symbol$$

- Symbol types:
  - *terminal* that corresponds to a token-type.
  - *non-terminal* that denotes a set of strings.
- Left-Hand Side (LHS) - *non-terminal*.
- Right-Hand Side (RHS) - *terminals* or *non-terminals*
- *Start Symbol* - A special *non-terminal*.
- Each production specifies how terminals and non-terminals may be combined to form a substring in language.
- Easy to specify recursion:

$$stmt \;\longrightarrow\; IF\ exp\ THEN\ stmt\ ELSE\ stmt$$

Start Symbol

- String of token-types is in language described by grammar if it can be derived from a given *start symbol*
- Derivations:
  1. begin with start symbol
  2. while non-terminals exist, replace any non-terminal with RHS of production

Example

Non-Terminals:

```
stmt      : Statement (Start)
expr      : Expression
expr_list : Expression List
```

Terminals (tokens):

```
SEMI ";"
ID
ASSIGN ":="
LPAREN "("
RPAREN ")"
NUM
PLUS  "+"
PRINT "print"
COMMA ","
```

$$
\begin{aligned}
stmt &\rightarrow stmt; stmt\\
stmt &\rightarrow ID := expr\\
stmt &\rightarrow PRINT\ (\ expr\_list\ )\\
\\
expr &\rightarrow ID\\
expr &\rightarrow NUM\\
expr &\rightarrow expr + expr\\
expr &\rightarrow (\ stmt,\ expr\ )\\
\\
expr\_list &\rightarrow expr\\
expr\_list &\rightarrow expr\_list,\ expr
\end{aligned}
$$

Example Derivation

Show that the following expression can be derived from start symbol.

```
ID := NUM; PRINT(NUM)
```


Another Derivation

The previous derivation is called *left-most* because the left-most non-terminal is always expanded first. We can also find a right-most derivation.
Show that the following expression can be derived from start symbol.

```
ID := NUM; PRINT(NUM)
```

Ambiguous Grammars

A grammar is ambiguous if it can derive a string of tokens with two or more different parse trees.

Non-Terminals:

```
expr : Expression
```

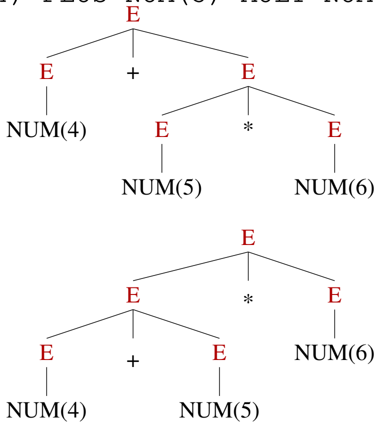
Terminals (tokens):

```
ID
NUM
PLUS  "+"
MULT  "*"
```

$expr \rightarrow ID$
$expr \rightarrow NUM$
$expr \rightarrow expr + expr$
$expr \rightarrow expr * expr$

Consider: $4 + 5 * 6$

`NUM(4) PLUS NUM(5) MULT NUM(6)`



Ambiguous Grammars

Non-Terminals:

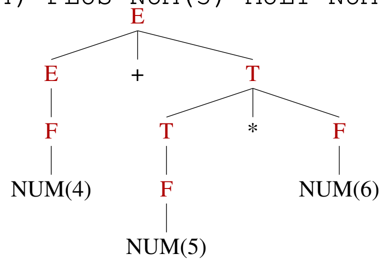
```
expr : Expression
term : Term (add)
fact : Factor (mult)
```

Terminals (tokens):

$expr \rightarrow expr + term$
$expr \rightarrow term$
$term \rightarrow term * fact$
$term \rightarrow fact$
$fact \rightarrow ID$
$fact \rightarrow NUM$

$4 + 5 * 6$

`NUM(4) PLUS NUM(5) MULT NUM(6)`



Parse Trees

- *Parse Trees* - Graphical representation of derivation.
- Each internal node is labeled with a non-terminal.
- Each leaf node is labeled with a terminal.
- Parse Tree of the example using right-most derivation production:

Ambiguous Grammars

- *Problem*: compilers use parse trees to interpret meaning of parsed expressions.
  - Different parse trees may have different meanings, resulting in different interpreted results.
  - For example, does $4 + 5 * 6$ equal 34 or 54?
  - Languages with ambiguous grammars are DISASTROUS! Programmers can't understand what their programs mean!
- *Solution*: rewrite grammar to eliminate ambiguity.
  - Operators have a relative *precedence*. We say some operands *bind tighter* than others. ("*" binds tighter than "+")
  - Operators with the same precedence must be resolved by *associativity*. Some operators have *left associativity*, others have *right associativity*.

End-of-File Marker

- Parse must also recognize the End-of-File (EOF).
- EOF marker in the grammar is “$”
- Introduce new start symbol and the production $E' \rightarrow\ E\$$

Grammars and Lexical Analysis

- Grammars can also describe token structure:

```
( a | b )* abb
```

$W \rightarrow aW$
$W \rightarrow bW$
$W \rightarrow aX$
$X \rightarrow bY$
$Y \rightarrow bZ$
$Z \rightarrow \epsilon$

- Can combine lexical analysis and syntax analysis into one module.
- Disadvantages:
  - Regular expression specification is more concise.
  - Separating phases increases compiler modularity.

Context-Free Grammar and REs

- Context-free grammars are more powerful than regular expressions.
  - Any language that can be generated using regular expressions can be generated by a context-free grammar.
  - There are languages that can be generated by a context-free grammar that cannot be generated by any regular expression.
- As a corollary, CFGs are strictly more powerful than DFAs and NFAs.
- The proof is in two parts:
  - Given a regular expression R , we can generate a CFG G such that L(R) == L(G).
  - We can define a grammar G for which there there is no FA F such that L(F) == L(G).

Context-Free Grammar and REs

Base Cases:

- Symbol (a):
  $RE \rightarrow\ a$
- Epsilon (ϵ):
  $RE \rightarrow\ \epsilon$

Inductive Cases:

- Alternation ($M|N$):
  $RE \rightarrow\ M$
  $RE \rightarrow\ N$
- Concatenation ($M\ N$):
  $RE \rightarrow\ M\ N$
- Kleen closure ($M*$):
  $RE \rightarrow\ M\ RE$
  $RE \rightarrow\ \epsilon$

Further Exploration

We have been talking about Context-Free Grammars.

What is a **context-sensitive grammar?**

Context-Free Grammar with no RE/FA

$$
\begin{aligned}
S &\rightarrow (S) \\
S &\rightarrow \epsilon
\end{aligned}
$$

- FAs have a FINITE number of states, N
- FA must “remember” number of “(”s, to generate “)”s
- At or before $N + 1$ “(”s FA will revisit a state.
- That state represents two different counts of “)”s.
- Both counts must now be accepted.
- One count will be invalid.

Representations

- Regular, right-linear, finite-state grammars: FAs
- Context-free grammars: Push-Down Automata