Outline

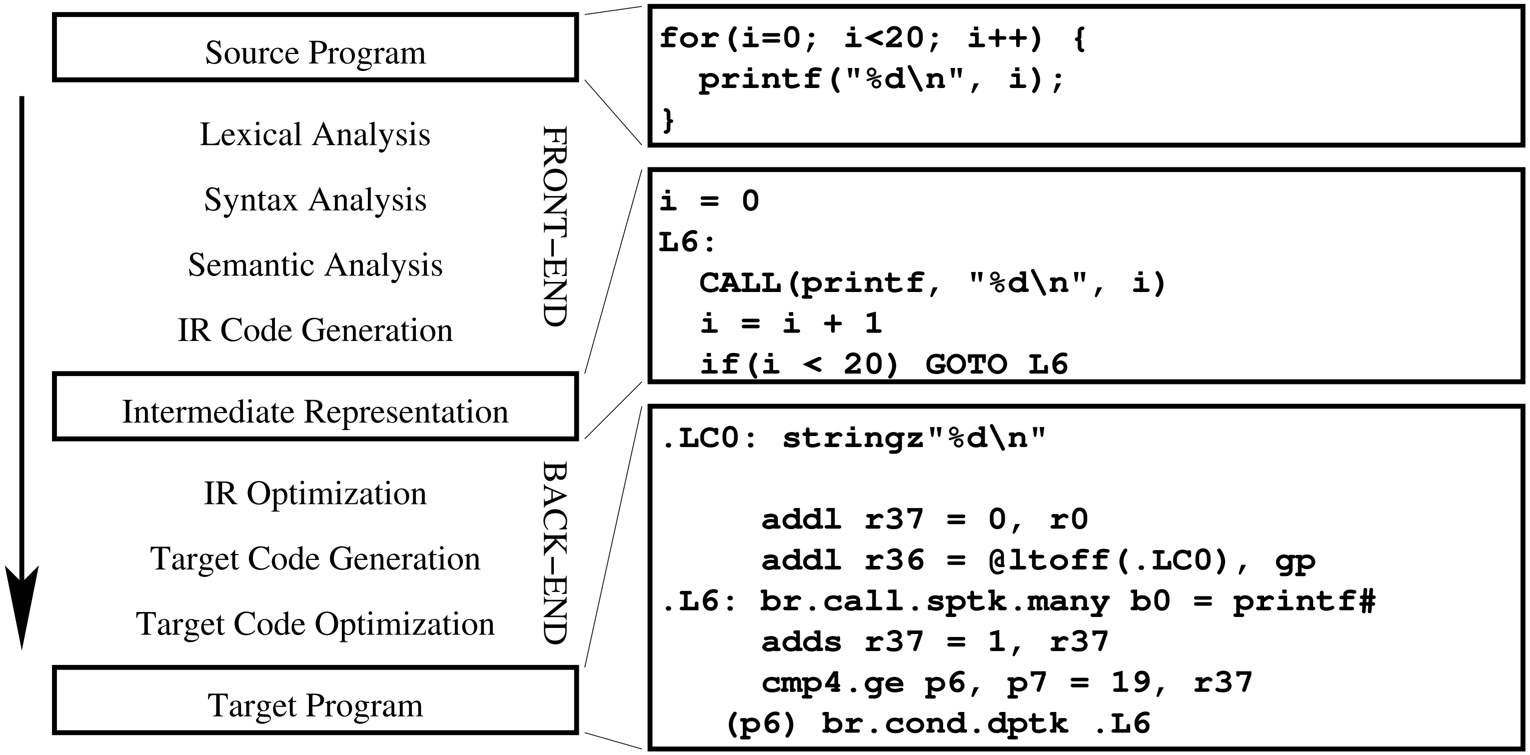
Last Week:

- Introduction to ML

Today:

- Lexical Analysis
- Homework: Read Chapter 2 of Appel

The Compiler



- Lexical Analysis: Break into tokens (think words, punctuation)
- Syntax Analysis: Parse phrase structure (think document, paragraphs, sentences)
- Semantic Analysis: Analyze meaning of programs; type check

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`	$ID(\texttt{foo})$
-100	$NUM(-100)$
10.45	$REAL(10.45)$

- White space and comments often discarded.

Lexical Analysis Example

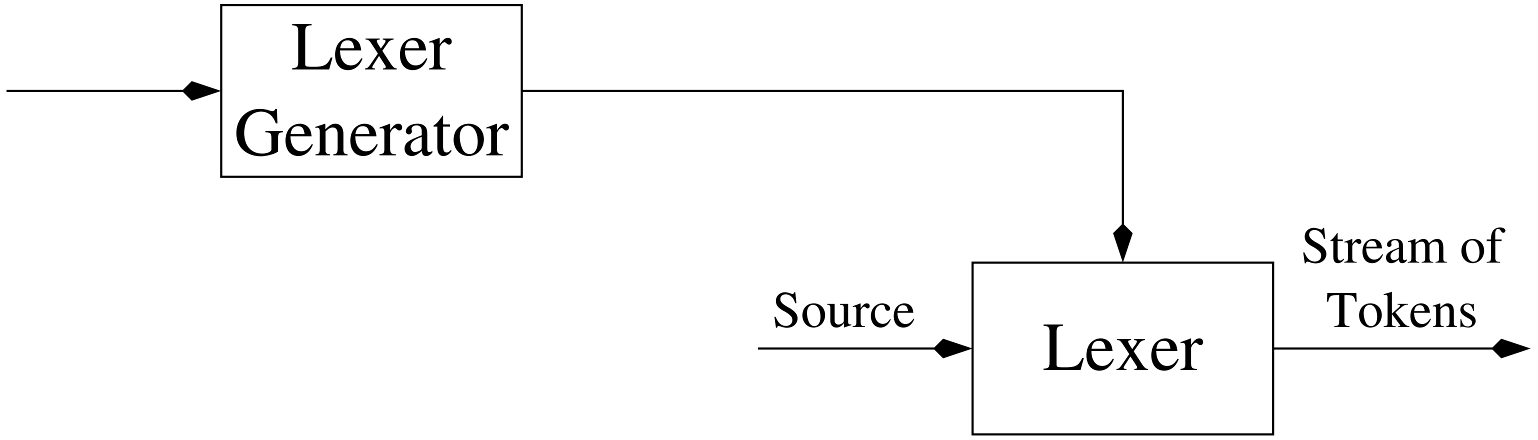
```
x = ( y + 4.0 );
```

Lexer Implementation

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

Implementation Options:

1. Write Lexer from scratch. (Boring and error-prone)
2. Use Lexical Analyzer Generator. (Quick, fun and easy!)



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

Using another language of course! We will use *Regular Expressions*
So the Lexer Generator is really another kind of compiler!

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`.

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

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

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

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

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

Regular Expression Examples

Ambiguous Token Rule Sets

- A single regular expression is a completely unambiguous specification of a token.
- Sometimes, when we put together a set of regular expressions to specify all of the tokens in a language, ambiguities arise:
  - i.e., two regular expressions match the same string
- For example:

Ambiguous Token Rule Sets

- We resolve ambiguities using two rules:
  - Longest match: The regular expression that matches the longest string takes precedence.
  - Rule priority: The regular expressions identifying tokens are written down in sequence. If two regular expressions match the same (longest) string, the first regular expression in the sequence takes precedence.

Lexer Implementation

Implementation Options:

1. Write Lexer from scratch. (Boring and error-prone)
2. Use Lexical Analyzer Generator. (Easy, quick and fun!)

- **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());
```

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

```
(* -*- 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

```
% 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
```

Implementing Lexers

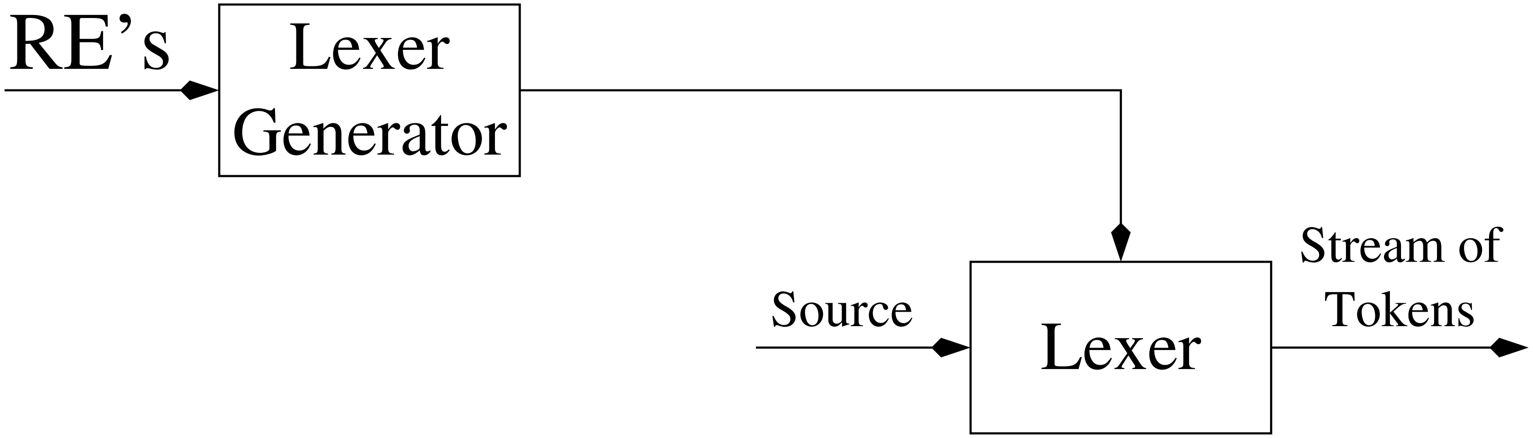
We can implement a lexer by using the following technique:
Programmer:

- Writes down *regular expressions* (RE) to specify the tokens.

Lexer generator:

- Convert regular expressions into a *non-deterministic finite automaton* (NFA).
- Convert NFA into a *deterministic finite automaton* (DFA).
- Convert DFA into executable program.

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

- Every regular expression can be systematically converted into a nondeterministic finite automaton
- Every finite automaton produced will have a particular structure that includes
  - an incoming edge labeled with a symbol.
  - a single final state.

RE to NFA Example

NFA to DFA Conversion

- NFA's are hard to implement directly.
- They require us to "guess" the right path.
- Solution: we will convert an NFA to a DFA
- Main Idea:
  - Avoid guessing by trying all possibilities simultaneously.
  - Each DFA state will remember all possible states we might be in in the corresponding NFA.

NFA to DFA Example

NFA to DFA Conversion

Basic Functions

- $edge(s, a)$ = All NFA states reachable from state s by traversing label a.
- $closure(S) = T$ All reachable NFA states from $s \in S$ by traversing one or more ϵ labels.

  Algorithm:

  ```
  T := S;
  repeat
    Told := T
    T    := Told U (U(s in Told) edge(s,e))
  until (Told = T)
  ```

- $DFAedge(D, a)$ = All reachable NFA states from $s \in D$ by traversing a and ϵ edges.

$$DFAedge(D, a) = closure\left(\cup_{s \in D}\, edge(s, a)\right)$$

DFA Representation

It is fairly straightforward to implement a DFA now:

- Rather than actually keeping track of sets of NDFA states, identify each NDFA state with an integer.
- Use a matrix to represent the state transition relation of the DFA.
  - integer states index columns
  - alphabet symbols index rows
  - entries of matrix are next state
- Use an array to map final states to tokens.
- Write a simple loop to implement the automaton reading one symbol after another.

See Appel 2.4 for details.

Recommended exercise: Implement the DFA on the previous slide in ML.

Longest Token

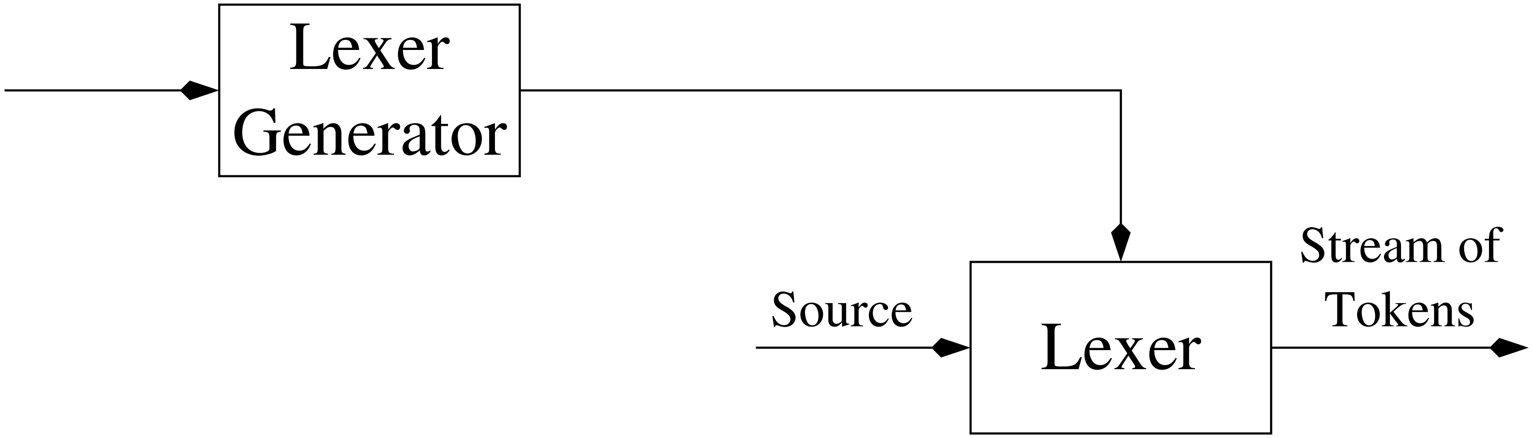
Lexer must find longest matching token.

```
ifz8 zz       ID, ID not IF, ID, ID
iff 17        IFF, NUM not IF, ID, NUM
```

- Start the automaton in the initial state.
- Save most recent final state and position in stream
- Update when new final state found
- Commit to the last final state found when the automaton cannot make any more valid transitions.
- Restart the automaton in the initial state at the last saved position.

Summary

A *lexer* is a program that takes ordinary text as an input and figures out the sequence of *tokens*.



We can implement a lexer by using the following technique:

- Write down regular expressions to specify the tokens.
- Convert regular expressions into NFA.
- Convert NFA into DFA.
- Convert DFA into executable program.

Homework

Read Chapter 2 of Appel.

Equivalent states:

- Eliminate redundant states, smaller FA.
- Recommended: Exercise 2.6.

Next time, we will look at software tools for lexer generation.