Outline

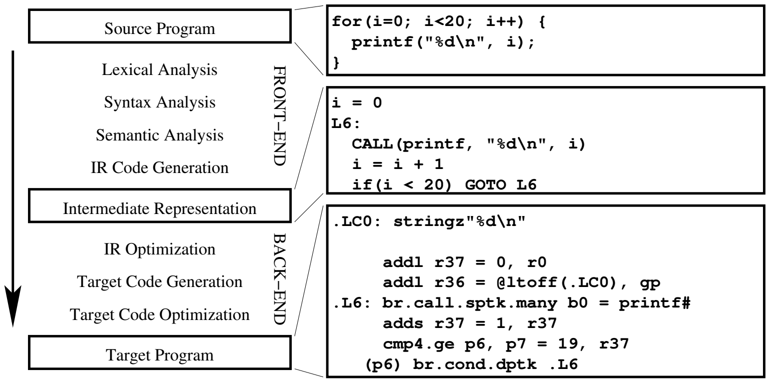
Last Week:

- Introduction to ML

Today:

- Lexical Analysis

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

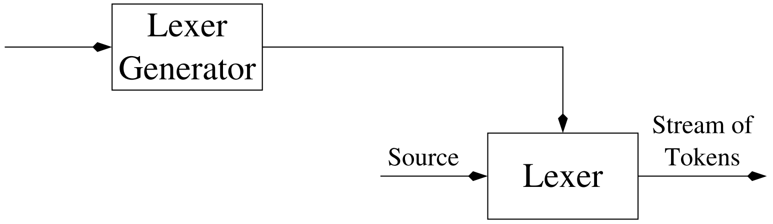
```
x = ( y + 4.0 );
```

Lexer Implementation

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!
So the Lexer Generator is really another kind of compiler!

Lexer Implementation Strategy

We can implement a lexer by using the following technique:

- Write down *regular expressions* (RE) to specify the tokens.
- Convert regular expressions into a *non-deterministic finite automaton* (NFA).
- Convert NFA into a *deterministic finite automaton* (DFA).
- Convert DFA into executable program.

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`.

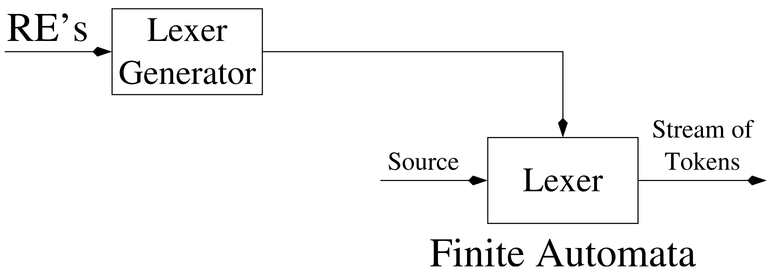
$$\texttt{(a | b)(a | c)} \rightarrow \{\texttt{aa, ac, ba, bc}\}$$

- 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 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)

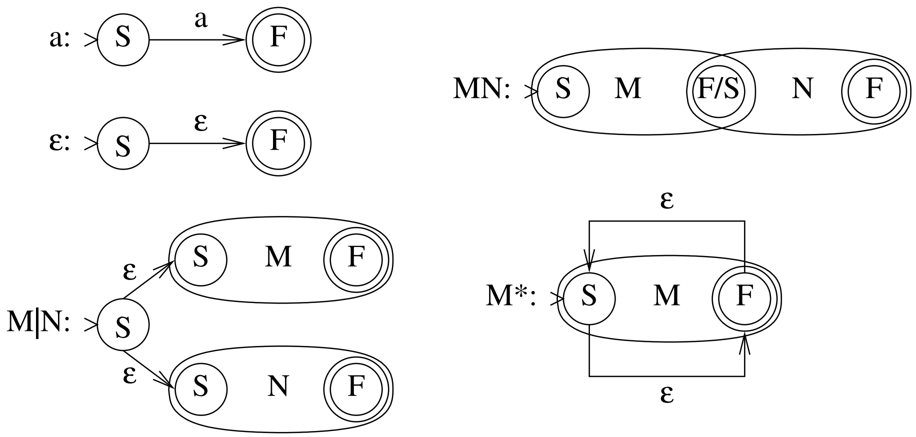
- Two or more edges leaving a node can be identically labeled.
- An edge can be labeled with ϵ.

Implementing Lexer:

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

NFA Example

RE to NFA Rules



RE to NFA Example

NFA to DFA Example

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

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.

DFA Representation

It is fairly straightforward to implement a DFA using standard techniques now:

- Use a matrix to represent the state transition relation of the DFA.
- Use an array to map final states to tokens.
- Write a simple loop to implement the automaton reading one symbol after another.

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

Homework

Read Chapter 2 of Appel.

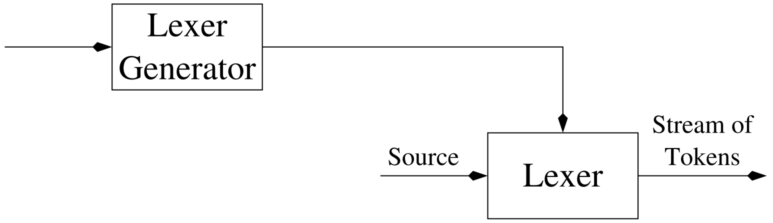
Equivalent states:

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

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

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.