
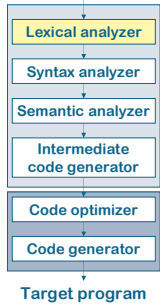


Lexical Analysis: Finite Automata

CS 471
September 5, 2007



Phases of a Compiler



Lexical Analyzer

- Group sequence of characters into lexemes – smallest meaningful entity in a language (keywords, identifiers, constants)
- Makes use of the theory of regular expressions and finite state machines
- Lex and Flex are tools that construct lexical analyzers from regular expression specifications

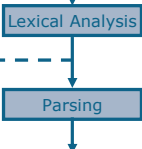
1 CS 471 – Fall 2007

Last Time: Tokenizing

- Breaking the program down into words or “tokens”
- Input: stream of characters
- Output: stream of names, keywords, punctuation marks
- Side effect: Discards white space, comments

Source code: `if (b==0) a = "Hi";`

Token Stream: `if (b == 0) a = "Hi" ;`



2 CS 471 – Fall 2007

Regular Expression Syntax

<code>[abcd]</code>	one of the listed characters (a b c d)
<code>[b-g]</code>	<code>[bcdefg]</code>
<code>[b-gM-Qkr]</code>	<code>[bcdefgMNOQkr]</code>
<code>[^ab]</code>	anything but one of the listed chars
<code>[^a-f]</code>	anything but the listed range
<code>M?</code>	Zero or one M
<code>M+</code>	One or more M
<code>M*</code>	Zero or one M
<code>"a.+*"</code>	literally a.+*
<code>.</code>	Any single character (except \n)

3 CS 471 – Fall 2007

Breaking up Text

`elseif=0;`

`else x = 0;`

`elseif = 0;`

- REs alone not enough: need rules for choosing
- Most languages: longest matching token wins
- Ties in length resolved by prioritizing tokens
- RE's + priorities + longest-matching token rule = lexer definition

4 CS 471 – Fall 2007

Lexer Generator Specification

- **Input** to lexer generator:
 - list of regular expressions in priority order
 - associated *action* for each RE (generates appropriate kind of token, other bookkeeping)
- **Output:**
 - program that reads an input stream and breaks it up into tokens according to the REs. (Or reports lexical error -- “Unexpected character”)

5 CS 471 – Fall 2007

Lex: A Lexical Analyzer Generator

- Lex produces a C program from a lexical specification
- (Please read the man page)

```

%%
DIGITS [0-9]+
ALPHA [A-Za-z]
CHARACTER {ALPHA}|_
IDENTIFIER {ALPHA}{CHARACTER}{DIGITS}*
%%
if {return IF; }
{IDENTIFIER} {return ID; }
{DIGITS} {return NUM; }
([0-9]+ "." [0-9]*) | ([0-9]* "." [0-9]+) {return REAL; }
. {error(); }

```

6 CS 471 - Fall 2007

Lexer Generator

- Reads in list of regular expressions R_1, \dots, R_n , one per token, with attached actions


```
-?[1-9][0-9]* { return new Token(Tokens.IntConst, Integer.parseInt(yytext())); }
```
- Generates scanning code that decides:
 - whether the input is lexically well-formed
 - corresponding token sequence
- Observation: Problem 1 is equivalent to deciding whether the input is in the language of the regular expression
- How can we efficiently test membership in $L(R)$ for arbitrary R ?

7 CS 471 - Fall 2007

Regular Expression Matching

- Sketch of an efficient implementation:
 - start in some initial state
 - look at each input character in sequence, update scanner state accordingly
 - if state at end of input is an *accepting state*, the input string matches the RE
- For tokenizing, only need a finite amount of state: (*deterministic*) *finite automaton* (DFA) or *finite state machine*

8 CS 471 - Fall 2007

High Level View

source code → Scanner → tokens

specification → Scanner Generator → Scanner

Compile time / Design time

Regular expressions = specification
Finite automata = implementation

Every regex has a FSA that recognizes its language

9 CS 471 - Fall 2007

Finite Automata

- Takes an input string and determines whether it's a valid sentence of a language
- A finite automaton has a finite set of states
- Edges lead from one state to another
- Edges are labeled with a symbol
- One state is the start state
- One or more states are the final state

10 CS 471 - Fall 2007

Language

Each string is accepted or rejected

- Starting in the start state
- Automaton follows one edge for every character (edge must match character)
- After n -transitions for an n -character string, if final state then accept

Language: set of strings that the FSA accepts

11 CS 471 - Fall 2007

Finite Automata

Automaton (DFA) can be represented as:

- A transition table

$["^"]^*$

	"	non-"
0		
1		
2		

- A graph

Implementation

```

boolean accept_state[NSTATES] = { ... };
int trans_table[NSTATES][NCHARS] = { ... };
int state = 0;

while (state != ERROR_STATE) {
    c = input.read();
    if (c < 0) break;
    state = table[state][c];
}
return accept_state[state];

```

RegExp → Finite Automaton

- Can we build a finite automaton for every regular expression?
- Strategy: consider every possible kind of regular expression (define by induction)

a

R_1R_2

$R_1 | R_2$?

Deterministic vs. Nondeterministic

Deterministic finite automata (DFA) – No two edges from the same state are labeled with the same symbol

Nondeterministic finite automata (NFA) – may have arrows labeled with ϵ (which does not consume input)

DFA vs. NFA

- DFA:** action of automaton on each input symbol is fully determined
 - obvious table-driven implementation
- NFA:**
 - automaton may have choice on each step
 - automaton accepts a string if there is *any way* to make choices to arrive at accepting state / every path from start state to an accept state is a string accepted by automaton
 - not obvious how to implement efficiently!

RegExp → NFA

$-? [0-9]^+$ $(-|\epsilon) [0-9][0-9]^*$

Inductive Construction

The diagram illustrates the inductive construction of NFA operations:

- R_1R_2** : Shows two NFA components, R_1 and R_2 , connected in sequence. A transition labeled 'a' goes from the start state of R_1 to its final state, which then connects to the start state of R_2 , ending at its final state.
- $R_1 | R_2$** : Shows two NFA components, R_1 and R_2 , connected in parallel. A single start state branches into two paths, one for R_1 and one for R_2 , both leading to their respective final states.
- R^*** : Shows an NFA R with a start state and a final state. Two new states are added: one before and one after R . Transitions labeled ϵ connect the new start state to the original start state, the original final state to the new final state, and a loop back from the new final state to the new start state.

18 CS 471 - Fall 2007

Executing NFA

- **Problem:** how to execute NFA efficiently?
"strings accepted are those for which there is some corresponding path from start state to an accept state"
- **Conclusion:** search all paths in graph consistent with the string
- **Idea:** search paths in parallel
 - Keep track of subset of NFA states that search could be in after seeing string prefix
 - "Multiple fingers" pointing to graph

19 CS 471 - Fall 2007

Example

- **Input string:** -23
- **NFA States**

The NFA has states 0, 1, 2, 3. Transitions are: 0 to 1 on '0', 1 to 0 on '1', 1 to 2 on '2', 2 to 3 on '3', and 0 to 1 on ϵ .

- **Terminology:** ϵ -closure - set of all reachable states without consuming any input
 - ϵ -closure of 0 is $\{0,1\}$

20 CS 471 - Fall 2007

NFA→DFA Conversion

- Can convert NFA directly to DFA by same approach
- Create one DFA for each distinct subset of NFA states that could arise
- States: $\{0,1\}, \{1\}, \{2,3\}$

The DFA has states $\{0,1\}$, $\{1\}$, and $\{2,3\}$. Transitions are: $\{0,1\}$ to $\{0,1\}$ on '0', $\{0,1\}$ to $\{1\}$ on '1', $\{0,1\}$ to $\{2,3\}$ on '2', $\{1\}$ to $\{2,3\}$ on '3', and $\{2,3\}$ to $\{2,3\}$ on '0'.

21 CS 471 - Fall 2007

DFA Minimization

- DFA construction can produce large DFA with many states
- Lexer generators perform additional phase of *DFA minimization* to reduce to minimum possible size

The DFA has states 0, 1, 2. Transitions are: 0 to 1 on '1', 1 to 0 on '0', 1 to 2 on '1', 2 to 1 on '0', and 2 to 2 on '0'.

What does this DFA do?

Can it be simplified?

22 CS 471 - Fall 2007

Automatic Scanner Construction

To convert a specification into code

1. Write down the RE for the input language
2. Build a big NFA
3. Build the DFA that simulates the NFA
4. Systematically shrink the DFA
5. Turn it into code

Scanner generators

- Lex and flex work along these lines
- Algorithms are well known and understood
- Key issue is interface to the parser

23 CS 471 - Fall 2007

Lexical Analysis Summary



- **Regular expressions**
 - efficient way to represent languages
 - used by lexer generators
- **Finite automata**
 - describe the actual implementation of a lexer
- **Process**
 - Regular expressions (+priority) converted to NFA
 - NFA converted to DFA

Next Time

- **C Primer (PA1) is due tonight**
- **Lexer for Tiger (PA2) coming up**
- **Next Week: Parsing**

